The End of GCode

Intelligent Modular Machines with Model Based Control

by

Jake Robert Read

B.Arch., University of Waterloo, 2016
M.Sc., Massachusetts Institute of Technology, 2022

Submitted to the Program in Media Arts and Sciences, School of Architecture and Planning
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

Massachusetts Institute of Technology

May 2026

© 2026 Jake Robert Read. All rights reserved.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Jake Robert Read
Program in Media Arts and Sciences
May 15, 2026

Certified by: Neil Gershenfeld
Director, MIT Center for Bits and Atoms, Thesis Supervisor

Accepted by: Joseph Paradiso
Academic Head, Program in Media Arts and Sciences

The End of GCode

Intelligent Modular Machines with Model Based Control

by
Jake Robert Read

Submitted to the Program in Media Arts and Sciences, School of Architecture and Planning on May 15, 2026 in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

ABSTRACT

The making, maintenance and use of digital fabrication machines is a core competency for industrial economies. However, innovation in this domain is hampered by the pervasive use of GCode: an antiquated interface between machine users and machine controllers. GCode only permits one-way data transfer and prevents us from smoothly crossing the boundary between high- and low-level planning and control tasks even though those tasks have deep physical connections. This makes it difficult to develop new machines, train new operators, and educate new machine designers. This thesis develops a feedback-native control architecture and applies it to machines that can think about what they are doing, providing new insights for their operators and designers.

The architectural contributions rebuild machine controllers as modular hardware and software systems at three layers: networks, programs, and frameworks. The Open Systems Assembly Protocol (OSAP) is a network for low-latency message passing, clock synchronization and device discovery across heterogeneous devices and data links. It establishes a distributed operating system that spans high- and low-level devices. To write programs for these systems I contribute PIPES (Programming in Piped EcoSystems) to combine dataflow for real-time operation with scripting for machine tasking; it enables global discovery and configuration of systems. Finally I contribute MAXL (Modular Acceleration eXecution Library) to describe and deploy core motion control routines as dataflow modules.

Articulating machine control as a constrained optimization task unlocks performance and simplifies workflows. My architectural contributions enable the development of machines that learn their own constraints by fitting models of their dynamics. I contribute a velocity planner that uses motion models directly to optimize machine speed and show that it out-performs the state-of-the-art and extend it using new models for 3D printing physics that are fit to almost any material or nozzle using computational metrology. I contribute a 3D printing workflow that is based on feedback from these physics rather than feedforward parameter tuning and deploy it on a modified state-of-the-art printer. Using this workflow, I can automatically operate the printer with performance that is comparable to or exceeds the state-of-the-art. This enables us to print in new materials, better understand how machine physics relate to machine performance, tune controllers directly against constraints, continuously improve models, and monitor errors.

To show that these contributions generalize across machines and processes I develop a milling machine using model-based motor selections and motion control. Operation of this machine shows that we can predict important control outcomes before we run jobs and measure cutting forces by combining model predictions with real world data.

With these contributions, machine controllers are no longer black boxes; they are distributed algorithms made of recognizable software design patterns and operated via physical models rather than tacit knowledge and heuristics. If they are adopted, we could enable the next generation of engineers and scientists to rapidly invent, modify and deploy novel machine systems as they work through the coming decades' pressing issues.

Thesis Supervisor: Neil Gershenfeld
Title: Director, MIT Center for Bits and Atoms

The End of GCode

by

Jake Robert Read

This dissertation has been reviewed and approved by the following committee members

Neil Gershenfeld
Director, MIT Center for Bits and Atoms
Massachusetts Institute of Technology

Nadya Peek
Associate Professor, Human Centered Design and Engineering
University of Washington

Jonathan Seppala
Director, nSoft Consortium
National Institute of Standards and Technology

Acknowledgments

It is day 3177/3177 of my time as a student at MIT. It has been a spectacular place to live and learn. My time here has transformed me in ways I could not have expected.

I have my advisor Neil to thank for bringing me here. Neil is a famous collector of high-quality misfits, and my peers at the CBA are representative of that. Thanks to them for the soup, and to Neil for bringing us together and building the mostly unstructured jungle gym that we get to mess around (and sometimes hurt ourselves) in, also for the state disturbances that ensure none of us fall into any local minima. Neil has routinely reminded me that if you are "making a nice box" for your research, you aren't doing it right. That's code for: don't finish your projects, do the next one. Against his advice I tried to do both, the result of which is clear by the day counter above and the page counter in this document.

To balance Neil's approach I am glad to have Nadya and Jon. Nadya is thoughtful and incisive, and I am grateful for her guidance especially in these last stages as I have tried to make sense of this block of work; if it weren't for her, I would have probably handed this in and still not understood what was really in it. In places where the thesis makes sense, it is her fault. Where it does not, the blame is on me. Jon I have always admired because he is basically a piece of scientific infrastructure. That might not sound glamorous, but people like Jon make the world go around. He has helped me to understand how modelling and measurements are managed and made, and to see how the scientific-industrial complex operates behind the scenes.

Life is long and if you are lucky you will accrue many mentors. I have had this fortune; Saul Griffith gets credit for helping me to understand early on that the world is something we can rebuild, if we would like to. I worked for Forrest Meggers as he was starting his lab and he has been a friend and mentor since then, although try as he may to explain, I still don't think I really understand entropy. Sam Calisch was basically my MIT big brother, he showed me that one can be cool and smart and kind all at the same time. Thanks also to the undergrads and students that I have had the chance to mentor, especially Samuel Schuur.

Work in this discipline is impossible without shop maintainers; they work harder than most and their efforts are invaluable; Zach Fredin, Anthony Pennes and Dan Gilbert.

I have had an excellent group of collaborators and colleagues as well. Quentin Bolsée and I have debated and discussed most of the topics in this thesis at length. Leo McElroy is a recurring (and wonderful) CBA interloper, and Ilan Moyer re-appeared at MIT close to the end of my time here. Alex Zimmer has been a stalwart friend since we met, and I expect that we will be at one anothers' weddings and funerals (whichever comes first). These are my favorite people to talk about machines with. Also at the weddings and funerals are Emily and Ben Klein, Devora Najjar, Adam Haar Horowitz, and Elizabeth Antczak, just to name a few. To the countless friends that I have made here and elsewhere, I miss you all.

I have Miranda Li to thank for keeping my spirits afloat in these last months; it is a brighter world when we have a project to work on together.

Behind everything else is my family. Thanks to Mom for believing in me the whole time, and to Dad for teaching me how the world of stuff works and letting me make messes in the garage.

Thanks above all to my Brother, whose friendship and steadfastness I can always rely on.

Colophon

Speaking of systems, this thesis was typeset using Typst (0.14.2) via Quarto (1.9.36) from source files written in Markdown.

I elected to use Quarto because it also compiles to HTML; for a document of this length that covers many interconnected topics, the online version will likely provide a better reading experience. Also, it has videos. I will maintain links to the online version (and a downloadable copy of this PDF) at the following URLS:

ekswhyzee.com/the-end-of-gcode

jakeread.github.io/thesis

Table of Contents

Acknowledgments 4
Colophon 5
List of Figures 12
List of Tables 30
List of Listings 32
1 Introduction 35
1.1 A Small GCode Primer 37
1.2 A (Very) Small CAM Primer 41
1.3 Problems in State-of-the-Art Machine Control 42
1.3.1 Machines are Thoughtless 42
1.3.2 Machines are Poorly Represented 43
1.3.3 GCode Interpreters Obscure Important Control State and Data 45
1.4 Systems Integration and the Partitioning Problem 47
1.4.1 The Economic Partitioning Problem 48
1.4.2 The Technical Partitioning Problem 50
1.5 Machine Operation is a Constrained Optimization Problem 53
1.6 Key Questions and Contributions 55
1.6.1 Guiding Questions 56
1.6.2 Contributions in Mechatronic Systems Architecture 57
1.6.3 Contributions in Machine Control 59
1.6.4 Contributions in Machine Modelling and Workflow Development 60
2 Systems Integration for Modular Machine Control 61
2.1 Introduction 61
2.1.1 Machine Building Needs Flexible Network Architectures 62
2.1.2 Machine Building Needs Flexible Software Architectures 64
2.1.3 Chapter Overview 65
2.2 Background in Systems Integration 69
2.2.1 Steps Required to Understand and Extend a GCode Interpreter 70
2.2.2 Background on Networks 75
2.2.3 Background on Distributed Programming Models 79
2.2.4 Background on Machine Control Architectures 84
2.3 OSAP: Machine Systems Interconnect 90
2.3.1 Runtime 91
2.3.2 OSAP Services 103
2.4 PIPES: a Programming Model for Partitioned Systems 109
2.4.1 Runtimes are Schema Sources, not Sinks 109
2.4.2 Functions as the Basic Building Block 111
2.4.3 Types and Serialization 118
2.4.4 PIPES' Systems Programmer's Model: Scripted Dataflow 120
2.5 MAXL: Motion Control with Dataflow 125
2.5.1 MAXL's Operating Principle 125
2.5.2 The MAXL Gap 127
2.5.3 Synchronized Basis Splines 129

2.5.4 MAXL Blocks 131
2.6 Comparisons to Other Systems Integration Architectures 139
2.6.1 Other Distributed Systems and PIPES / OSAP 139
2.6.2 Systems Integration from Related Model-Based Control Research 147
2.6.3 Comparisons to Other Machine Control Architectures 148
2.7 Deploying OSAP, PIPES and MAXL 150
2.7.1 Architectural Deltas and Results 150
2.7.2 Bootstrapped Systems Assembly 151
2.7.3 From GCodes to Code 159
2.7.4 Reconfiguration of Systems for Model-Building and Deployment 164
2.7.5 Reconfiguration of Modules for Machine-Building 167
2.7.6 Visibility of Internal Control States 169
2.7.7 Time Synchronized Sensing and Data Collection 170
2.7.8 Enabling the Application of Large Compute in Machine Control 172
2.7.9 Timing Performance Analysis 172
2.7.10 Enabling ad hoc Systems Development 181
2.8 Discussion 185
2.8.1 Key Architectural Limits 185
2.8.2 Simpler Representations for Composable Systems 186
2.8.3 Is OSAP an (RT)OS? 187
2.8.4 OSAP and the OSI Model 188
2.8.5 GCode, Timing and the Partitioning Problem 188
2.9 Future Work 190
2.9.1 Improvements to OSAP's Runtime and Structure 190
2.9.2 Expanding PIPES' Reach 192
2.9.3 Improvements in Systems Integration Tooling 193
2.9.4 Redistributing MAXL 196
3 Modular Circuits for Machine Control 198
3.1 Introduction 198
3.1.1 Challenges in Modular Hardware Design 200
3.2 Knuckles Devices 201
3.3 Knuckles Backpacks 203
3.4 Knuckles vs. NICs 209
3.5 Knuckles Hubs 210
3.6 Using Knuckles 210
3.6.1 Device Re-Use 210
3.6.2 Retrofitting Machines 215
3.7 Discussion 221
3.7.1 The Hardware Partitioning Problem 221
3.7.2 About Power Routing and Delivery 221
3.7.3 About Hackable Systems 222
3.8 Future Work 222
4 Motion Control with Machine Models 224
4.1 Introduction 224

4.1.1 Key Limits to Classical Velocity Planning ... 225
4.1.2 Chapter Overview ... 226
4.2 Background in Motion Control ... 229
4.2.1 Essential Physics for Motion Control ... 229
4.2.2 Sophisticated Physics for Motion Control ... 229
4.2.3 The Role of PIDs and Hierarchical Control ... 235
4.2.4 Trapezoidal Planners ... 236
4.2.5 Geometry Smoothing and Filter-Based Methods ... 238
4.2.6 The State-of-the-Art in Industrial Motion Control ... 239
4.2.7 Model-Based Control in Robotics ... 242
4.2.8 Integrating Model-Based Control with Machines ... 245
4.2.9 Comparisons to This Work ... 246
4.3 Models for Motion Control ... 249
4.3.1 Motor Models ... 249
4.3.2 Kinematic Models ... 252
4.3.3 Combining Models for Constraint-Based Solving ... 254
4.4 Field-Oriented Control on (almost) Any Stepper ... 258
4.4.1 The Motor's Software and Data Interfaces ... 262
4.5 Fitting Motion Models ... 265
4.5.1 Fitting Motor Models ... 265
4.5.2 Fitting Kinematic Models ... 268
4.6 Optimization Based Velocity Planning ... 270
4.6.1 Optimization Formulation ... 270
4.6.2 Pipelining the Planner ... 278
4.6.3 Soft Functions in the Planner ... 279
4.7 Results ... 282
4.7.1 Trapezoids vs. Model-Based Control ... 282
4.7.2 Model-Fit Quality ... 285
4.7.3 Continuous Improvement of Models ... 285
4.8 Discussion ... 289
4.8.1 Machining with Trapezoidal vs. Model-Based Velocity Planning ... 289
4.8.2 Solving in *Phase Space* ... 290
4.8.3 About Performance and Hyperparameters ... 293
4.8.4 Notes on Motor Models and Motor Heterogeneity ... 293
4.8.5 Key Limits ... 295
4.9 Future Work ... 297
4.9.1 A Better Representation for Δt and $\vec{a}_{\text{req}}$ ... 297
4.9.2 Exploiting Time-Varying Motor Dynamics ... 298
4.9.3 The Missing Models for Motion ... 298
5 Model-Based Control of FFF 3D printers ... 301
5.1 Introduction ... 301
5.1.1 Feedforward FFF Configuration and Operation is Lossy and Limits Machines and Operators ... 302
5.1.2 Chapter Overview ... 304

5.2 Background in FFF 3D Printing 306
5.2.1 Fundamentals of FFF 3D printing 307
5.2.2 Parameters in the State-of-the-Art FFF Workflow 309
5.2.3 Physics Management in the State-of-the-Art FFF Workflow 315
5.2.4 Related Research in FFF 319
5.3 A Feedback-Based FFF Workflow 326
5.4 Hardware for Feedback-Based FFF 330
5.4.1 The RheoPrinter 330
5.4.2 The FrankenPrusa 331
5.5 Models for FFF Flow 334
5.5.1 How Flows are Modelled in This Work 334
5.5.2 Fitting Steady-State Flow 336
5.5.3 Fitting Isothermal Flow and Compression 338
5.5.4 Interpolating Between Steady-State and Isothermal Operation 340
5.6 Models for Extruder Motors 343
5.7 Models for FFF Cooling 344
5.7.1 Volumetric Heat Capacity 345
5.7.2 Layer Cooling Time 348
5.8 High Level FFF Planning 351
5.8.1 Selecting Flowrates using Models and Heuristics 351
5.8.2 Applying Acceleration and Jerk Heuristics 352
5.8.3 First Layer Speed 354
5.8.4 Time-Optimal Nozzle Temperature Selection 354
5.9 Real-Time FFF Planning 359
5.9.1 Optimization Formulation 359
5.9.2 Extending the Cost Function, Update, and Output Steps 364
5.9.3 Connecting the Planner to Hardware 364
5.10 Evaluating Feedback-Based FFF 366
5.10.1 Printing with the Model-Based FFF Workflow 366
5.10.2 Tables of FFF Prints 384
5.10.3 Reducing Parameter Spaces 386
5.10.4 Print Speed vs. SOTA 387
5.10.5 Qualitative Print Quality vs. SOTA 388
5.10.6 Flow Model Fit Quality 390
5.11 Discussion 397
5.11.1 Learning from Printer Data Outputs 398
5.11.2 Model-Based Knobs 401
5.11.3 Learning from Flow Models 406
5.11.4 Using a Full Range of Possible Nozzle Temperatures 408
5.11.5 Exploiting Time-Varying Printing Dynamics 408
5.11.6 Resilience to Human Errors 412
5.11.7 Key Limits 412
5.12 Future Work in FFF 419
5.12.1 Properly Combining Flow and Thermodynamic Models 419

5.12.2 Continuous Improvement and Deployment 420
5.12.3 Smaller Improvements to the System 421
5.12.4 Pareto Frontier Printing 423
6 Model-Based Development and Use of a CNC Milling Machine 425
6.1 Introduction 425
6.1.1 CNC Physics 426
6.1.2 The State-of-the-Art CNC Workflow 428
6.1.3 Interaction of Cutting Parameters and Acceleration Control 429
6.1.4 The State-of-the-Art in Model-Based CNC 430
6.2 Machine Design Tools 432
6.2.1 Virtually Comparing Motor and Machine Dynamics 433
6.2.2 Selecting Motion Control Parameters using Models 433
6.3 Toolpath Inspection Tools 434
6.3.1 Visualizing Real Chipload 434
6.3.2 Cutting Force Estimates 435
6.4 Results and Discussion 437
6.4.1 Hardware Developed and Tested 437
6.4.2 Model-Based Selection of Motors and Motion Parameters for CNC 437
6.4.3 Machining with Models 444
6.4.4 Chip Load Deviation and Cutting Force Estimates 449
6.4.5 Summarizing Discussion 451
6.5 Future Work 453
6.5.1 More Instrumentation and Faster Networks 453
6.5.2 Predicting Cutting Forces 454
6.5.3 Intelligently Selecting Feeds, Speeds, and Engagement Areas 455
6.5.4 CNC Machining as an Explicit Constrained Optimization Problem 455
6.5.5 Generating Useful Outputs for Machinists 456
7 Discussion 457
7.1 Crossing the Gap: Partitioning Machines 458
7.2 Wait, it's all Models? 460
7.2.1 As a Core Representation for Systems Integration 460
7.2.2 Computational Metrology 461
7.2.3 Over Machine Lifespans 465
7.2.4 But Leave Room For Heuristics 466
7.2.5 With Regard to Economic Partitions 467
7.3 Wait, it's all Connected? 469
7.4 Basis Splines and Systems Assembly 470
7.4.1 Spline Errors and Motion Constraints 470
7.4.2 Splines and Systems Integration Constraints 471
7.4.3 Constraints-Based Spline Configuration 473
7.5 Open vs. Closed Loop Motion, and Why Stepper Motors 474
7.6 About GPUs for Motion Control 475
7.7 The Connection to "AI" and Reinforcement Learning 476
7.7.1 Connections to Path Planning 476

8 Future Work 478
8.1 Developing A Spec for Machine Digital Twins 478
8.1.1 Discovering Kinematics 479
8.2 End-to-End Fitting of Models 480
8.3 An Open Ecosystem for Machine Control 480
9 Conclusion 483
9.1 What is Possible Now that Wasn't Before? 483
9.2 Feedback Systems Assembly 484
9.3 Closing Note: Imaginary Worlds 484
Bibliography 486

List of Figures

Figure 1.1 Machine systems that were originally developed to toil in factories are escaping: we find them now in the classroom, the laboratory, and in small prototyping firms. In each of these contexts we need controllers developed for flexibility, adaptability, and inspectability. 36

Figure 1.2 The actions carried out by the machine based on the GCode snippet from Listing 1.1. 37

Figure 1.3 The first computer-controlled CNC machine, the 'whirlwind,' was developed at MIT around the same time that programming languages and compilers were being invented. 37

Figure 1.4 Here I compare layers of abstraction in computer systems (at right) to machine workflows (left), spanning high to low level descriptions of operation. 38

Figure 1.5 Just a feed-forward instruction following automaton, when it could be so much more! 42

Figure 1.6 In this thesis I develop computational representations of machines that let us simulate them in software. This frame shows a 3D printer (left) and it's digital twin (center, top), alongside time series data (right) of key process variables. 44

Figure 1.7 This figure shows velocity over time for the GCodes generated by a CAM tool (which requests instantaneous changes in velocity), vs. the actually realized speeds as adjusted by the machine's controller. 46

Figure 1.8 Here I show the actual velocities of two toolpaths *after* they have been transformed by a trapezoid-based velocity planner (see Section 4.2.4). At the left, we target $100mm/s$ and on the right, $500mm/s$; in both cases, acceleration is limited to $1000mm/s^2$. The histograms (top row) render the actual velocity distributions of each job. 46

Figure 1.9 Machine control tasks can be organized by the *time scales* that each task operates within: high level tasks occur in days, hours, and minutes, whereas low level realtime tasks must be done in the span of seconds, milliseconds and microseconds. 50

Figure 1.10 Machine operation can be described as a constrained optimization task that spans the same layers as in Figure 1.9. Here, optima are described at a high level but constraints emerge from low-level physics. GCode makes it difficult to connect these two ends of the problem. 54

Figure 2.1 Here I show the FFF printer used in Chapter 5. The system is composed of multiple circuits, each with a particular task. The systems assembly challenge is to make the assembly of devices into a coherent controller; PIPES is the programming model used for this, and other machines in this thesis. MAXL is the motion control framework. 61

Figure 2.2 A simplified OSI layering, including OSAP (network layers), PIPES (presentation layer) and MAXL (an application built on top of PIPES). Each layer presents well-defined software APIs to surrounding layers in an effort to make components of the system easy to interchange. 65

Figure 2.3 The specification that defines what the `M3` GCode will do and what its expected parameters are, from the RepRap GCode Table — a community resource that is maintained by hand by volunteers. 71
Figure 2.4 Two OSAP Runtimes `A` and `B` each containing a local program and program functions that are wrapped as Pipes Functions. Link gateways and their indices are "below" the runtime, with functions attached via ports "above." Each of these have indices, which are used in packet routing. Two routes are shown and routing instructions for each are shown in the text below. 97
Figure 2.5 Here I show three runtimes with two routes, one of which uses an intermediate runtime being used purely as a message passing device. A unique property of OSAP is that runtimes and "routers" are not distinct devices. 98
Figure 2.6 These plots show data from two clock synchronization tests on the FrankenPrusa (5.4.2). At left, 120 seconds of sync without hot-starting the clock skews. At right, the first two seconds of sync with hot-start. The first rows plots clock errors in μs, second rows are a y-axis zoom on the *devices'* own estimates of their clock errors, and the third plots skew control terms. The bottom row plots round trip times from the test script (in Python) out to devices. 105
Figure 2.7 The same plot as in Figure 2.6, with data from the milling machine. 107
Figure 2.8 Functions tagged with `PIPES_FUNC` in Listing 2.16 are wrapped by PIPES and source their data schema and network interactions, to produce systems-wide programming models. 111
Figure 2.9 Diagram of the Pipes Function state machine with its user-defined implementation (center), typed input and output buffers and control logic. These embed the function into networks, allowing it to be called directly using simple RPC semantics or used as a component in a dataflow compute graph (or both). 112
Figure 2.10 A view of seven Pipes Functions whose inputs and outputs are "woven" together using Pipe `argmap` configurations. In this example a motion control planner's output from `on_new_pt` (at left) is a tuple of `(time, x, y, z, e)` control points. That is mapped into a series of OneDOF MAXL blocks (see Section 2.5.4.2), each of which is applied to one axis only: the maps are e.g. `(0, 1) -> (0, 1)` for the second OneDOF (in x) and `(0, 3) -> (0, 1)` for the third OneDOF (the z axis). 115
Figure 2.11 Four MAXL blocks, each of which is implemented as a Pipes Class containing Pipes Functions. Only some of these functions are used directly in dataflow graphs while others are used in scripts via remote procedure call, although any function can be used in both ways. 116
Figure 2.12 The PIPES / MAXL system that runs the FrankenPrusa (5.4.2). In this case the system is rendered in a configuration where only parts of the underlying system are in use, to perform a test. 121
Figure 2.13 Using time-based representations for motion that *include* the results of lower level machine controllers (velocity planners) allows us to more easily add components to (or align data with) those trajectories. For example to synchronize the on/off states of the LEDs used for this light painting from the original MAXL paper, the LED controller needs to know exactly *when* it should flip those switches. 126

Figure 2.14 A MAXL dataflow graph, overlaid in PIPES. Top left is the timer, which issues time stamps that stream through motion components. Here we have a velocity planner (left, as "solver"), multiple OneDOF blocks, a CoreXY transform and a bed levelling utility. At right, one of the machine's motors is shown with the basis spline interpolator. A representation of the network is also shown, the `prs_rtr_a` is a message passing device, dashed lines represent network segments. 127
Figure 2.15 The MAXL *gap* is key to the scheme's operation. The timer block generates timestamps that lead the system clock by this gap, which leaves enough time for values that are computed on top of that timestamp to reach the devices that will interpolate them. This figure diagrams MAXL blocks in green, which are spread across three runtimes (in blue), and connected across two network links (in yellow). Buffers of control point history can be stored in time-series to recreate trajectories seen by whichever block chooses to store them. I also show sensor data flowing back through the same system and being added to a synchronized stream of samples. 128
Figure 2.16 MAXL uses basis spline interpolation as a generic intermediate representation for motion. In this figure, I render a subset of spline control points that were generated by MAXL to stream to motors during a print job. At left is a velocity-planned trajectory segment in 3D rendered via its control points alone. At right: A 2D subsection of a basis spline, showing a stream of control points ($PA \rightarrow PG$) and the resulting curve. 130
Figure 2.17 Basis splines can be interpolated through position, velocity, acceleration, and jerk values for the curve that they represent. Control points are defined by their position only, the other derivatives are implied by the spline formulation itself. For example the solid section of this curve (in red) lies primarily between $PC \rightarrow PD$, but is defined by points $PB \rightarrow PE$. All four points are combined to define the curve shape, its velocity, etc.; moving PE to the right would increase the velocity throughout this segment. New points can be freely added to the tail of these curves, which is a useful property for defining continuous motion. For example the next curve segment (in blue, dashed) is defined by $PC \rightarrow PF$ and so-on. 130
Figure 2.18 The timer block (left) is the single source of any MAXL graph, and the interpolator (right) is the final sink; there can be many interpolators. 131
Figure 2.19 The model-based velocity planning solver interface, bed leveller, and CoreXY MAXL blocks. 132
Figure 2.20 CoreXY motion systems use two motors (a and b) in parallel to control cartesian (x and y) motion systems. In this render of the RheoPrinter (Section 5.4.1), a and b belt paths are rendered in blue and red. 132
Figure 2.21 Kaos kinematics are similar to those in CoreXY. 133
Figure 2.22 The `little guy` machine, and its nonlinear kinematics where motion of the tool-tip in cartesian space is composed of arcs traversed by the machine's two arms, which are connected to one another with rotary joints. 133
Figure 2.23 While 3D printer performance and reliability depends on their beds being well aligned to their motion systems, this is often not the case. We can apply mathematical corrections to account for these offsets. 134

Figure 2.24 To integrate embedded devices into middleware-based distributed systems, a *bridge* architecture is used. Here, embedded devices are represented to the middleware using virtualized interfaces that run a separate networking "agent" which communicates to the device over an out-of-band network link connected to a "client" in the device. 141
Figure 2.25 A comparison of middleware-based architectures and OSAP / PIPES, with a colour coding for each component's approximate relation to the OSI model; I collapse transport and data / presentation layers here. In middlewares, networking is hidden by the shared data layer whereas OSAP and PIPES expose networking within the program configuration itself using source routes. 142
Figure 2.26 Top row: terminal interfaces to the Prusa: all the electrical lines (left) and one of the motors, onto which we attach a diametrically magnetized disk magnet which serves as an interface to our FOC Stepper's encoder (Section 4.4). Second row: the Prusa's hotend with new controllers mounted and hardware interfaces labelled, front and back. 152
Figure 2.27 This image includes (from left to right) a Backpack module, the HBridge Knuckle with exposed backpack interface, the HBridge with the USB Dually Backpack mounted and finally the HBridge with the UART/RS485 Backpack 153
Figure 2.28 Calibrating the load cell in the FrankenPrusa. 155
Figure 2.29 For many simple modelling and development tasks, using RPCs alone in simple scripts that contain only one or two devices is often sufficient. The PIPES SOM is still useful in these cases as it contains function signatures that can be used to write proxies for RPCs. Other existing devices can also be useful in these cases; for example the Knuckles hub (Section 3.5, centered in this SOM) contains power management circuitry that can remotely reset or disable devices that are under test. 155
Figure 2.30 A screenshot from my Python IDE where the bed levelling routine is defined. This shows steps to acquire the load cell hardware / software object, tare it, configure its internal comparator, and connect it as a graph component as a virtual endstop. .161
Figure 2.31 The PIPES / MAXL layout of a chirp test. The system is configured to instantiate *chirp* generators in front of each axis. These are used to drive the axes-under-test, and time-series data is collected from the motor (right) into a dataset collector (left). 165
Figure 2.32 The FrankenPrusa's (5.4.2) planner (highlighted in orange) makes up one component in that machine's PIPES graph 2.4. This view is centered on the Python components in the machine, hardware is represented out of frame. 166
Figure 2.33 Cap modular reconfiguration allows us to use the same hardware (in this figure, a motor driver) to drive many systems: a CNC milling machine (left) 3D printer (middle, the printer's extruder motor), or (at right) in less serious explorations like the Blair Winch project (in Section 2.7.10). 167
Figure 2.34 Core networking components we re-used as well, this is a hub device deployed in the RheoPrinter (at left) and on the Little Guy (right, integrated on the underside of the machine). 167

Figure 2.35 A series of plotters developed by participants in a workshop that I co-hosted at the CBA with Ilan Moyer and Leo McElroy. Participants were each experienced machine builders, but not experienced controls engineers. We were able to use MAXL to develop working kinematic models for each of these machines. 169
Figure 2.36 On the left, I show a speed histogram and heatmap for a print layer with infill in the Y axis (where it is misaligned with the machine's "preferred" axis for high acceleration), and at the right, the same system with infill aligned on X (where it is aligned). 169
Figure 2.37 Collecting time-synchronized data streams from many sources was a key capability for other work in this thesis. This frame shows (at right) data from the FrankenPrusa's velocity planner (Section 5.9) that is sent to hardware (and stored) with timestamps that are synchronized with the clocks in each of the machine's modules (from Chapter 3). Planner data and predictions are in blue, and sensor readings are in orange. These are additionally synchronized with frames from the printer's microbolometer (a small thermal camera), one frame of this data stream is in the lower-left corner of this figure. 170
Figure 2.38 How to assemble a controller that spans GPU and CPU, big OS, and embedded devices? 172
Figure 2.39 Round-trip times for clock configuration request messages between a Python script analyzing their clock synchronization performance and devices in the CNC milling machine (above) and the FrankenPrusa (below). 176
Figure 2.40 Quentin plays a CNC Xylophone interactively, using computer vision to track the position of his fingers above a keyboard so that the Xylophone can slew towards the anticipated note *before* he hits the key, which then sends a MiDi packet through his interstitial controller that is subsequently connected to the machine systems developed in this chapter. 181
Figure 2.41 A light painting, drawn with a cable-driven "Blair Winch" machine using controllers from this thesis. 184
Figure 2.42 At left, the computer controlled winch from the Blair Winch Project, and at right, a diagram of the control scheme for controlling the orb's position using cable tension. 184
Figure 2.43 This is a screenshot from the development environment UI that I implemented during my masters thesis. The system was functionally similar to work presented in this thesis, albeit of lower quality systems design. The graph visualizer was nonetheless a promising tool, allowing users to coordinate data-flows across all layers in the system. 194
Figure 2.44 In this demo, I show a prototype graph interface to the PIPES system, which lets us discover and then connect an embedded device (an accelerometer) to a software block (an orientation viewing tool). Using a tool like this, programs can be configured and visualized *while* they are running, enabling a tight workflow between systems development and testing. 196
Figure 2.45 Here I show configuration of dataflows across more complex network graphs, including live discovery of devices and dataflows. 196

Figure 3.1 I have been developing this reusable kit of modular circuits and firmwares over the course of my time at MIT. With the most recent set of these (pictured here), I focused on re-usability of individual circuits. 198
Figure 3.2 This is a "monolithic" motion control board in a Prusa 3D printer. Monolithic boards are designed to cover a range of machine designs, but can be difficult to adapt to more complex systems where i.e. the number of motors required by the machine exceeds the number of motor drivers installed on the board… 199
Figure 3.3 The Field-Oriented Control Stepper (4.4), which uses an ATSAMD51 mcu with two DRV8251A H-Bridges and an AS5047 magnetic encoder. The board (top and bottom) at left, and set up on the RheoPrinter (5.4.1) at right. 201
Figure 3.4 The HBridge Knuckle, which uses an ATSAMD21 mcu with the DRV8251A hbridge. The board (top and bottom) at left, and set up to control a hobby servo (right). 201
Figure 3.5 The Loadcell Amplifier Knuckle, which uses an ATSAMD21 mcu with the CS5311 wheatstone bridge ADC. The board (top and bottom) at left, and set up in the RheoPrinter (5.4.1) (to measure extrusion pressure) at right. 201
Figure 3.6 The Accelerometer Knuckle, which uses an RP2354A mcu with the LSM6DSV16X inertial measurement unit. The board top (left) and bottom (center), and set up in the FrankenPrusa (5.4.2) at right. 202
Figure 3.7 The *"Deadbugger"* Knuckle, which uses an ATSAMD21 mcu. The board (top and bottom) at left, and set up to control a hobby servo (right). 202
Figure 3.8 These are an early backpack design, showing a motor driver (orange), and backpacks for: Full-Duplex UART over RS485 (at left, the same link technology as I use extensively in this thesis, and in Figure 3.10), Half Duplex UART over RS485 (right, top), and a daisy-chained UART architecture (right, bottom). 203
Figure 3.9 Left: a backpack with two 2x3 leaf spring style connectors (black with gold contacts) and through-hole surface-mount standoffs (zinc-plated cylinders). At right, the HBridge Knuckle with the reciprocal interface; GND and VCC connect via contact with the surface-mount standoffs (underneath these holes are threaded standoffs, an M2 screw bolts them together), and leaf springs touch down on exposed pads. A complete pinout is in Table 3.1. The catellated pins (towards the bottom of the figure) are a second interface for programming and debugging of the module, this is not part of the backpack interface. 203
Figure 3.10 The UART over RS485 Backpack, bottom (left) and top (right). 207
Figure 3.11 The USB Dually Backpack, bottom (left) and top (right). 207
Figure 3.12 NICs (right) vs. Firmware-Defined Link Drivers (left) 209
Figure 3.13 The Knuckles Hub circuit, shown with one USB Dually Backpack and two UART/RS485 Backpacks. 210
Figure 3.14 The hub with all Backpack interfaces exposed (left) and connected (right) to one USB Dually and four UART Backpacks. 210
Figure 3.15 The Knuckles Hub as integrated in the RheoPrinter (5.4.1). This is one of two Hubs used there, which is connected to the "top half" of the machine: the motors which control it's motion in XY, and the extruder system. 210

Figure 3.16 The Knuckles Hub as integrated in the *Little Guy,* a small plotting machine that I developed to test MAXL's ability to manage nonlinear kinematics (2.7.5.1). 210
Figure 3.17 Here I show the knuckles H-bridge driver used in three contexts: to drive a solenoid (left), a servo (middle) and to control the FrankenPrusa's (5.4.2) hotend heating element and part cooling fan (right). 213
Figure 3.18 The FOC Stepper's GPIO and thermistor plugs used to interface with the FrankenPrusa's filament detector and cold-side thermistor, respectively. 213
Figure 3.19 The Knuckles Deadbugger used in two contexts: to drive the Little-Guy's Z-Lifting servo (left), and (right) to control an LED driver and a small line laser module in the RheoPrinter (deadbugger mounted near the top of the end effector, beside the LED driver (green circuit, top left) line laser (silver cylinder) and LED (square with aluminum heatsink) near the tip). 214
Figure 3.20 The FrankenPrusa a Prusa Core One FFF 3D printer that was retrofit with controllers from this thesis, using the Knuckles control hardware. 215
Figure 3.21 The original (left) and retrofit (right) Prusa *Nextruder* Printhead 215
Figure 3.22 Control boards removed from the Prusa (left) and added (right). 217
Figure 3.23 Control boards removed from the Prusa (left) and added (right). 217
Figure 3.24 Wiring removed from the Prusa (above), and *just* the terminal connectors (below), i.e. plugs that go directly into hardware rather than into another control board or connector interposer. 217
Figure 3.25 FOC Stepper Knuckles mounted on the CNC milling machine's X, Y and Z axes. 218
Figure 3.26 The microbolometer on the FrankenPrusa hotend (left) and a prototype accelerometer module. Both were new devices that I integrated in an "ad hoc" manner. 222
Figure 4.1 Backlash is *strange behaviour around the origin.* On the left is one practical example, on the right is a diagram of the driving gear's position (X axis) and the output gear's position (Y axis). In a zero-backlash system, this would be a simple linear plot, but in reality there is normally some dead band here. 230
Figure 4.2 The `little guy` machine, and its nonlinear kinematics where motion of the tool-tip in cartesian space is composed of arcs traversed by the machine's two arms, which are connected to one another with rotary joints. 231
Figure 4.3 At left: A trapezoid in the $\vec{v}(t)$ (velocity *along* the path, in time) is s-curve shaped in position, and has square edges in acceleration, and infinite impulses in jerk (the third derivative of position). In this plot, I am extracting these states from a basis spline that encodes the trajectory (2.5.3); that encoding smooths the infinite jerk pulse, inserting a small ramp in the acceleration trace. At right: trapezoid profiles that have been linked using the junction deviation algorithm. 236
Figure 4.4 At left, a motor torque curve generated by the model in this section, showing torque (normally τ, on the y-axis, in Nm) produced by the motor at varying drive currents (color-mapped), depending on the motor's velocity (x-axis, here in $rads/s$). At right: a disassembled Stepper Motor, showing the stator with windings (copper) and teeth (steel laminations). At left, the rotor on its own showing the tooth offset

on the top and bottom laminations — the biasing permanent magnet is in between those. 249
Figure 4.5 Coulomb friction, rendered with parameters from the x-axis on the milling machine in this thesis. We can see subtle smoothing of the function near the origin, which prevents the function's derivative from jumping to infinity at that point. 254
Figure 4.6 The motor driver I developed in order to complete the work in this thesis. The board can be mounted onto "almost any" stepper motor with the small addition of a diametric magnet being attached to the shaft. 258
Figure 4.7 A diagram of the FOC control algorithm as implemented on the motor driver, showing the main hardware inputs and outputs (yellow), the MAXL interpolator (blue) that drives its closed-loop position and velocity controller, key control components (orange) and internal states that are available as PIPES outputs (dashed lines towards the green block). 258
Figure 4.8 The encoder calibration is critical to the motor's operation because the encoder's readings are nonlinear and FOC requires that we accurately map between rotor angles and electrical angles. Above is a plot of the total mechanical correction applied to an encoder, and below shows how this correction is applied to produce an electrical angle for each mechanical angle. 260
Figure 4.9 The FOC-Stepper mounted to the desk and attached to a disk with a known inertia. 265
Figure 4.10 Simulation (orange) and data (blue) of time-series data from our model and motor during a "spin up" test. 267
Figure 4.11 The torque curve from a motor-model fit (right), and the motor (left). 267
Figure 4.12 Time-series data from a test trajectory through a CNC milling machine's X axis, using measurements (in blue) to produce model-estimated motor currents (orange). 269
Figure 4.13 A frame from the planner's output visualizer, which can run in realtime along with the machine. This is from an "adaptive" CNC machining path. The trajectory is shown in 2D on the left side of the frame, and spatial-series data is on the right-hand side. $v(s)$ is second from the top, with corresponding $\Delta t(s)$ in the third plot. Fourth and fifth are $\vec{I}_{\text{req}}(s)$ (with their limits, dashed lines) and $\dot{\vec{I}}_{\text{req}}(s)$. The sixth plot contains *time*-series velocities for each axis, which are computed for transmission to hardware (4.6.1.5). Finally, the top and bottom plot are related to the planner itself: top shows the gradient of $v(s)$ with respect to the cost function (the direction that the planner thinks it should update velocities in the next iteration), and the bottom plot shows convergence over planner iterations. 272
Figure 4.14 Another frame from the same planner run, now in a section of the path that features a 2D pocket machining path. 272
Figure 4.15 The CNC milling machine from Chapter 6, configured to use the planner for motion planning. Motors are rendered on the right side of the frame. 278
Figure 4.16 One frame of a video that overlays optical video (middle) with thermal video (top left: from an off-the-shelf thermal camera, bottom left: from the FrankenPrusa's integrated microbolometer 5.4.2) and time-series solver predictions (right, in blue)

with measurements of the same values (orange), and a plot of system saturations (bottom right) that are calculated using solver data, see Section 5.11.1. 278
Figure 4.17 A plot of `soft_clamp` from Listing 4.6 280
Figure 4.18 A plot of `soft_err_over_under` from Listing 4.7 281
Figure 4.19 At left: states visited by a trapezoidal planner when it was configured to operate within the shaded rectangle (at a fixed maximum acceleration and velocity). At right: states visited by the planner as it optimizes velocities for the same path, where the force margin is the model-defined limit. 282
Figure 4.20 Position, velocity, acceleration, and jerk time-series data from a trapezoid planner (left) and ours (right). 283
Figure 4.21 Position, velocity, acceleration, and jerk time-series data from a trapezoid solver (left) and our planner (right). 283
Figure 4.22 3D heat map of deviations from target velocity from the same target trajectory, using a trapezoid planner (left) and the planner from this section (right). 283
Figure 4.23 Distribution of deviations from target velocity from the same target trajectory, using a trapezoid planner (left) and the planner from this section (right). 283
Figure 4.24 3D trajectory data overlaid with current estimate errors, the difference between the planner's estimated actuator currents (for A and B motors on the 3D printer from Chapter 5) and measurements. 285
Figure 4.25 Time series comparison the system's motion measurements (dotted lines) with the planner's predictions (solid lines), for a subset of a print job. Lines of opposite color are including improved current estimates, calculated using model updates from a set of refit parameters. 285
Figure 4.26 Speed-dependent weights for refitting: sample speeds (left) are weighted (middle) according to their distributions in a histogram of speeds (right) that defines the distribution. 285
Figure 4.27 Comparing the trapezoid-based velocity planner (left) to the optimization-based planner (right). 290
Figure 4.28 Chip welding on the endmill used in this experiment. 290
Figure 4.29 An earlier version of the planner, shown here, used the direct-shooting MPC formulation. 292
Figure 4.30 Same, but different. These two motors are the same manufacturer model (StepperOnline 23HS22-2804D), but have slightly varying `kt` values, as shown in Figure 4.31 and Table 4.6 295
Figure 4.31 Motor `dut_c` (left) and `rig_y` (right), are the same manufacturer model, but have different model parameters, indicating that their internal construction is slightly different. 295
Figure 4.32 Our print of the Circuit Mount in PETG with Carbon Fill using a 0.4mm High Flow Nozzle. Left: the print, showing "ringing" errors (vertical bands). At right, deviations (yellow) from the planner's position target and the stepper motor's actual positions. 295
Figure 5.1 A diagram of the FFF process. Gears drive filament into a hot zone, where pressure is generated and filament is melted before it is forced out of a nozzle. Some nomenclature: the *melt zone* is the heated part of the nozzle, *inflow* (or *inlet flow*) is

the volumetric velocity into the top of the melt zone and *outflow* is the volumetric velocity at the tip of the nozzle, and we call the melting-and-then-flowing mass of plastic the *melt flow* — this is where the most devious physics happens. 307
Figure 5.2 A nozzle-camera view of FFF 3D printing (left), showing a track of molten filament being extruded on top of the prior layer. At right, a printed part near completion, with the nozzle printing the final layers. 307
Figure 5.3 A 3DBenchy printed with no cooling effects applied (left), and with model-selected cooling (right). Bottom row shows frames from thermal videos of the prints. 309
Figure 5.4 A diagram of the state-of-the-art FFF workflow. Part geometries are imported (normally as `.stl` mesh files, which are exported from design software), and slicers are set up with parameters to match the machine, material, and extruder configuration. The slicer writes a GCode, which is transferred to the machine controller to produce the part. 310
Figure 5.5 A screenshot of some parameters available to configure an FFF print job, from PrusaSlicer. The set which is displayed is incomplete, see proceeding tables in this section. Inset image in the lower right shows the "sliced" part as decomposed into line segments with infill (red), perimeter (yellow), and external perimeter tracks (orange). 312
Figure 5.6 Colours here map each trajectory segment to the machine physics that limited overall velocity at that point: green tracks are flowrate limited, yellow, deep blue and orange tracks are limited by extruder system dynamics and purple and light blue tracks are limited by the motion system. I describe how it is generated using the methods from this chapter in Section 5.11.1. At any time, the physics that limits overall speed vary depending on machine performance and track geometry, for example here we see two tracks that are straight *but* have varying thickness. In these cases the machine must decelerate *while* operating the extruder system such that the width change is controlled precisely; motion and extrusion systems are deeply coupled. 315
Figure 5.7 Like Figure 5.6, this plot maps physics limits to trajectory segments. In this case we see two corners, as the machine approaches each it must slow down to satisfy motion control constraints (motion limits are in light blue and purple) while also dynamically controlling the extrusion system in sync. Both plots also show infill sections where the machine repeatedly traverses small corners, and show how overall limits routinely change between motion and extrusions systems. 316
Figure 5.8 This system is measuring polymer flow from the nozzle tip using a camera (to the right of the nozzle) that measures track widths. The modelling in this chapter uses computational metrology and the flow system's *internal* pressure state to make the same measurements, eliminating the need for these types of out-of-band measurements. 318
Figure 5.9 The workflow that I develop and evaluate in this chapter, a description of which is in the surrounding text. Compare to the state-of-the-art workflow in Figure 5.4. 326

Figure 5.10 The RheoPrinter is an FFF 3D printer that deploys the controllers I have developed in this thesis, and includes instrumentation for flow- and motion model building. Here I roughly diagram the controller's physical layout. 330
Figure 5.11 The RheoPrinter's hotend features a load cell between the hotend and extruder motor (which allows us to measure nozzle pressure), and a filament sensor that measures linear flowrate as well as filament thickness. 330
Figure 5.12 The retrofit Prusa Printhead with labelled control circuits. 331
Figure 5.13 Left: A small microbolometer (the black cylinder at right, pointed at the nozzle) was added to record parts' thermal histories. The nozzle (middle) is surrounded by the print head's part cooling fan (PCF) shroud (left). At right: the Prusa extruder's integrated load cell (the small dog bone shaped cutout in the center of this part). 331
Figure 5.14 Time series data from the steady-state decay test, generated with the routine described in Listing 5.1. In this plot we see six flowrates (colour coded). Nozzle pressure (lower plot) rises to some power as the temperature (top plot) drops over the course of the test. We can also see that faster flowrates decay nozzle temperature faster, which we will use later on to characterize the materials' volumetric heat capacity. 337
Figure 5.15 Estimates for preliminary extruder limits, calculated from data in Figure 5.14. . 337
Figure 5.16 This contour plot (right, nozzle and material that it represents at left) predicts the steady-state flowrate (y axis) for any extrusion force (x) and nozzle temperature (z contour). The slice through maximum extruder load (at right) provides an operating boundary of flowrates (y) for any nozzle temperature (x). 338
Figure 5.17 This plot shows time series data from one of the chirp tests used to fit parameters for the flow dynamics ODEs in Equation (5.6) and (5.7) 340
Figure 5.18 A time zoom of Figure 5.17. Looking closer at the fit and data, we can see that the outflow prediction Q_{out} lags the inflow Q_{in} due to the compression dynamics. . . 340
Figure 5.19 These plots render two flow models, which relate pressure (along the x axis, N) to flowrate (y, mm^3/s). These are complete models that span a range of temperatures (color mapped), and show three operating boundaries: maximum temperature (a hard limit related to hotend material choices), the minimum temperature (taken to be the temperature where flows less than $15mm^3/s$ are impossible) and a *steady state boundary,* which is the same boundary as presented at the right side of Figure 5.16 (it represents maximum continuous flowrate at that temperature). spring rates are also labelled at minimum and maximum temperature. Both models were taken on the FrankenPrusa, using the same 0.6mm diameter nozzle. PETG is on the left, with PLA on the right. 340
Figure 5.20 The steady-state model for flow in PETG with a 0.6mm brass nozzle (left), and the isothermal model (right). 341
Figure 5.21 At left: steady-state (cividis, faded) and isothermal (viridis, opaque) models overlaid on one another in 3D. At right: the result of subtracting the isothermal temperature from the steady-state temperature across the space of (pressure, flowrate) (above), and taking a slice through a particular nozzle temperature (240° C) (below). 343

Figure 5.22 Here I plot extruder current (y axis, both plots) against the load cell reading (x-axis, left plot), rate (colormap, left plot, and x-axis right plot) and acceleration (colormap, right plot). 344
Figure 5.23 This is a time series plot of data used to fit the extruder coupling parameters from Equation (5.8). From the top, we have Nozzle Temperature, Extruder Load, Inflow, Acceleration, and the resulting motor Current, which we use to estimate the Extruder $\dot{I}$. The Extruder Current Estimate (in orange) is calculated using the model described here, after it has been fit using this time series data. 344
Figure 5.24 Left: the same steady-state data presented in Figure 5.14, now overlaid with simulations using a combined loss. Right: combined loss across rates, and our estimate for independent ambient loss and flowrate-dependent loss parameters. . 347
Figure 5.25 Here I plot time series data of the nozzle temperature rise (gray, dashed lines) during heating cycles when we are delivering full power to the nozzle. These data are used to fit k_{in}, which is used with previously fit k_{loss} to generate the simulated temperature rise plotted here in black. 347
Figure 5.26 A time series simulation of layer cooling for ASA+CF across a range of nozzle temperatures, including target temperature and the material's zero flow temperature. 348
Figure 5.27 Isothermal models for PLA and PETG (both using 0.6mm brass nozzles), showing boundaries of operation: short dashes at the maximum nozzle temperature, longer dashes at the steady-state boundary and a solid black line at the minimum temperature (defined with a minimum flowrate heuristic). 351
Figure 5.28 Input heuristics and model-mediated flowrates (tables, bottom), plotted in the isothermal map for PETG with a 0.6mm brass nozzle (left) and for PLA (right, same nozzle). 352
Figure 5.29 This figure shows layer-by-layer (horizontal axis) segment lengths (top row) and times (second from top), colored by segment types: infills, perimeters, small perimeters, and outer perimeters. To meet a minimum layer time (horizontal dashed line, second from top), rates for these segments are skewed: skew proportions are plotted for each category in the bottom four rows. 355
Figure 5.30 This visualizes the layer skews in the figure above in a 3D plot. Most of the skewed segments are within the infill, but we can see that smaller layers (near the top) have larger skews than others: these have been slowed to account for layer cooling time. 355
Figure 5.31 For a circuit enclosure (top) and the RheoPrinter's toolchanger plate (bottom), we see the total print time across temperatures, as well as the minimum layer times and infill flowrates (left plots). These are the same plots as in Figure 5.29 and 5.30 for the 3D Benchy. The Benchy (~60x30mm) has a minimum print time at the minimum nozzle temperature, while the plate (~60x80mm) has a minimum print time around the maximum nozzle temperature. The circuit enclosure (~40x40mm) shows that at crossover sizes, a clear optimum emerges. 355
Figure 5.32 This is a screen capture of the planner's real-time output. The target trajectory is rendered at left and plots on the right show planner outputs. From the top: gradients of $v(s)$ with respect to the cost function, the melt flow temperature

estimate, extruder pressure estimate, outflow, inflow, inflow acceleration, extruder current, $v(s)$, actuator current estimates, and the planner's convergence. 361
Figure 5.33 The rest of the PIPES graph that runs the machine. The large blue block contains objects running in Python, others are hardware. 365
Figure 5.34 Flow Models (left: Steady-State, right: Isothermal) for PLA, using a 0.6mm Brass Nozzle. 368
Figure 5.35 3DBenchy in PLA, using a 0.6mm Brass Nozzle. 368
Figure 5.36 3DBenchy at 1.5x it's normal size in PLA, using a 0.6mm Brass Nozzle. 369
Figure 5.37 Flow Models (left: Steady-State, right: Isothermal) for PETG, using a 0.6mm Brass Nozzle. 370
Figure 5.38 3DBenchy in PETG, using a 0.6mm Brass Nozzle. 370
Figure 5.39 Flow Models (left: Steady-State, right: Isothermal) for PETG (Carbon Fiber Filled), using a 0.4mm High Flow Nozzle. 371
Figure 5.40 3DBenchy in PETG with Carbon Fiber Fill, printed using a 0.4mm High Flow Nozzle. Geometry for this part was mistakenly generated for a 0.6mm nozzle. . . . 371
Figure 5.41 The Kleat in PETG with Carbon Fiber Fill, printed using a 0.4mm High Flow Nozzle. Geometry for this part was mistakenly generated for a 0.6mm nozzle. . . . 372
Figure 5.42 A Circuit Mounting Component in PETG with Carbon Fiber Fill, printed using a 0.4mm High Flow Nozzle. Geometry for this part was mistakenly generated for a 0.6mm nozzle. 372
Figure 5.43 Flow Models (left: Steady-State, right: Isothermal) for ASA, using a 0.4mm High Flow Nozzle. 373
Figure 5.44 3DBenchy in ASA with Glass Fiber Fill, printed using a 0.4mm High Flow Nozzle. Geometry for this part was mistakenly generated for a 0.6mm nozzle. 373
Figure 5.45 A Brake Duct in ASA with Glass Fiber Fill, printed using a 0.4mm High Flow Nozzle. Geometry for this part was mistakenly generated for a 0.6mm nozzle. . . . 374
Figure 5.46 Flow Models (left: Steady-State, right: Isothermal) for Nylon (Glass Filled), using a 0.6mm High Flow Nozzle. 375
Figure 5.47 3DBenchy in Nylon (PA6) with Glass Fiber Fill, printed using a 0.6mm High Flow Nozzle. 375
Figure 5.48 A BMW E36 Hood Latch Handle in Nylon (PA6) with Glass Fiber Fill, printed using a 0.6mm High Flow Nozzle. 376
Figure 5.49 A machine's motor mount in Nylon (PA6) with Glass Fiber Fill, printed using a 0.6mm High Flow Nozzle. 376
Figure 5.50 Flow Models (left: Steady-State, right: Isothermal) for PCBlend (Carbon Fiber Filled), using a 0.6mm High Flow Nozzle. 377
Figure 5.51 3DBenchy in PCBlend with Carbon Fiber Fill, using a 0.6mm High Flow Nozzle. 377
Figure 5.52 An FPV Racing Drone, printed using this workflow in PCBlend with Carbon Fiber Fill on a 0.6mm High Flow Nozzle. 378
Figure 5.53 Drone Fuselage Lower Cap in PCBlend with Carbon Fiber Fill, using a 0.6mm High Flow Nozzle. 379
Figure 5.54 Flow Models (left: Steady-State, right: Isothermal) for Timberfill, using a 0.4mm High Flow Nozzle. 380

Figure 5.55 3DBenchy in Timberfill, printed using a 0.4mm High Flow Nozzle. Geometry was mistakenly generated for a 0.6mm nozzle, see -Section 5.11.6. 380
Figure 5.56 An Easter Island Statue in Timberfill, printed using a 0.4mm High Flow Nozzle. Geometry was mistakenly generated for a 0.6mm nozzle, see -Section 5.11.6. 381
Figure 5.57 A kitchen knife case in Timberfill, printed using a 0.4mm High Flow Nozzle. Geometry was mistakenly generated for a 0.6mm nozzle, see -Section 5.11.6. 381
Figure 5.58 Flow Models (left: Steady-State, right: Isothermal) for AluFill, using a 0.6mm High Flow Nozzle. 382
Figure 5.59 3DBenchy in AluFill (67%), printed using a 0.6mm High Flow Nozzle. 382
Figure 5.60 A motor controller circuit enclosure / heatsink in AluFill (67%), printed using a 0.6mm High Flow Nozzle. Printed at 1.18x original size, to sinter to 1.0x. 383
Figure 5.61 The 3DBenchy in PLA, on a 0.6mm Brass Nozzle, left: SOTA, right: ours. 388
Figure 5.62 The 3DBenchy in PETG, on a 0.6mm Brass Nozzle, left: SOTA, right: ours. ... 388
Figure 5.63 Arms of the drone, left: SOTA, right: ours. 388
Figure 5.64 Fuselage cap of the drone, left: SOTA, right: ours. 388
Figure 5.65 The Kleat in PETG with Carbon Fill, 0.4mm High Flow Nozzle. Left: SOTA, right: ours. 388
Figure 5.66 Our print of the Kleat in PETG with Carbon Fill using a 0.4mm High Flow Nozzle. Left: the print, showing errors in areas of short surface infill. At right, deviations from the planner's estimate of nozzle pressure, $err = F_{meas} - F_{slvr}$, with the color map scaling from -25 to 70 Newtons, blue to yellow. Figure 5.68 shows time-series data from two parts of this trajectory, one (left) where models fit well and the other (right) where the model fit poorly. 390
Figure 5.67 Our print of the Drone Arms in PCBlend with Carbon Fill using a 0.6mm High Flow Nozzle. Left: the print, showing over-extrusion errors. At right, deviations from the planner's estimate of nozzle pressure, $err = F_{meas} - F_{slvr}$, with the color map scaling from -3 to 70 Newtons, blue to yellow. 390
Figure 5.68 Time-series selections from the Kleat print in Figure 5.66, traces in blue are planner predictions and in orange are measurements. The plot at left is a two-second slice around $t = 1298$, an area of the print where models did well to predict measurements. At right is a two-second slice around $t = 1676$, an area of the print (highlighted in yellow in 5.66) where the models fit poorly. Data top-to-bottom: velocities selected by the planner, positions, positioning error, a motor currents, b motor currents, outflow predictions, melt flow dwell time T_{melt}, inflow, load cell readings, extruder current, and saturations from Section 5.11.1. 394
Figure 5.69 This figure renders a section of a printed part in 3D, with trajectory segments binned by whichever machine component (or model consideration) constrained the overall system speed at that segment. Distributions for the total print are below, showing total time that the system spent limited by each constraint. Above is a subsection of the print, where limits are rendered on the trajectory itself. 398
Figure 5.70 The vector sum of A and B motor currents deployed during this section of the toolpath. While we may expect these to peak mostly in corners, here we see that rapidly changing *track widths* also lead to motion system current peaks because the

machine needs to rapidly accelerate and decelerate according to the flow system's limits. 398
Figure 5.71 Left: data output for nozzle pressure during a print. Right: total filament "dwell time" during a print: how long the segment of filament currently being extruded has spent in the melt zone. 398
Figure 5.72 In smaller parts like the drone fuselage cap (at left), cooling tends to dominate total process time — but only in smaller layers. Larger parts like the drone's arms (at right) are dominated by *flow* and motion dynamics. In Section 5.12.1 I discuss how we could extend model-based control to use variable nozzle temperatures throughout a print. 401
Figure 5.73 To express that we would like to maximize speed (at the potential cost of quality), we can increase the *scalars* that describe how much of the underlying maximum actuator motive force (in XY and in the Extruder motor) should be used and push the target extruder force during infill past its steady-state limit. At the top are heuristics / parameters applied to each part, and then we have saturations and bins, and the resulting change to the distribution of accelerations. 401
Figure 5.74 Pressure scalar selections from Figure 5.73; on the right hand we set the infill pressure scalar at $1.75x$ the maximum *steady state* load. The dynamic flowrate can be even higher when we can plan for time-varying melt flow temperatures. 401
Figure 5.75 Updating the layer height (from $0.2 \rightarrow 0.3mm$) doesn't change the flowrate targets in this workflow. Instead, flowrates are selected using the pressure scalars. Changing layer height causes the machine to spend less time overall printing the part (from $12.9 \rightarrow 9.6$ min), and changes the distribution of limiting factors: less time is spent against the motion system's limits but a higher proportion is limited by flow and cooling. 401
Figure 5.76 Isothermal models for PLA, PETG, and ASA (left to right), using the FrankenPrusa and the 0.6mm High Flow Nozzle. 406
Figure 5.77 Isothermal Models for PETG (left) and PETG with Carbon Fill (right), using the 0.6mm High Flow Obxidian Nozzle. 407
Figure 5.78 Isothermal models for PLA (middle) and PETG (bottom), using 0.6mm Brass (left column) and 0.6mm high flow (right column) nozzles. The high flow nozzles have a longer melt zone and some internal geometry designed to improve conduction into the filament. Flow models quantitatively capture the size of the improvement across temperatures and flowrates; in both materials here the steady state boundary (dashed line) is expanded. 408
Figure 5.79 Flowrates and springrates extracted from the same flow model (for Nylon-GF) at two different temperatures: 230° C for the 3DBenchy (left) and 272° C for the hood latch (right). Plots in the second row estimate total print time across temperatures according to the layer cooling optimization. 408
Figure 5.80 A highlighted segment from the model-based velocity planner for FFF, where we see that the planner can leverage some of the flow system's time varying dynamics; a complete explanation is in Section 5.11.5. 410

Figure 5.81 3DBenchy printed in Timberfill, with a 0.4mm High Flow nozzle installed, but systems configured under the conditions where the system is mislabelled as if it had a 0.6mm High Flow Nozzle installed. Left: our workflow, right: SOTA. 412
Figure 5.82 The Kleat printed in PETG-CF, with a 0.4mm High Flow nozzle installed, but systems configured under the conditions where the system is mislabelled as if it had a 0.6mm High Flow Nozzle installed. Left: our workflow, right: SOTA. 412
Figure 5.83 The steady-state model (cividis, faded) and the isothermal model (ciridis, opaque) overlapped in the space of nozzle pressures, flowrates, and temperatures (XY and Z axes). 414
Figure 5.84 Ringing (highlighted here) happens when machines vibrate after making sharp accelerations through corners. 416
Figure 5.85 Bridges are unsupported print layers that span two supported sections. 416
Figure 5.86 I made an attempt (with Quentin Bolsée) to measure die swell using the FrankenPrusa's microbolometer, but its limited resolution made this difficult. . . 422
Figure 6.1 This figure diagrams the basic physics happening when a CNC machine cuts through metal. The cutter (yellow) has a number of edges (called *flutes,* two in this case) and spins at some user-determined RPM. It is moved through the material at a *translational rate,* such that each revolving flute engages with a chip of a set width (called the *chip load*). 426
Figure 6.2 CNC endmills from left to right: fresh, used with small built-up edge formation, and crashed due to undersized chip loading (which leads to melting, rather than cutting). 427
Figure 6.3 These are just a few of the many CNC machining *strategies* available in one modern CAM tool (Autodesk Fusion). Each strategy has some unique parameters to tune, though most re-use a similar subset. 428
Figure 6.4 This is a screenshot of the parameters available to configure one cutting strategy (a 2D Pocket) in Autodesk Fusion. 428
Figure 6.5 Here we combine motor models (from left) and kinematic models (right), to generate **force margin** plots. 433
Figure 6.6 A one-axis force margin plot, which is mapped over axis velocities ($x, m/s$) and accelerations ($y, m/s^2$) with a contour (color map, N) rendering the force available from our motion system under those accelerations and velocities. 433
Figure 6.7 Selecting motion parameters (maximum velocity, maximum acceleration) using motor models, kinematic models, and a force margin. At left, a selection that maximizes acceleration ($2200mm/s^2$) but only allows for a maximum speed of $5mm/s$ and at right, we make the opposite trade-off and select for high speed ($88mm/s$) with only $400mm/s^2$ of acceleration. 434
Figure 6.8 3D output of the chip load deviation analysis tool, in a selected region of the pocket toolpath. This highlights segments of the trajectory where deviations are large. 435
Figure 6.9 An histogram across chip load deviation in the entirety of the pocket toolpath. . 435
Figure 6.10 3D output of cutting force estimates, mapped onto the trajectory. This sample is from one stepdown pass in the pocket toolpath. 437
Figure 6.11 Histogram of cutting forces sampled across the entire pocket toolpath. 437

Figure 6.12 Pictured: the Rat Rig Mill used in this work, outfit with a Saunder's Machine Works fixture plate and vise, stock aluminum, GPenny Spindle (1.5kW ER16, 24000RPM, 65mm Diameter), and fit with NEMA23 stepper motors, and controllers as developed in this thesis. 437
Figure 6.13 All possible motors (top) and their torque curves (bottom). Each was generated using measured parameters for the motor's resistance, inductance, and rotor inertia, as well as model-fit terms for the motors' k_t, k_e and rotor damping terms k_d, using the method described in Section 4.3.1. 440
Figure 6.14 Motors (left column) and machine axes (top row) can be combined virtually to produce force margin plots of each combination's expected performance across a range of velocities and accelerations. 441
Figure 6.15 Selection for maximum velocity and acceleration parameters using force margin. Left: x-axis, right: y-axis. 441
Figure 6.16 Selection for maximum velocity and acceleration, for the z-axis, using force margin. 441
Figure 6.17 The two toolpaths that I tested, a 2D Pocket (above) and 2D Adaptive Clearing path (below). Both were generated in Autodesk Fusion 360, with parameters listed below in Table 6.4. 444
Figure 6.18 First Chips on the CNC Machine, cutting at 1/2 of the final depth, feed rate, and spindle speed. Image at right is a thermal camera frame around the same time. . 446
Figure 6.19 Chips ejected during cutting. Looking at these provides valuable feedback; it is not uncommon for a machinist to measure chip thickness to ascertain what *realized* chip loads are during their toolpaths. 447
Figure 6.20 Machining the adaptive path with final parameters. Image at right is a thermal camera frame around the same time. 447
Figure 6.21 The site of failure in the first pocket toolpath. 448
Figure 6.22 Machining the pocket path with final parameters. Image at right is a thermal camera frame around the same time. 448
Figure 6.23 This plot compares chip load deviation (6.3.1) for the pocket (left) and adaptive (right) toolpaths, including a sample of the 3D trajectory with deviations highlighted and histograms from the entirety of each toolpath (not including rapid / traversing motion). All plots are normalized to the same range $(0-1)$ where 1 indicates a full stop (total deviation) and 0 indicates that the machine is moving at the target feed rate, matching the programmed chip load. 449
Figure 6.24 Distributions of chip load deviations from all moves *labelled* as cutting moves in CAM outputs (above) vs. from all moves *with cut force estimates* $> 25.0N$. 451
Figure 6.25 This plot compare cutting force estimates (6.3.2) for the pocket (left) and adaptive (right) toolpaths, including a sample of the 3D trajectory with forces highlighted and histograms from the entirety of each toolpath (not including rapid / traversing motion). All plots are normalized to the same range $(0-150N)$. 451
Figure 7.1 Interpolation errors (top row, millimeters) along a trajectory in three axes; x (blue), y (orange), and z (green). These correlate well with axes acceleration (second row). The trajectory is the adaptive machining test generated with my velocity planner from Section 4.8.1. 471

Figure 8.1 A digital twin for the Little Guy. 478
Figure 8.2 A diagram of how users of Open Source Software developers interchangeably use components from a commons of functional modules, and develop and publish their own. Software has many "built-in" tools for modularity, but hardware tends to resist generalization. Modular hardware approaches try to bridge this gap, to enable the development of a commons of reusable devices. 481
Figure 9.1 Piranesi's *Imaginary Prisons* (Plate 5) (left), and ABB Robots in cells (right). . 484

List of Tables

Table 2.1 Bytecodes for the routes in Figure 2.4, including the reversed B0:A3′ route. 97
Table 2.2 Bytecodes for the routes in Figure 2.5, including the reversed C2:A0 route. 98
Table 2.3 Opcodes and arguments at the runtime level in OSAP. Codes have variable length, code keys take up the first two bits `[0:1]` of any given code, and arguments are inserted piecewise into the succeeding bits, `[inclusive, exclusive]` and zero-indexed. 99
Table 2.4 An OSAP packet header for a message that is routed between two ports in two runtimes across one link forwarding step. 100
Table 2.5 A serialized Pipe Output as described in Section 2.4.2.4. The succeeding packet header would encode the Pipe's transmit route and receiving function's port index (see Table 2.4 from the OSAP section on networking 2.3.1.3). I am using the argument mapping from Figure 2.9 and types from the load cell's `getComparatorReading` defined in Listing 2.16. 119
Table 2.6 Overhead for packets transmitted through PIPES / OSAP. 175
Table 2.7 Devices, routes, and round trip time statistics for devices in the FrankenPrusa. . 178
Table 2.8 Intervals, data rates, gaps, and total bandwidth for PIPES / MAXL systems with varying underlying performance. The second row is representative of the 3D printing systems in this thesis and the third is representative of the CNC milling machine. 180
Table 3.1 The Backpack pinout. 206
Table 4.1 Table of motor model symbols, units and values, including notes on how the motor controller in this chapter 4.4 interacts with them. 251
Table 4.2 Some new symbols for this step, to define boundaries of motor model parameters, and driving states. 256
Table 4.3 Motor motion states that can be transmitted via PIPES at runtime. 262
Table 4.4 A more extensive list of states (and debugging values) that I use during motor modelling. 262
Table 4.5 Motion model parameters before and after refitting on new data from the FrankenPrusa 285
Table 4.6 Model parameters for two individual motors of the same manufacturer model. . . 295
Table 5.1 Slicer parameters that relate directly to toolpath geometry. 314
Table 5.2 Slicer parameters that relate most directly to printers' motion system. 314
Table 5.3 Slicer parameters that relate most directly to polymer flow physics, cooling, and extruder dynamics. 314
Table 5.4 Slicer parameters that relate to layer cooling. 314
Table 5.5 A table of the heuristic inputs that are delivered to the feedback-based FFF workflow's *preprocessor*. 327
Table 5.6 A table of the contents of the feedback-based FFF workflow's *run file*. 327
Table 5.7 Data returned from the FrankenPrusa's hotend. Each measurement is sent alongside a timestamp. Here I include motor states that are most relevant to printing, the full list of values sent by the motor is in Table 4.3. 333

Table 5.8 Symbols and units for Equation (5.9). 345
Table 5.9 Symbols and units for Equation (5.10). 347
Table 5.10 Values and Parameters for the layer cooling model. 349
Table 5.11 A complete table of flow models generated, including key model parameters. . . . 384
Table 5.12 A table of prints completed using the feedback-based workflow where state-of-the-art comparisons were completed. 386
Table 5.13 A table of prints completed using the feedback-based workflow, where state-of-the-art comparisons were not completed or not possible. 386
Table 5.14 Table of prints that were compared to the state-of-the-art workflow, evaluating time optimality. 388
Table 5.15 Parts printed, their nozzle temperatures, average flowrates, total print times t and errors between the planner's motion system predictions and measured values (in Amps, for the XY stage) and between pressure predictions and measured values (in Newtons). 394
Table 5.16 Compressing saturation bins further into this subset, I tallied the total average saturation across all the prints in Section 5.10.1. We see that *flowrates* are the primary speed limit for that selection of geometries, followed by cooling, extruder dynamics, and *then* motion dynamics. 398
Table 6.1 Motors for comparison, and their parameters. 440
Table 6.2 Kinematic Parameters from our milling machines' axes. 440
Table 6.3 Selected motors, force margins, velocities, and accelerations for our machine's axes. Includes a comparison to the machine manufacturer's selections. 441
Table 6.4 Cutting parameters for the machining experiments. 444

List of Listings

Listing 1.1 An example GCode program that cuts a small square from a 1/8" piece of stock using a CNC Router. 37

Listing 2.1 This line of GCode from Listing 1.1 tells the machine to "turn the spindle on," but how do we actually trace that code path to its implementation? 70

Listing 2.2 Finding the entry point for the GCode that we are trying to modify or extend. . 72

Listing 2.3 This is continued in Listing 2.4. Where interpreters use compile-time configurations to adapt across machines, it can be difficult to ascertain what our machines will do for each different code — even when we can see the source code, as in this case where many potential code paths are available, and are switched using compiler flags like `#if ENABLED()` etc. 72

Listing 2.4 Continued from Listing 2.3. 72

Listing 2.5 This is boilerplate code to instantiate the OSAP runtime in an embedded device in an Arduino build system. An example of link gateway instantiation is in Listing 2.9. Ports are similar, but are most often instantiated via Pipes Function software objects, an example of which is in Listing 2.34. The runtime's main loop routine is described in Listing 2.13. 93

Listing 2.6 A boilerplate scripting utility to establish an OSAP runtime in Python. Here, the loop operates alongside Python's native concurrency framework `asyncio`. Link gateways can be instantiated dynamically or set up at the beginning of the program, which is what I do here: `AutoUSBPorts()` scans for and opens any USB-CDC devices that present with device IDs that are likely to contain OSAP runtimes. It does not matter which `COM` / `tty` number they associate with in the OS because systems configurations will be automatically laid on top of these network addresses later on. This utility also instantiates the `MetaManager,` which builds the PIPES Systems Object Model (Section 2.4.4.1) via network scan. 93

Listing 2.7 The interface between OSAP's runtime and software: each software module connects to a bi-directional Port. 94

Listing 2.8 The software interface between OSAP and a Link Layer, a "Link Gateway." Each connects to an external network segment (the link) using a software driver that extends this interface. 94

Listing 2.9 Links are added to embedded runtimes by instantiating link gateways, each of which wraps a network segment (the underlying link). Each of these examples is a class that is built on the base gateway class in Listing 2.8. In this example I use firmware configurations i.e. `USE_UART_LINK` to define which gateways the runtime will load. 95

Listing 2.10 Packet instruction pointers are read from the packet's first byte, and operation types are in the first two bits of the instruction byte with arguments packed behind. 97

Listing 2.11 The "Virtual Packet" structure used in OSAP: runtimes keep a stack of these in a linked list, which allows them to be insertion-sorted by their service deadlines under "earliest deadline first" scheduling. This struct also stores the time of the packet's

arrival in the runtime, and a pointer to the system component that owns it: packets can be passed internally via pointer ownership without relocating them in memory. 100
Listing 2.12 Embedded link gateways and ports use these functions to check for, claim and relinquish space on the shared packet stack. 100
Listing 2.13 The main OSAP loop. 102
Listing 2.14 Pre-defined message types that can be sent to an OSAP Runtime's Model-View Controller 104
Listing 2.15 The clock synchronization routine that runs in each OSAP runtime. 105
Listing 2.16 PIPES allows device programmers to define which functions in an OSAP runtime will be exposed as network interfaces that can be interacted with at a systems level as Remote Procedure Calls (RPCs) or as dataflow elements. These two functions from the Knuckles Loadcell module (from Section 3.2 and used in the 3D printer Section 5.4) are authored in the device firmware, rather than adding a separate layer of glue code to expose them against an IDL, they are wrapped with the `PIPES_FUNC` macro to generate interfaces. 111
Listing 2.17 The function signature of `setupComparator` from Listing 2.16. 113
Listing 2.18 The function signature of `getComparatorReading` from Listing 2.16. 113
Listing 2.19 Configuration values for a PIPES Pipe. 115
Listing 2.20 Declaring that this Python class (the OneDOF block 2.5.4.2) should be transformed into a PIPES Class using a class decorator. Only the class constructor is shown. 116
Listing 2.21 Pre-defined message types that can arrive at a Pipes Function port, to be handled either by the Pipes Function's Model-View Controller, or processed as inputs to the function itself. 117
Listing 2.22 Type Keys used in PIPES to annotate function signatures and runtime data streams. Quentin Bolsée and I designed this scheme together to be extensible in the future against array- and tensors of the same underlying values. Keys at left, numeric values, and behind comments `//` are names and bitwise representations. 119
Listing 2.23 An enumeration of the objects in the PIPES Systems Object Model. 121
Listing 2.24 The proxy class definition in Python for the Load Cell Knuckle from Section 3.2. 124
Listing 2.25 The function calls that connect a 3D printer's velocity planner (here it is named as the "solver," highlighted in 2.12) to a MAXL Timer (2.5.4.1), and then to MAXL OneDOF blocks (2.5.4.2) 124
Listing 2.26 MAXL's implementation of CoreXY kinematics. 132
Listing 2.27 MAXL's implementation for Scara kinematics, which I developed with Miana Smith. 133
Listing 2.28 A calibration utility script for the Load Cell Knuckle from Section 3.2. 155
Listing 2.29 Once proxies are collected, we can write software defined machine objects that bundle devices to describe systems. 159
Listing 2.30 Configuration includes instantiating systems-specific Pipes Classes / MAXL Blocks (in Listing 2.29) and connecting them: here internal connections are made between

two OneDOF blocks and the CoreXY transform, and then that is connected to hardware devices in the cases where those devices are present. Not all the configuration steps are shown, Figure 2.12 renders more of the complete system that includes e.g. the MAXL Timer block, the printer's model-based velocity planner, data collection blocks, and other hardware. 159
Listing 2.31 Implementation of the Bed Leveler MAXL Block. 161
Listing 2.32 How the bed levelling probe routine is defined in software. 163
Listing 2.33 In this snippet, Quentin opens a connection to a separate Python process (not figured) that calculates his fingers' positions from a video frame. That process sends commands to this snippet, which interfaces directly with MAXL to control the machine hardware. Building machine controllers in software packages where we can more easily "wrap" them in *additional* software-defined interfaces can help to develop ad hoc interactive systems. 181
Listing 2.34 Attaching an accelerometer's main data producing function to PIPES in firmware source code. Rather than declaring these data as a hierarchical structure, a flat tuple of values is often enough. 186
Listing 4.1 Inverse kinematics definition for the FrankenPrusa (5.4.2), with conversion between motor units (*radians*) and machine coordinates (*mm*), and the corexy transform. 253
Listing 4.2 Inverse kinematics definition for the CNC milling machine from (6.4.1), with conversion between motor units and machine coordinates, and a trivial linear kinematic function. 253
Listing 4.3 The coulomb friction model as implemented in Python, using *tanh* to smooth the function near the origin, and an heuristic parameter to define the interval over which *tanh* is applied. 253
Listing 4.4 The motor's mechanical model, and loss function used to fit k_t, k_d. 266
Listing 4.5 The outer loop of the planner, where the cost function's gradients are used to generate updates for the solution. 276
Listing 4.6 A *soft* function to clamp the input value `x` within the `lo, hi` range. 280
Listing 4.7 A *soft* function for calculating exceedances of a value that has both minimum and maximum bounds, like the actuator currents in the planner. 281
Listing 5.1 The test procedure that I use to generate melt flow steady state data. 336

1 Introduction

Machine building is a core competency for any industrial society because the technology that is available on the factory floor drastically changes how productive the economy is [1] *and* the quality (and quantity) of jobs that are available there [2], [3]. Perhaps more important is the manner in which workers interact with that technology — or if they can at all; some half a million manufacturing jobs were unfilled in the USA in 2025 [4], [5] and thirty thousand in Canada [6][1], largely due to a gap in the availability of skilled workers. These valuable jobs [7] often involve the operation and use of digital fabrication[2] machines.

In this thesis I will present new methods and insights that improve our ability to build and operate automated machinery, but a prevalent anxiety with automation is that it will *replace* people with machinery, supplanting the labour share of value (which we all have access to) with capital expenditure (which is unevenly distributed). As David Noble relays in *Forces of Production,* the first CNC machines did so by replacing machinists' tacit knowledge with computer programs that could be developed and managed in a top-down structure [8]. This centralized power and removed intelligence from the factory floor. Not only is this part of why there is a dearth of skilled work in today's factories (operating as an automaton is much less rewarding than almost any other task), it contributes to our factories' lagging productivity: the hype surrounding Industry 4.0 is driven by the notion that we can use technology to bring intelligence *back* to the factory floor, where it is most useful.

Doing so doesn't have to be a zero-sum game: if we apply new tools in a way that we *create new tasks* for humans we can increase productivity overall without the dread, as Acemoglu and Restrepo report in [9]. Aspirationally our new manufacturing jobs should involve machine *development* and refinement as well as use. MIT studies have found that factory workers were hopeful about automation that allows them to do more complex tasks [10] and that flexible manufacturing (which involves rapid re-tooling of equipment) is of key importance if we want to bring manufacturing back to the west where automation has been poorly adopted [11], [12].

This motivates one of the central question in this thesis: *how can we make the* ***development*** *and use of machines simpler?* In particular, I am interested in replacing a pervasive and opaque representation for machine control (GCode, which lies between high- and low-level machine control tasks) with a flexible and interpretable systems architecture that can be modified and operated using commonplace programming design patterns. If we can do this successfully, we can make *development* cycles for new machine tools faster while also connecting them to more advanced algorithms. This in turn helps us to build new workflows for machine users that are based on real

[1] That's about 0.14 unfilled manufacturing jobs *per capita* for the USA, and 0.07 in Canada.

[2] In this thesis, I will take *Digital Fabrication Equipment* to mean CNC Machines, 3D printers or other direct-write fabrication equipment. These have similar internals to the *automation* equipment and robotic systems that are deployed more broadly in factories.

world feedback rather than iterative guess-and-check cycles. Those new algorithms can also provide feedback in a manner that more directly reflects the *physics* that machine operators develop tacit knowledge for.

Machine building and use is also becoming increasingly democratized. For decades, manufacturers have used digital fabrication to build products in large and small volumes. More recently even small engineering firms are using machines to prototype designs in house, build automation equipment and tooling, and to perform process-specific tasks and tests. Alongside these engineers we find scientists, [13], [14], STEM educators [15], [16], [17], craftspeople [18], and autodidacts who all use (and develop) machines to learn: about our world, about mathematics and artistry, and about their ability to change the world while participating in our material economy. It is not surprising that engineers might develop and deploy machinery, but our scientists, educators etc. belong to groups whose *main concern* is not the machines themselves — these people's heads are already filled with other (important!) knowledge. Understanding their tools should not require that they give up an exceptional amount of that valuable headroom.

Machine control is a task that I and many of my peers have found to be surprisingly nuanced; it runs us through a gauntlet of topics from power electronics and current control through time, space, and frequency domains and into model predictive control and nonlinear optimization. I have seen many people wade into these problems after having designed some hardware that they then want to control — assuming this must be a well solved problem — only to find the task riddled with "gotchas," "traps for young players,[3]" and frustratingly opaque tooling.



Figure 1.1: Machine systems that were originally developed to toil in factories are escaping: we find them now in the classroom, the laboratory, and in small prototyping firms. In each of these contexts we need controllers developed for flexibility, adaptability, and inspectability.

[3]A *"trap for young players"* is a term of art for surprising or non-intuitive design patterns, issues, or strategies that are commonly known to initiates in the dark arts, but cause issues for newcomers. As coined by the venerable Dave Jones at the EEVblog youtube channel.

Besides my own friends and colleagues, I have been able to meet many small (and large) companies who are developing new machines e.g., for composite fiber layup, novel additive manufacturing processes, modular circuit assembly, valve calibration, medical device manufacturing, etc. Many of these firms reached out to our research group because they were in the process of rebuilding a controls stack internally, especially in cases where motion controllers were tightly coupled to process controllers.

Behind all of these technical activities are the real motivating questions that led us to developing that new hardware in the first place: why do these tools not feel like *tools*[4], why can't I easily modify an old machine to do a new thing, and why is it so hard to understand why my machine broke, or why the part failed, or why the output is less precise than I had hoped? These are all *systems* level questions that are difficult to answer if and when any of the system components are *black boxed,*[5] which is often the case with our machines.

If nothing else, I hope that this thesis can serve as a guide through these tasks, towards a world where building machines is more like building software; where we can find very good solutions to recurring problems available in *the commons* [21] and easily yoke them to our particular applications.

1.1 A Small GCode Primer

Understanding how GCode works, what niche it fills, why we might want to replace it (and why it is so pervasive) is important background for this thesis. To begin on that front, Listing 1.1 (matched to Figure 1.2) contains a basic GCode "program[6]" that cuts a square out of a piece of stock using a CNC milling machine (a subtractive tool, one-such machine features in Chapter 6).

GCode was developed in the 1950s [22], [23] *around* the same time as programming languages, compilers, and instruction sets: in computer science we call these "layers of abstraction" and they help us to build complex things without worrying about the particulars. Programming languages allow software engineers to write algorithms without manually writing out machine instructions [24], and compilers rely on static *Instruction Set Architectures* (ISAs) to generate low-level codes without concern for how they are actually implemented in hardware (see Figure 1.4).

I compare abstractions made in software layering to those assumed by GCode in Figure 1.4. In computing, the approach works well: softwares describe programs at a high level, where we have lots of semantic meaning and complexity but relatively few "lines of code" for the actual description. Programming languages provide rules for software that ensures those descriptions are regularized

[4]The *tools* framing is one that I borrow from my friend and colleague Ilan Moyer, whose research appears in a number of the background sections of this thesis. I think it's a wonderfully simple question that turns out to be difficult to answer; when we use digital fabrication, why doesn't it feel like craft?

[5][19] is an excellent philosophical discussion of the *results* of black boxing, its origins, and expounds on the value of a *technical mentality* that includes understanding of lower-level processes rather than attempts to *abstract* those away and turn systems into "big green buttons," see also the conclusion to [20].

[6]A GCode "program" is perhaps not actually a "program" since GCodes are (often) not turing-complete, i.e. they are more just a series of operations, rather than a proper computing language. We can fix this.

Listing 1.1: An example GCode program that cuts a small square from a 1/8" piece of stock using a CNC Router.

```
L01   G21                   ; use millimeters
L02   G28                   ; run the homing routine
L03   G92 X110 Y120 Z30     ; set current position to (110, 120, 30)
L04
L05   G0  X10 Y10 Z10 F6000 ; "rapid" in *units per minute*
L06
L07   M3  S5000             ; turn the spindle on, at 5000 RPM
L08
L09   G1  Z-3.5 F600        ; plunge from (10, 10, 10) to (10, 10, -3.5)
L10   G1  X20               ; draw a square, go to the right,
L11   G1  Y20               ; go backwards 10mm
L12   G1  X10               ; go to the left 10mm
L13   G1  Y10               ; go forwards 10mm
L14   G1  Z10               ; go up to Z10, exiting the material
L15
L16   M5                    ; stop the spindle
L17   G0  X110 Y120 Z30     ; return to the position after homing (at 6000)
```

well enough that they can be compiled into *machine codes,* which are written for whichever ISA that it will target. Machine codes are very low level: individual operations are very simple (add two numbers, move a value from one location in memory to another, etc.), but have little semantic meaning and there are very many of them; it is very difficult to make any *sense* of a program by reading the instructions emitted by a compiler. These in turn run well at the physical layer because computing hardware is extremely regularized: it has a clear interface for which instructions are possible and it works identically every time (barring e.g. a stray shot of radiation to the wrong memory cell). Computer science and engineering has gone a long way to ensure that this is the case. These abstractions are in place for reliability and consistency, and they also make code portable: high level programs can be compiled to run across many individual computer chips or even types of computer chips thanks to these regularizing steps.

GCode was meant to take a similar place in machines, allowing machine programmers (formerly *machinists*) to develop manufacturing routines that could be deployed on variable hardware [8]. This approach is stymied by two important caveats.

Firstly, while there are only a few ISAs for compilers to contend with (and even then, a real homogeneity around x86, ARM and now RISC-V[7] in practice) there are perhaps tens of thousands

[7]Instruction Set Architectures encode the lowest level operations that a given computing device can execute: add, compare, move, etc. x86 (originally developed by Intel) runs most PC's and was basically the market dominant until ARM (developed in Cambridge, UK and licensed worldwide) arrived, which offers a simpler set of instructions and is favourable in low-power devices. As it turns out, simpler ISAs are preferred in modern designs because they are more deterministic and easier to predict: instructions complex ISAs like x86 can take more than one clock cycle to complete, for example, which makes compiler-level optimizations more difficult to articulate. The new trend is towards RISC-V (Reduced Instruction Set Computer, 5th Generation), which is open source and even simpler than ARM.

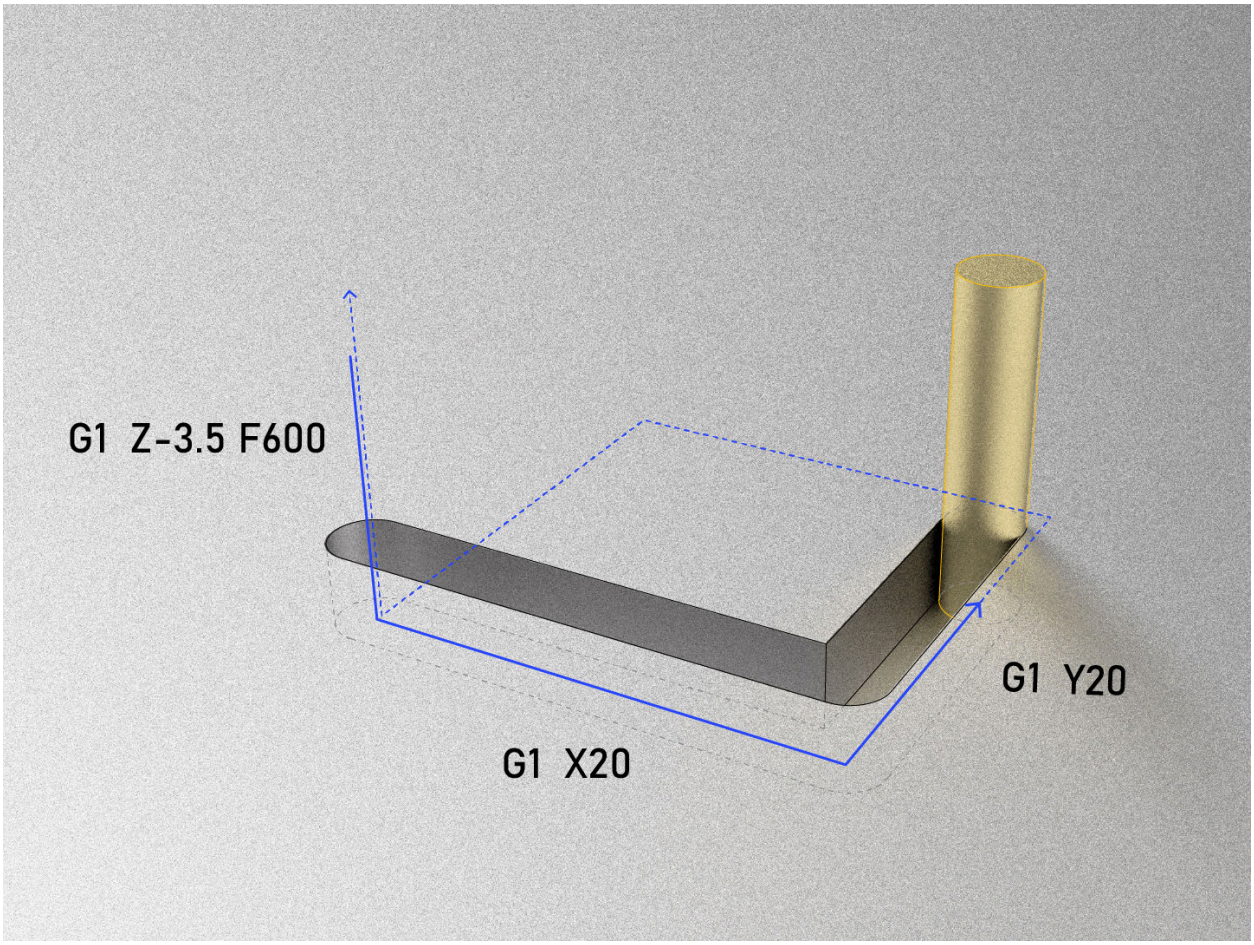


Figure 1.2: The actions carried out by the machine based on the GCode snippet from Listing 1.1.

of unique GCode interpreters[8] and each has a unique "flavor" of GCode because each machine has different capabilities and configurations.

Secondly, physical processes themselves are also heterogeneous: even if we run the same job on a machine thousands of time, each is bound to be different: cutting tools wear out, feed stocks are of slightly different sizes and compositions, external factors like heat soak and ambient air temperature all change. Whereas computer science goes to great lengths to ensure the homogeneity of the lowest layers of its stack (the implementation of ISAs), it is simply not possible in manufacturing to do the same. All the while, GCode provides no recourse to access real-time information from the machine at runtime, i.e. it is entirely *feed forward:* if we want to describe an algorithm that controls a machine intelligently based on physical measurements, GCodes are simply not an option.

GCode also contains a mixture of simple instructions like `G1` (goto position) alongside larger subprograms like `G28`, a homing routine that may involve coordination of many of the machine's components. If we want to interrogate what `G28` is actually going to do (i.e. to see which axis is

[8] A *GCode interpreter* is the controller that reads GCodes and translates them into actual machine outputs, which means two things: first, it contains the hardware configuration (which actual motor is `X`, etc). Second, it is responsible for *dynamics planning* — scaling velocities along the GCode's path such that certain acceleration and velocity limits (also set in the interpreters configuration) are not exceeded. See Section 1.3.3.

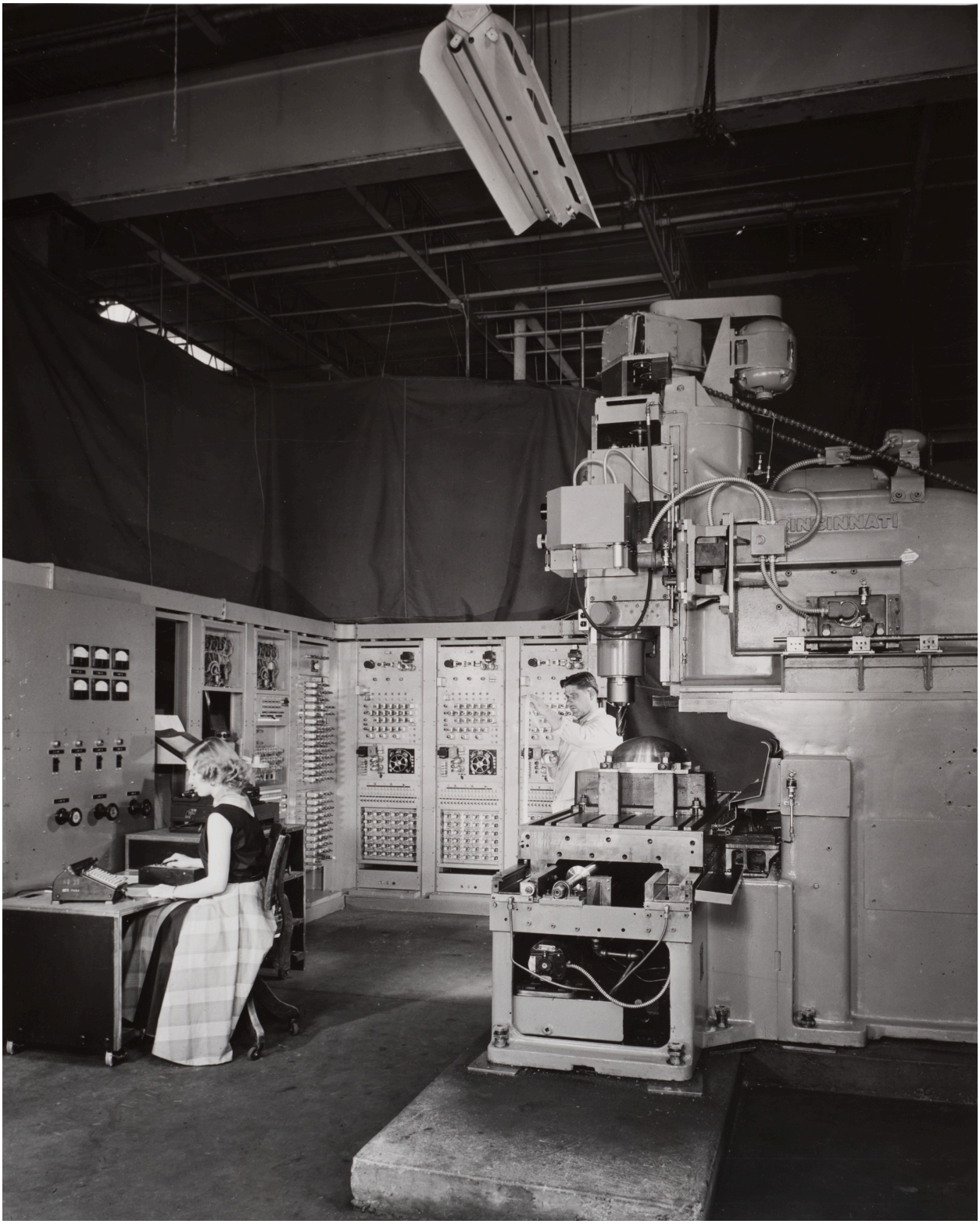


Figure 1.3: The first computer-controlled CNC machine, the 'whirlwind,' was developed at MIT around the same time that programming languages and compilers were being invented.

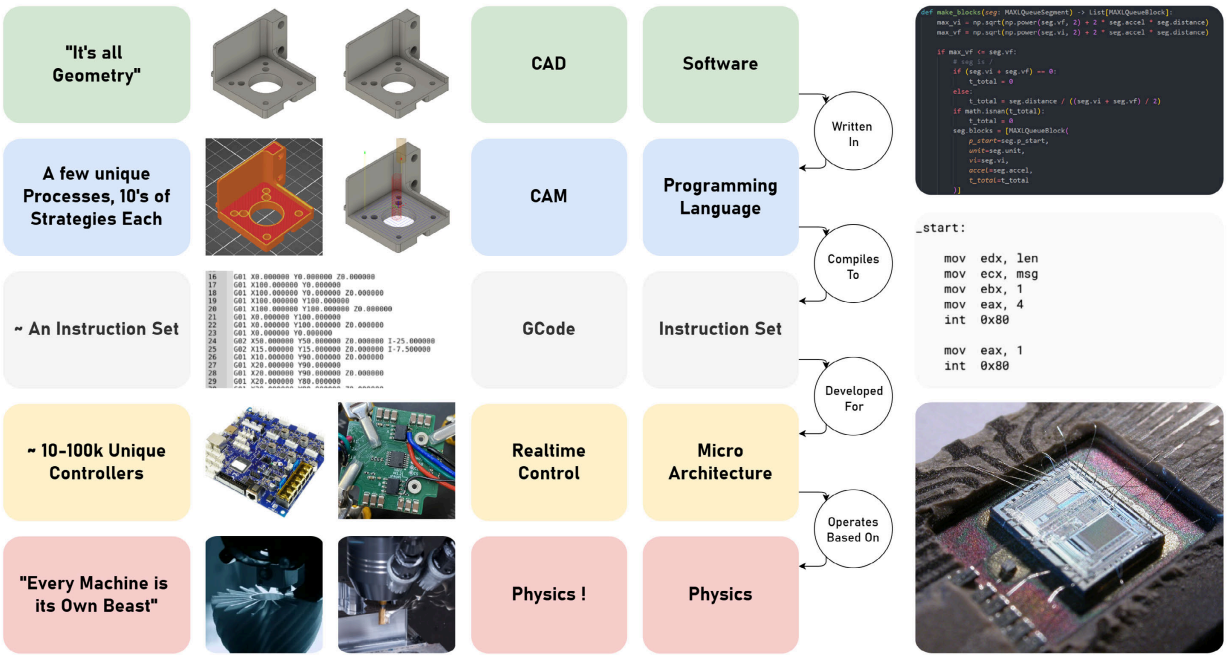


Figure 1.4: Here I compare layers of abstraction in computer systems (at right) to machine workflows (left), spanning high to low level descriptions of operation.

going to move first), there is no way to inspect "one layer down" from the `G28` that is blankly staring at us, unless we have access to the interpreter's source code.

1.2 A (Very) Small CAM Primer

While GCodes can be written by hand, they are most often generated using CAM tools (for Computer Aided Manufacturing). When someone *learns how to make things with digital fabrication,* much of their effort is devoted to learning how to use CAM[9]. We often call this "machine programming." CNC machine programming is notoriously difficult, whereas 3D printing CAM tools (known as *slicers*) are much easier to use (this is one of the main reasons that 3D printing has become so ubiquitous). Each has its own quirks, but follow a similar pattern: a 3D file is imported and positioned in a virtual work volume, and then various parameters are configured such that the software can generate a *path plan* (which will become GCode) for the selected machine. Figure 5.5 and Figure 6.4 give examples of printing and milling parameter sets respectively, and Figure 5.4 for a diagram of state-of-the-art 3D printing workflow. For a longer discussion of each, see Section 5.2.2 and Section 6.1.2.

[9]Another important lesson in this learning process: how to hold your hand over the emergency stop as you hit *go* on the machine, in case your GCode is misinterpreted.

For now the main issue to point out is that CAM tools are *disconnected* from the machines that they write instructions for. GCode does not encode any way to get data from the machine into CAM, nor are the parameters that users set in CAM related to *physical* models of the process.

1.3 Problems in State-of-the-Art Machine Control

1.3.1 Machines are Thoughtless

The simplest way that I have found to articulate the main issue here to say that *digital fabrication equipment does not think about what it is doing,* nor can it tell a user what it is capable of or anticipate errors that may arise from a particular program. When mismatches arise between CAM outputs and real constraints they simply fail. For all of its advance, most digital fabrication seems about as sophisticated as the inkjet printing we are all familiar with: sometimes straightforward and unsurprising, or else error-prone and frustrating, but never enlightening.

For example, a CNC milling machine itself has no 'knowledge' of the physics of metal cutting, chip formation (see Figure 6.1) or structural resonances even though these physics govern how we might whittle a block of aluminum down into the desired shape. In the same way, a 3D printer knows nothing of the rheology involved in heating, squeezing and carefully depositing layers of



Figure 1.5: Just a feed-forward instruction following automaton, when it could be so much more!

plastics in order to incrementally build 3D parts (see Figure 5.1). It is difficult to say whether any computer system has 'knowledge' of anything; in this case I mean that our machines do not use any information about the material physics that they are working with while they operate (most also have scant knowledge of their own physics). Where state-of-the-art machines *do* use sensing and feedback it is normally relegated to one subsystem: for example most industrial milling machines have encoders for positional feedback on each axis, and some high-precision mills also measure temperature along each axis — but these are used in the subsystem that positions that respective axis, feedback doesn't make its way back to higher order controllers, nor to our CAM tools (or to us). In this thesis, I try to close a longer loop across the machines' global control controller, using matched computational models of the whole system (process *and* motion).

This means that in order to operate successfully, an understanding of these physics (constraints!) must be embedded within GCodes, but those are developed by hand in CAM tools where we tune *low level parameters* that are directly correlated to instructions, not physics. For a quick example (see Section 5.11.2 for a much longer discussion), we tune directly the speed that our machines should travel at during operation. Speed is physically constrained by the machine's actuators, mass, and frictions — and changing it also changes process parameters, like how much cut force will be generated. When we're looking at these values in a CAM tool, nothing there can show us how close to those limits we might get if we try to increase the speed. A similar story is true for machine builders (especially non-professionals like those scientists and educators I mentioned earlier), because many state-of-the-art machine *controllers* are configured using low level / feed-forward parameters.

This poses a problem for machine builders (who need to carefully tune their motion systems by hand) and machine users (who need to intuitively tune a slew of CAM parameters). It also means that many machines may not operate near their real world optimal limits because hand tuned parameters tend to have considerable safety margins. Finally, it prevents us from learning from our machines: where the limits are, where we are operating with respect to those limits, and how we might best change our machines or our approaches in order to do better.

1.3.2 Machines are Poorly Represented

The root of those problems is that machine controllers and users don't have good *representations* of machines — to model what they are, how they are configured, what they can do, and what they are doing. A slightly more precise way to say this is that many state-of-the-art machine representations are *lossy* — they are often misaligned with reality, they fail to capture important phenomenology, and are difficult to correct.

For example, in line 10 of Listing 1.1, we tell the machine to do `"G1 X20"` — move the `X` axis `20mm` to the right (from its previous position established in line 5). It is impossible to know from the code alone which motor (or combination of motors) is actually going to execute this move; there is a system configuration that is hidden from us that is unique to the individual machine. This seems like a simple problem (simply write down the configurations, right?) but is a major hurdle for

companies like AutoDesk, whose CAM software (Computer Aided Manufacturing, see Section 1.2) must interface to many machines, and needs to know exactly how e.g. the X axis moves relative to the rest of the machine in order to simulate collisions and show the user what their program will do, and so that their software can write the correct GCodes. This is even more difficult with advanced machines like mill/turn machines[10] where many users result to writing GCodes directly — a clear failure of what is meant to be a low-level layer; in software, writing machine instructions directly is incredibly rare.

In the state-of-the-art, this configuration is set up in the machine's firmware[11], which is difficult to inspect and edit. These "hidden states" cause all manner of headaches for educators, hobbyists, and scientists who want to re-purpose off-the-shelf controllers for their own inventions, especially if their machines deploy nonlinear or novel kinematics[12]. One common result of this problem is that we find machines in the wild whose controllers are convinced that they are a 3D printer when in reality they are e.g. a gel extruder [25] or a liquid handling robot [26], or cases where authors have had to develop their own ad hoc control systems (rather than re-using available designs) [27], [28].

What a machine builder normally wants is a *computational model* of their machine that is *aligned with reality* — this is half semantic alignment (any axis can be named "X" if we declare it to be, but our CAM and controller should agree on the declaration) and half tuning and model fitting

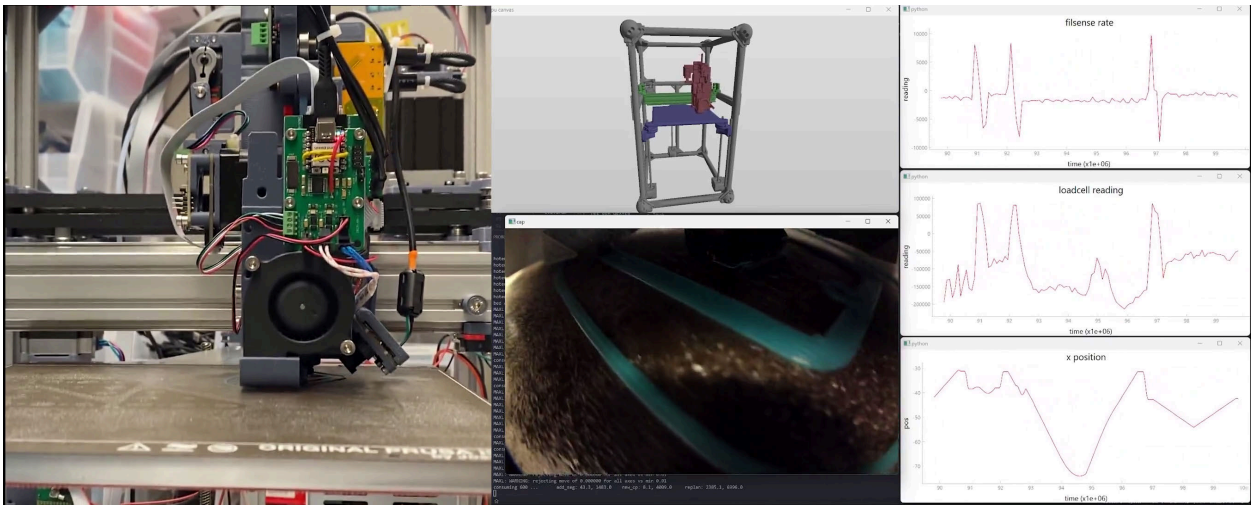


Figure 1.6: In this thesis I develop computational representations of machines that let us simulate them in software. This frame shows a 3D printer (left) and it's digital twin (center, top), alongside time series data (right) of key process variables.

[10]Mill/Turn machines are many-machines-in-one: they can operate as a lathe ("turn") or a CNC mill, and they commonly mix modes. Often they have up to 12 axes, each of which can collide with others depending on configuration, tooling which is installed, and of course the work piece.

[11]We call the low-level computer codes that run on microcontrollers 'Firmware' — the name reflects the difficulty in changing it, and its closeness to hardware. Firmwares typically run on small computing environments (which today means MHz and MBs), vs. Software which normally runs on i.e. a laptop, on top of an operating system. Firmwares are an important component in any hardware system, since they allow programmers to have tight control over their program's timing. Operating systems and high-level language interpreters (on the other hand) sometimes interrupt program execution to switch threads or run garbage collection routines.

[12]Kinematics: how a machine moves, i.e. how axes are arranged and how motors connect to axes.

(how are the kinematics arranged, how much torque is available, and what happens if we apply some amount of it over the course of one second?). With a complete computational models of our machines, we would be able to simulate and visualize them in software to learn what they might do given certain instructions. The danger in this approach is that the simulations could easily be misaligned with reality, which is why I focus on using *in situ* instrumentation to build the models used to represent machines: i.e. we use the machine to build models of itself.

We also want models for the process physics that our machines control. In Figure 1.9 I show how different aspects of machine physics, each of which operates at varying time scales in varying aspects of the controllers, all relate to overall machine operation. As I discussed, GCodes nor CAM tools represent these physics directly. In Section 1.5, I discuss how we might use models to represent this aspect of machine control as well, and how doing so allows us to frame machine operation as a *constrained optimization* task (which it already is, just implicitly) more directly.

1.3.3 GCode Interpreters Obscure Important Control State and Data

Going a little deeper into the GCode *interpreter* I want to look at one of the very *particularly* lossy components of the CAM / controller system: the interpreter where GCodes are translated into real control outputs.

Expert CAM users know that process physics alone are not enough to intuit where a job's parameters should be set. The machine's own physics also present constraints in the form of actuator limits: the velocity of a moving mass cannot change instantaneously — doing so would generate infinitely large forces![13] To counter this, interpreters perform a velocity scaling of the instructions they are given.

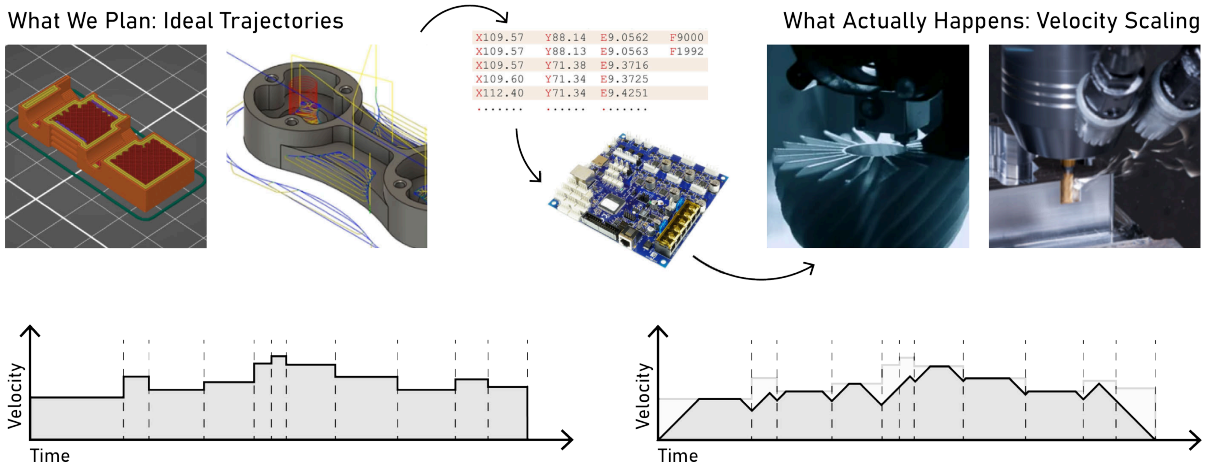


Figure 1.7: This figure shows velocity over time for the GCodes generated by a CAM tool (which requests instantaneous changes in velocity), vs. the actually realized speeds as adjusted by the machine's controller.

[13] That is: $F = MA$

These changes modify process parameters, sometimes in important ways. In CNC machining, when we slow the machine down without changing the spindle speed, we change the size of the chip that the tool cuts, which can drastically change the cutting physics (transitioning from *cutting* to *rubbing,* see Section 6.1.1).

They also make it harder to predict actual machine behaviour, further disconnecting our parameter selections in CAM from what we observe in reality. For example in the figure below, I show the actual feed rates realized by a classical motion solver (using trapezoids) for two target velocities (100 and 500mm/s). We would expect that the second is five times faster, but in reality it is only **14%** faster because the solver's maximum acceleration is limited to 1000mm/s^2 in both cases. This is a simple example of what I mean by a *lossy* layer, or a bad machine representation: the parameters that we set in any given CAM tool may not be realized in practice.

That this velocity scaling happens *beneath* the GCode layer makes it very difficult to understand our machines, and the reason is quite simple. Suppose we add some sensors to our machine to collect time series data in an attempt to make models of some process or of the machine itself. The interpreter's time scaling means that if we want to correlate that time series with the GCodes that we used to drive the machine, we are out of luck: a time interpolation through the GCode (which does not contain real speed information) will quickly diverge from what really happened.

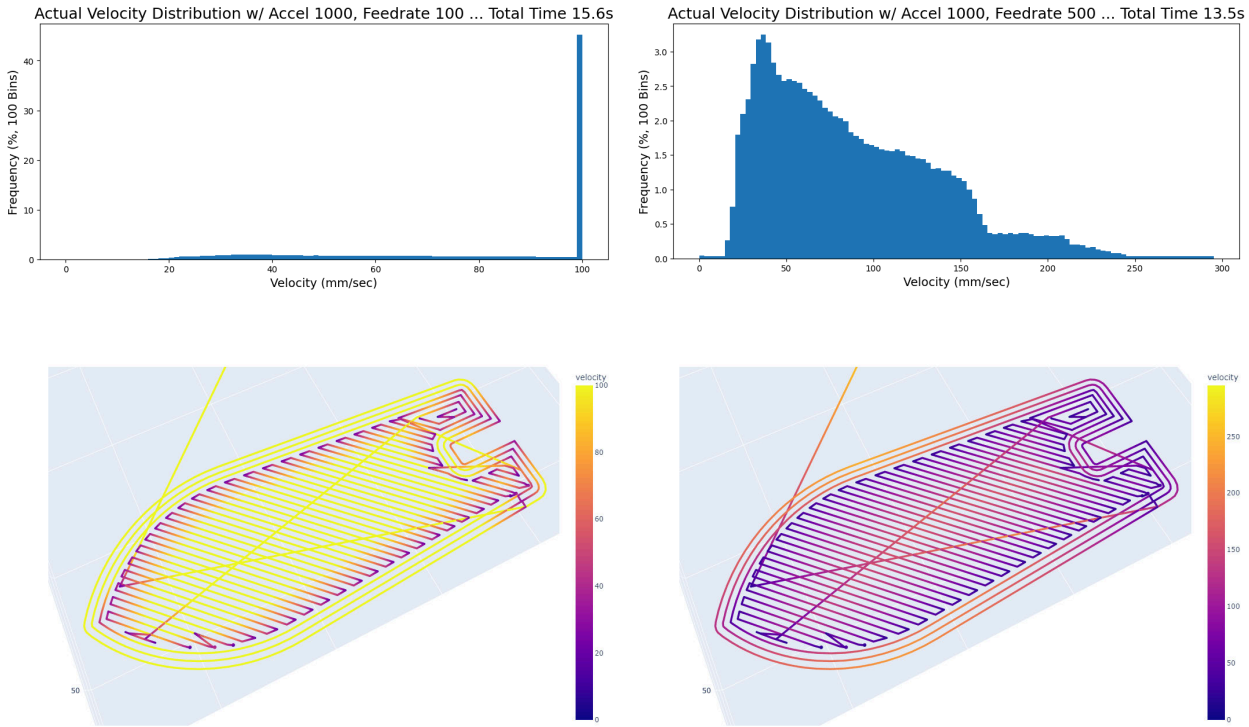


Figure 1.8: Here I show the actual velocities of two toolpaths *after* they have been transformed by a trapezoid-based velocity planner (see Section 4.2.4). At the left, we target 100mm/s and on the right, 500mm/s; in both cases, acceleration is limited to 1000mm/s^2. The histograms (top row) render the actual velocity distributions of each job.

We could add more sensors to track the machine's position in time alongside our process sensors, but we would have to calibrate those with respect to the interpreter's internal offsets and then carefully dead-reckon our time series with the lines of GCode. Properly, we would want to integrate our GCode interpreter with this new sensor system, modifying its firmware to include some new functionality that communicates with our data collection routines. In Section 2.2.1, I show one example of how this might be done — but it is only possible if we have access to the interpreter's source code.

But the hardware that the interpreter is controlling can also be used for model building; actually, motor controllers themselves turn out to be excellent sources of data (see Section 4.4), so direct integration with those would also be helpful. But in some cases those are directly integrated with the dynamics planner and in most others they only have one network interface out: if we want to use them for control, and also measure their internal states, we are out of luck unless we re-write the interpreter.

Really, the content of Chapter 2 is about taking on this task: making machine systems in a manner where they are extensible and comprehendible so that it becomes easier to edit them (across levels) and understand them. In 2.7, I show how that works at a program representation level, and how it solves some of these problems. Building on that system, I continue on to make them smarter in Chapter 4 and Section 5.9.

1.4 Systems Integration and the Partitioning Problem

OK, problems aside, we can start working our way towards solutions and answers. In this thesis I take primary interest in how *systems integration* (SI) relates to machine control and machine operation. Good systems integration often involves breaking a big problem up in to lots of little problems, but where should we delineate those boundaries, and how do we ensure that we don't reduce system performance or capability when we do so? I call this the *partitioning problem,* simply put:

> *Given a system with certain requirements, how should it be decomposed into components (modules) and how should they relate to one another (interfaces)?*

Systems integration [29] is widely recognized as a critical and difficult part of the engineering process. In my time building and using machines I also found that really difficult problems emerge most often during this step: when we start putting things together, new behaviours emerge that weren't apparent from component behaviour alone. Besides my own personal experience[14] (and observations of students' troubles while teaching [30], [31]) there are a few famous SI bugs from

[14]There is much informal *lore* around SI as well, for example the term *Heisenbug* for an issue that disappears as soon as we study it. This is a classic occurrence during systems integration, and especially in networked systems because debugging (which often involves sending error messages over the network) changes that network's behaviour and changes the *timing* of other events in the system.

massive engineering projects to recount; the Mars Climate Orbiter failed due to a divergence between a software interface *specification* and the actual implementation [32], the first flight of the Ariane 5 rocket failed due to an integer overflow[15] error that arose because developers re-integrated old firmware in the new system [33], and the Airbus A380 was delayed for years thanks to a difference in the *software version* that two collaborating teams used, leading to wiring harness length mismatches [34].

In studying machine control, I found a pervasive gap between research on low level machine physics (that govern their operation) and high levels of control (that must manage those physics), which I explore in the background sections for 3D printing (5.2) and CNC machining (6.1.4). In many instances, researchers in these fields are hampered by limits to SI presented by GCode: they cannot readily apply their new methods or models to existing hardware and instead need to "work around" GCode interpreters, in some cases going to great length to model the behaviour of those controllers separately because they cannot access their internals directly.

Conversely, when we *can* tackle big SI problems we stand to gain many new insights on these processes — a selection of which I will relay in this thesis. These are valuable not only to machine researchers but to users, developers, and to society writ large since improvements to these systems have fairly direct relation to our collective ability to produce *real* and valuable new things.

1.4.1 The Economic Partitioning Problem

This is a thesis on machines, not economics, but state-of-the-art systems are often partitioned along economic boundaries (which firms build which parts of a system), not just technological boundaries (which aspects would most optimally go where for performance and legibility). This is especially true in larger, older, and broader reaching industries like manufacturing. At the same time, technological innovation can change economic partitions, so it is worth covering some background here so that we can understand this aspect of GCode's pervasiveness and position, and also to motivate our research efforts.

At economic scales SI relates to how individual firms develop and manage the many components, suppliers, and networks that enable their activities [35], [36]. There is an excellent historical example of *good* economic scale systems integration that occurred during the emergence of the Silicon Valley that we know today that is recounted in *Design Rules Vol. 1;* [37] the outcome of which is a world of computing hardware that *interoperates across firm boundaries,* even amongst competitors. We build PCs and servers with RAM built by *Micron,* SSDs from *Samsung,* processors from *Intel* and *AMD,* and these all work with keyboards, displays, etc. from a litany of other manufacturers. Engineers in these firms do not coordinate interoperation with one another directly, they rely on standards and shared practice. There is a parallel story in the development of the *software* that runs on this hardware, where an open ecosystem of interoperable modules built using

[15] When we represent an integer in a computer using a limited number of bits, the variable can only count so high. If we try to increment it past that limit, it "overflows," either resetting to zero (if it is unsigned) or to its maximum negative size (if it is signed).

portable languages and compilers now powers most of the programs that the world uses day-to-day [38], [39] even though prominent computer scientists though this was impossible as recently as in 2004 [40][16]. Economic researchers call this a form of *collective intelligence* [41] or *collective invention* [42]. Together these represent a world of *tooling for the development of information technology* that has by many measures powered the world's economy over the last three decades, culminating in the emergence of much-lauded *"AIs."*[17]

On the other hand tooling for machine builders is notoriously *proprietary* — even large firms like Fanuc, Haas or Robodrill contract the *controls* for their machine to other firms like Siemens and Heidenhain, whose algorithms are kept close to the chest [43]. This is despite the fact that all the world's digital fabrication tools rely (especially at lower levels) on the same core set of algorithms and control methods for operation: motion control being top amongst them. If state-of-the-art design patterns for these core competencies were more broadly understood and easier to implement and extend, we may see a similar level of growth in the "real" part of the economy as we did recently in the infotech and finance part, i.e. the sectors that provide us with physical, manufactured goods and their byproducts: housing, healthcare, and food.

I could go on at length on this topic, but I will stick to the point: advances in *technical* solutions to the partitioning problem have major ramifications for *economic* partitionings [44], and efficient partitioning leads to better outcomes across layers of the economy: almost nobody looses. This is well studied, in particular in [37] and the field of Transaction Cost Economics [45]. Collaboration on new interfaces for these partitions is also often easier than we think, as Elinor Ostrom showed us [46], [47].

Conway's *How do Committees Invent* [48] shows that *organizational* partitions imprint in their technical products[18] (for better or for worse). GCode's persistence is perhaps most of all a result of this effect: developers of machines rely on it as a stable interface between their hardware and controllers and CAM companies' toolpaths. The schism between these two halves of the machining process (programming and operation) emerged during the development of GCode, as I discussed in Section 1.1. Once these interfaces have emerged, we also have industrial inertia [51] to contend with; existing interfaces lead to the emergence of firms on either side of the interfaces (CAM companies, control companies), who then develop new technical products that, given those firms' market position, do not easily *span* those interfaces[19].

[16]On the partitioning of technical labour, also see Section 2.2.3.5, where some background in distributed systems design (of "microservices") is related to this discussion.

[17]I would like to attest that none-such AIs were involved in the authorship of this thesis document, although I did use them to educate myself on many a troublesome programming task. See Section 7.7 for some discussion on this thesis' relation to AIs; LLMs, RL, etc.

[18]See also [49] for more connections between organizational design and product design and [50] to see how digital tools (like CAD and product data management systems) impact designs.

[19]At the CBA we have the privilege of working with firms on both sides of this story. Although everyone recognizes the value of (for example) a CAM company producing hardware that may help them to develop new toolpath optimizations using machine feedback, they do not have the internal capabilities to successfully design, deploy and sell that hardware. Nor have they developed their software such that it can *receive data from hardware,* because the GCode interface does not permit this.

This thesis will not contend directly with these economic effects, instead I hope to show the *value* of integrated digital fabrication systems to encourage more efforts in this regard. I will try to point to what we can learn and what is possible if we can more flexibly solve the partitioning problem.

For a concrete example of how improved partitioning can push changes up into economic structures, [52] is an excellent study that recounts how the NetScape browser was intentionally re-engineered with modularity in mind in order to launch it as an open source project. That effort was a success: it is now called Firefox.

In 1999, Internet Explorer held 99% of the browser market share, which was an alarming trend for proponents of free speech and democratic access to information [53]. The Firefox rewrite occurred between 2002 and 2004, and its market share peaked at around 30% in 2009. It is credited as one of the main reasons that Microsoft's monopoly was unseated [54] and its success enabled the establishment of the Mozilla Foundation, a key maintainer of and advocate for "openness, innovation and participation on the Internet."

1.4.2 The Technical Partitioning Problem

So, what are the technical constraints that drive various approaches to the partitioning problem? There are a huge number in total; we should focus the scope back down towards the relevant topics in this thesis — machine *control systems:* the software and firmware[20] that make machines move (and in this case, collect data from them), and the electronic hardware, motors and sensors to which those are attached. Machine controllers are just one component of machine *workflows:* the sets of operations that we carry out as we use machines. So, in this framing GCode is the interface that partitions smart / high-level steps in machine control workflows (CAM) from low level steps (control).

In Figure 1.9, I expand on the layering from Figure 1.4, organizing control tasks in terms of the *time scales* that are relevant for each. At the coarsest level we can bin these into two groups. High level tasks like design can happen slowly (years, even!), and *path* planning (generating toolpath geometries, i.e. CAM) can take hours or minutes. Low level tasks need to happen in realtime alongside machine operation, this includes *dynamics* planning (selecting exact control values for those paths) which typically happens at relatively low rates (hundreds to tens of milliseconds), and direct control of hardware needs to happen quite quickly (on the order of hundreds to tens of *micro*seconds).

So, time is one partitioning axis. Tasks along this axis tend to correlate with the time constants[21] of the physics that they control: motor current control needs to happen in microseconds because

[20]Whereas *software* runs in an operating system (which virtualizes hardware), *firmware* runs directly "on hardware" aka "on the metal," (silicon is actually a *metalloid*). No other program mediates between it and the actual CPU where it runs: when we want to make code interact with the world, it is often preferable to write firmware for this reason.

[21]The time constant (normally τ and in seconds) is a system characteristic that approximates how long it takes to respond to inputs. More precisely it is the time after which a system has reached 63.2% of the *steady state* value that it will eventually reach after a new input has been applied.

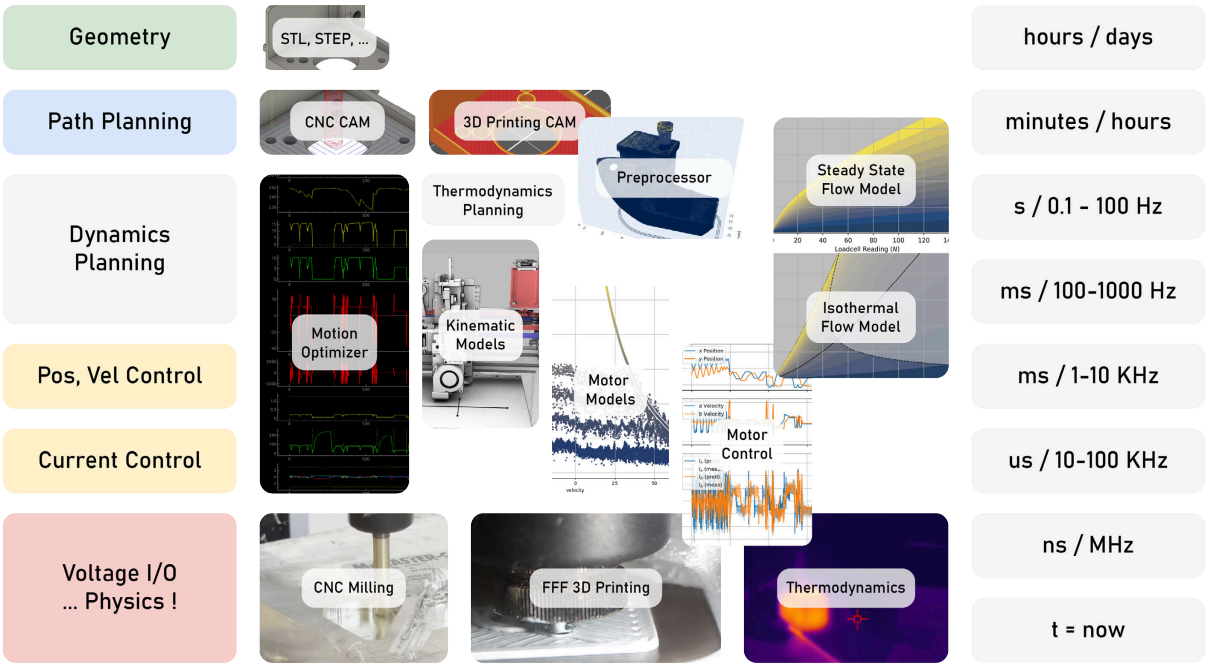


Figure 1.9: Machine control tasks can be organized by the *time scales* that each task operates within: high level tasks occur in days, hours, and minutes, whereas low level realtime tasks must be done in the span of seconds, milliseconds and microseconds.

voltage (which we apply to motor coils) very quickly turns into current, position, and velocity control in milliseconds because that is about as long as it takes for torque to change velocity, etc. The dynamics planning step can be slower than other realtime tasks. For example the top level *human* motor control loop is only about 10Hz [55], humanoid robots' top level loop can be anywhere between 50 and 250Hz, and e.g. the MIT Cheetah's is 30Hz [56].

Better planners can be slower because they are able to anticipate system dynamics farther into the future, but they also require more compute power — another axis in the partitioning problem. Luckily fast tasks are simple and complex tasks are slow. In reality this hierarchical setup has emerged in practice *because* of the partitioning problem: if we could simply run all of our controllers on one CPU and eschew the organizational complexity we would.

Along a compute partitioning axis, we also have the straightforward requirement that low level tasks *need* to run in firmware (where compute power is limited) because they interact directly with hardware. Path and dynamics planners don't have this requirement, but they do need to communicate with layers that do *or* be directly integrated with those layers, e.g. running motor control and dynamics planning steps on the same device. Direct integration of these layers is common in simpler, more accessible machines like those used by educators, hobbyists, and open source scientists. See Figure 3.2 and Section 2.2.4.1 for examples of those and Section 2.2.4.2 for examples of systems partitioned such that the dynamics planner communicates with motors and sensors over networks.

It is implicit so far that each of these control aspects must be connected to one another somehow — but how well we can do this is really what drives us to choose some partitions over others. It is difficult to simply run all of our controllers on one CPU and then (for example) transmit only the very lowest level, simplest control states directly to our hardware because of bandwidth limits between devices. We can try this as in [57] (and it can actually work very well[22] for simple systems), but doing so imposes performance limits that are directly related to those bandwidths.

Dynamics planners in particular sit in an interesting middle ground. They require much more compute than other tasks in the "low level" bin, but they don't require direct interaction with hardware (i.e., they can run on computing systems that are not exactly *embedded*). They are also very central to machine operation — they're our bridge between high and low level tasks. That means that the amount and quality of the bandwidth *into and out of* this layer is critical. Communication between path planners and dynamics planners is important because we want our machines to quickly respond to changes in the plan. In this thesis bandwidth from all lower layer of control into higher layers is key because we are trying to model machines using the data that they generate. Data from the dynamics planner itself is especially valuable because it is where global system states are predicted. Communication from the dynamics planner into lower layers needs to be fast and also *deterministic*[23] because if the planner fails to transmit new instructions to a machine's actuators they will make errors or enter unsafe states. Determinism is the last constraint that I will mention here, and I will also note that it is not a common property in large compute systems because their operating systems can arbitrarily interrupt programs that we deploy there. The same is true of high level languages, which have garbage collectors that can also interrupt operation. For this reason, state-of-the-art dynamics planners tend to run on unique compute hardware that sits on the boundary, normally small Linux-based computers that run specialized "realtime-patched" operating systems.[24]

By now you probably have a sense of why GCode lies where it does; it is exactly where we would expect a sensible partition should be such that we cleanly split big (but slow) computers from small (but fast) embedded[25] hardware. Unfortunately it is a one-way interface with limited descriptive ability; I discussed how this lobotomizes machine controllers in Section 1.3.1, how it

[22] I should also mention here Klipper [58], which does an exceptional job at this by building a highly compressed representation for motion that allows it, too, to relocate dynamics planning onto bigger CPUs (normally a single board computer). Klipper has advanced open source motion control significantly, but still has a limited and relatively inflexible interface between planning and low-level hardware, and is fundamentally still a GCode interpreter. I discuss it at more length in Section 2.2.4.4.

[23] Determinism is a measure of how *sure* we can be that something will happen in a certain manner. In programming, *deterministic* systems always use the same amount of resources (time and space) to complete. This is important in realtime systems: we need to know that (for example) a position control loop will take less time to run than the control interval, every time. High level languages are not deterministic, often simply because they run inside of operating systems, but also because they do things "behind the scenes" like run garbage collectors, etc.

[24] These systems are sometimes called *"crossover"* or *"threshold"* computers. They are often small (i.e. single board) computers with performance that lies between a low end desktop computer and a high end microcontroller, typically towards the high end of that range. Otherwise *regular* computers can also be modified to fit the same domain, i.e. running a realtime-patched linux build on an older laptop is common practice when these are not embedded directly into a machine chassis (or robot).

[25] *Embedded* code or computing is another way to specify that we are talking about *firmware* (see prior footnote). As in, this is compute that is embedded into the world.

makes it more difficult to understand and edit machine systems in 1.3.2, and how it prevents us from understanding how the parameters we set in CAM tools affect real machine physics in 1.3.3.

In this thesis I expand on a long line of systems architecture work from my own research group and elsewhere to develop some new solutions to the partitioning problem.

For more flexible partitioning of modular machine systems using networked control across big and little compute I developed OSAP (2.3). PIPES (2.4) introduces a design pattern for systems representation that combines dataflow with scripting; this lets us configure and task[26] partitioned machine systems in a unified representation. I extend that to develop motion planners explicitly as dataflow programs (2.5), using a synchronization pattern that helps us to more successfully relocate the dynamics planning step from embedded devices into larger computers, where we can take advantage of more flexible and powerful computing tools to extend their performance and make them smarter; I call this exercise *crossing the realtime gap.* I also build a set of modular circuits that show how the partitioning problem extends into the design of hardware (Chapter 3).

By developing new systems architecture for machine control, I want to show concretely that the *technical* reasons that drove historical partitionings are gone or disappearing. There have been many advances in i.e. embedded computing, parallel computing, software design patterns, and automatic differentiation and just-in-time compiling (almost) any program[27] between the 1960s (when GCode was invented) and today. These allow us to build machines that are flexible, intelligent, etc.

That this is possible is not a new insight, but efforts to actually do so have been limited; I get more specific about how the systems in this thesis differs from the state-of-the-art in Section 1.6.2. In 1.6.3 and 1.6.4 I describe contributions in control and modelling that would have been challenging to make without having done this organizational work first. On the other hand, some aspects of the systems that I developed hampered my efforts — I discuss those in Section 2.8 (where I also recount the successes).

1.5 Machine Operation is a Constrained Optimization Problem

Improvements to low level partitioning of our systems helps us to make better sense of them as a whole. Just a few moments ago I said that machine control is really a constrained optimization problem. But what does that really mean? Ignoring what we have discussed so far about how

[26]Configuration and tasking are two basic steps for any mechatronic system: configuration is the setup step: which devices are where, what do they do, what can they do, what parameters do they have, and tasking: telling systems what to do.

[27]Automatic Differentiation, aka `autodiff,` means the process of taking some mathematic function and automatically generating it's derivative, which can be a deviously complex task especially for complex functions. Derivatives (or their higher-dimensioned counterparts, gradients) are key to high performance optimization; most optimizers are based on *gradient descent,* including all of machine learning and AI. These tools have become exceptional in the last few years largely thanks to booming investments in AI. Just-In-Time Compiling (aka JIT) is what it sounds like: taking programs (or chunks thereof) that were written in a scripting language and turning them into compiled counterparts at runtime. These tools greatly improve overall performance of those programs, but they do have some "sharp parts." [59]

all of these machine systems are assembled, at a higher level we have optimal outcomes that are easy to name (making our parts: precisely, quickly), and constraints (which emerge from machine and material physics: we cannot cut with infinite force or move at unlimited speeds). If we can articulate constraints and optima computationally, we can solve the problems numerically, and if we can articulate them more directly to the human beings who design and use machines, we could help them to better understand the problem they are solving and make better decisions as they do so.

In the state-of-the-art, many of these constrains are implicit and require that we carefully author our GCodes such that they are not violated, meaning that the use and design of machines requires a lot of tacit knowledge and intuition that can take years to develop and is difficult to teach to others besides through apprenticeships and direct experience. Most machine controllers are also configured using parameters, not models, and so machine builders also have to guess-and-check for good motion control parameters or must develop their own models and model fitting routines for their machine's dynamics and translate those into controller configurations. That is hard to do using GCode based controllers.

So, the second major theme in this thesis revolves around this question: *can we frame this constrained optimization problem more explicitly?* And beyond that, *can we use the models that we develop to describe machines as an interface between otherwise partitioned layers of control?*

This comes in to play at first very directly (in dynamics planning) but also in a larger outer loop about how we organize workflows. Again, it is fairly well known that optimization based controllers

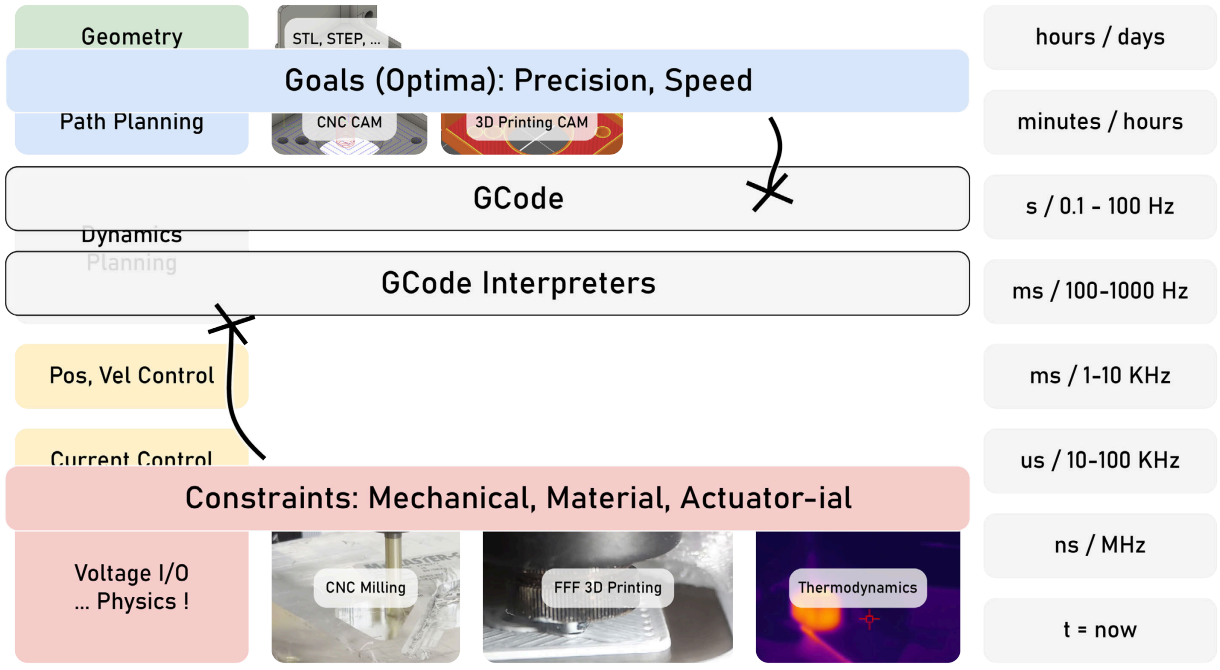


Figure 1.10: Machine operation can be described as a constrained optimization task that spans the same layers as in Figure 1.9. Here, optima are described at a high level but constraints emerge from low-level physics. GCode makes it difficult to connect these two ends of the problem.

can be more performant than their heuristics based counterparts, but there are limited examples of their application to machine tools, especially in 3D printing.

Consider again Figure 1.10: optima are described at the highest layer here, whereas constraints emerge from machine physics, all the way at the bottom. This means we need to learn which model formulations are appropriate for the task, which subsystems should be modelled together and which separately,[28] and how to build those models: can we use the hardware that they describe to build them directly, eschewing larger cycles through offline metrology equipment? Can we use the same models throughout, for heavy planning tasks and for fast control tasks, or will we need to lobotomize some of them? What does a dynamics planner look like when it optimizes over motion and process constraints together — how *much* performance would that unlock, and does it have any fundamentally new capabilities? Which models or constraints turn out to be most important: actuators, stiffnesses, processes?

And how about at the workflow level: can we use models to reduce or eliminate guess-and-check parameter tuning from state-of-the-art CAM? What about in cases where we still need heuristics, can we reframe those heuristics so that they more relate more to machine physics? If we are successful and our models accurately represent machine physics, what new things can we learn about machines *as integrated systems* that we couldn't learn from their components alone?

1.6 Key Questions and Contributions

The broad goal of this thesis is to take these scattered and sometimes misaligned machine systems for control and configuration and try to re-organize them into comprehensive, comprehendible and computational representations (models and programs). Doing so will let us operate our machines more effectively, reduce the parameter space that requires trial-and-error tuning, help us to rapidly build new machines and teach us about those that already exist, and enable us to connect them to smarter control algorithms.

Machine control is an old field that represents many years of collective effort, and I am proposing to replace some of its core assumptions. This is possible now largely due to the ever-increasing ubiquity of computing: the processing power we can now buy for less than one dollar and integrate into a motor driver outstrips (by a few orders of magnitude) what was available at an institutional level when GCode was first developed. This new availability, alongside years of my colleagues' work, leads to the handful of new contributions that I have been able to make.

Many of the *components* or key ideas in this thesis already exist piecewise in the state-of-the-art, but they are seldom integrated into digital fabrication technology. For example there are excellent reconfigurable systems design patterns for large distributed systems like the web, but they have

[28] So there is another partitioning across this axis: we can try to model the whole machine as one complex physical system, or we can try to break it up into smaller models — where should those boundaries go? The partitioning of models correlates also to system partitions, i.e. can we use the same model definitions for high level planning tasks as we use for dynamics control? This is possible if we partition the system such that both of those tasks can be authored in the same language, on the same computer…

not been mapped into the embedded devices that we use to operate machines. There are also excellent model-based control strategies from the world of robotics, but they are less often applied in digital fabrication, and where they are their deployments are limited to smaller subsections of the control problem, for example building predictive models but not controllers that can use them or building predictive controllers for *just* process control or *just* motion control, not connecting those as a unified optimization problem. This too is largely due to GCode's persistence: it prevents researchers from taking their contribution in one of these subdomains and applying it in an end-to-end manner. The overarching goal here is to help us imagine what is possible if we can manage this *systems integration* task, to connect machines, their models, and their users (and designers!) together across all of their emergent behaviors.

In this section, I will summarize the set of questions that this thesis asks[29] and which answers it provides, pointing out to relevant background sections to compare what I learned here to what we already knew.

1.6.1 Guiding Questions

A set of guiding questions runs through this thesis. The first asks what kind of systems architecture would enable us to *develop* feedback-based machine control workflows. What programming representation lets us easily use the same hardware for model building and model-based control? And rather than re-building controllers for each new machine and process, can we do this in a way that is modular and reusable across the heterogeneity of machine processes? How can machine controllers be described when they consist of both high- and low-level components, i.e. how do we *cross the realtime gap* without moving between formats or programming models? Machine control also requires that *software* is matched to *hardware*, alignment that is normally done with manual configuration which can introduce errors. How can we apply design patterns from larger distributed systems to automatically build high-level programming schemas that match hardware, and then detect errors when programs are misaligned with the hardware they should be controlling?

The second is a question of representation: what types of models do we need in order to capture and represent machine physics — for their motors, for their kinematics, and for their processes — and can we build those models using the machines themselves?

The third re-frames machine control itself. If machine control is fundamentally a constrained optimization task, and that optimization is performed intuitively in the state-of-the-art by machine operators, can we instead re-frame it as an explicit, numerically solved constrained optimization problem? Are the solutions that emerge faster or otherwise better than the state-of-the-art, or are the gains only marginal? Among the many tools and formulations available for constrained optimization, which set should we apply?

The fourth is a check on the previous. Models are sure to miss some aspects of machine control that remain easily understood intuitively, and model-based control risks turning well-understood and

[29] I already discussed a few of the more important questions in Section 1.4.2 and 1.5, but I will reiterate them all here so that we can see them all together.

productive processes into black-boxes; this would prohibit machine operators who already have a strong understanding of their hardware and processes from applying that knowledge, and make it more difficult for newcomers to learn. How then do we combine models with heuristics where they are still valuable, and can we present model and optimizer outputs in a manner that is informative for machine users and designers?

The fifth asks what a *complete* feedback-based digital fabrication workflow might look like, and how it performs against the state-of-the-art — in particular, how do we reflect heuristics-based tuning more directly against the underlying physical constraints?

Finally, I ask a similar question to the one above but framed around machine *design* as a practice: how can models and optimizers help designers to evaluate their machines, and how can models be used as tools in the machine design process?

1.6.2 Contributions in Mechatronic Systems Architecture

To answer the *first* question above, I develop a few core methods for the thesis, some of which are novel in surprising ways.

In Section 2.3 I build OSAP (for the Open Systems Assembly Protocol), a networking architecture that connects workstation-scale computing to low-level firmwares using mixtures of heterogeneous links, and builds in each context a consistent *runtime* where task scheduling is aligned with network scheduling. This resolves heterogeneous modular systems as a kind of distributed operating system in which we can build distributed machine programs. The networking scheme is based on source routing, a pattern most often used in spaceborne networks for its extreme reliability and robustness. Source routing is much simpler than the IP-based networks used in most other distributed controls architectures, whose main components require operating-systems-scale computing to operate. This enables us to include embedded devices as first-class members of our systems rather than building custom *bridges* that integrate them piecewise and limit the development of direct embedded-to-embedded dataflows. In fact, each device in an OSAP network is itself an internetworking device, not just an endpoint. OSAP's internal runtime is expressed as a small local network whose operation is defined by a bytecode that extends across the network. Those bytecodes can instruct the runtime to move data between local ports, or between ports in multiple runtimes by inserting extra source-routing instructions. Source-routed networks are simple to operate but require that transmitters understand their local network topology because routes must be written into packets at the source. Normally this is done by hand using pre-defined network configurations, which limits rapid reconfiguration. In OSAP I combine low-level network operation via source routing with high-level set up via discovery and remote configuration; the first step toward this is OSAP's network discovery service, which uses graph traversal over source routes to recover network topologies from the systems programmer's point of view. By simplifying existing synchronization patterns, OSAP also provides a clock synchronization service, which is the basis for coordinated control and sensing for the machines in this thesis.

Sections 2.4 and 2.5 cover my progress on programming models for modular machine control. In Section 2.4 I develop PIPES, a programming interface that represents distributed systems using dataflow and remote procedure call semantics. This allows us to develop machine algorithms whose components are made up of firmware and software modules together, and to both *configure* and *task* systems in the same script — i.e. get them *up and running* without considering multiple hidden layers of configuration. PIPES extends OSAP's discovery layer to add a systems discovery layer that builds a Systems Object Model (SOM) spanning network and program configurations. It can automatically build programmer-facing interfaces and reconcile real-world configurations against program-defined configurations at runtime, preventing errors that emerge if systems are subsequently modified. It does so using remote hardware and software modules' own internal descriptions of their capabilities; we go directly from remote source code, to shareable data schemes, to programming interfaces. PIPES can also configure dataflows between devices, because those flows are specified at a network layer on top of OSAP's source-routing scheme and combined with a dataflow model. Other distributed systems use more complex data-based routing schemes that are overlaid on sophisticated IP-based networks and run on operating-systems-scale computing. OSAP and PIPES instead use simple networks and simple rules for data transmission and program configuration, neither of which are mediated by stateful internal systems: we remove these internal states by coordinating configuration at a high layer that uses the SOM as an interface to write lower-level instructions against. While this is simpler than other distributed systems, it is more complex than statically-configured controllers, so to span these two domains (which are commonly combined using the *bridges* I mentioned earlier) without changing representations, we need to develop a series of small Model-View Controllers (a software architecture from the web) and combine their outputs to generate the SOM.

Within the PIPES framework I develop MAXL (Section 2.5), a motion control framework for distributed but coordinated execution of machine controls. MAXL is uniquely suited to be deployed within PIPES, and is architected to be useful in cases where we need to cross the gap between high-level computing and embedded computing — framing distributed trajectory generation and interpolation on a time-encoded basis, and introducing a configurable *gap* of time between these two ends of the problem. It uses a fixed-interval time encoding that makes network loading between these layers more deterministic when compared to spatially-defined trajectory representations, which (for example) require more bandwidth in areas of high geometric detail than they do along linear segments.

In Section 2.6 I make a complete comparison of the systems developed here against the state-of-the-art, and in Section 2.7 I summarize how those architectural deltas uniquely enabled the other work in this thesis. I show how they transform GCodes to just "codes" (programs), how they enable reconfiguration of hardware for new software, and reconfiguration of software for new hardware. I also show how they help to build machine models by revealing internal control states, aligning those to sensor readings, and allowing us to deploy powerful but indeterministic computing for model-based control. In Section 6.3.1 I show how even simple motion controllers, once exposed as software objects that can be used virtually, can provide valuable feedback and insight to machine users for CNC machining — especially when coupled with machine models, where we can then

generate process measurements that would be unavailable otherwise (Section 6.3.2). Finally, all of these experiments run on a modular kit of control boards that I develop in Chapter 3.

1.6.3 Contributions in Machine Control

With these new systems integration tools, I can begin to answer some of the questions about model-based control of machines.

In Chapter 4 I develop a velocity planner that expresses machine control as an explicit constrained optimization problem, and show that model-based control can exceed the performance of heuristics-based solvers that are common in the state-of-the-art for the same task. This requires that I develop a few additional methods: a closed-loop stepper motor controller that can learn some of its own model parameters (Section 4.3.1), models of machine mass, damping, and kinematics (Section 4.3.2), and the planner itself in Section 4.6.

A core goal of model-based control is to use models directly as constraints in constrained optimization planners. Even in state-of-the-art machine systems and in other research, motion controllers still use proxies for these constraints; they describe maximal jerk, acceleration, and velocity values to limit machine motion. The planner that I develop leverages high-power computing hardware (the GPU) and modern optimization frameworks (developed for Machine Learning) to articulate the problem directly against physical models. OSAP and PIPES make connecting this compute hardware to our machine hardware more straightforward, and I also carefully formulate the planner such that its program can be parallelized for the GPU. The framework that I use is written in Python, which is also a preferred language for model building. This means that we can use the exact same computational representations in machine modelling and in machine control, and update them without recompiling planner firmwares, which connects machine modelling to machine control more literally than was possible before.

To examine explicit optimization across not just motion but process control, I develop in Chapter 5 a 3D printer that models polymer melt flows (Section 5.5), and extend the planner to include these physics in Section 5.9. This demonstrates a few things. The velocity planner from Chapter 4 is flexible enough to describe not just motion control but also process control. These tasks are of course coupled in the real world, but state-of-the-art systems control each aspect separately, with process control *implicitly* configured by parameters in CAM tools and motion control via velocity planners that run in lower-level interpreters. Process models for 3D printing of thermopolymers can be developed using the same hardware that controls them, eschewing an outer loop through process metrology on standalone and expensive hardware. To do this I develop new models for the process that extend the state-of-the-art in two ways. First, they are modelled across a range of nozzle temperatures to capture a broader set of possible process parameters. Second, I combine their steady-state (in the thermodynamic sense) behaviour and their isothermal behaviour, which allows us to model their time-varying flow dynamics. I extend the velocity planner to include these time-varying dynamics, which is another unique capability for machine controllers — which to-date are all based on time-invariant dynamics.

1.6.4 Contributions in Machine Modelling and Workflow Development

In Section 5.8 I show how models and heuristics can be combined to complete a feedback-based digital fabrication workflow (Section 5.3), and evaluate this, along with the flow models, by printing a number of different materials with a number of different nozzles in Section 5.10.1. I compare these prints to a state-of-the-art workflow, showing that the new workflow captures existing practice and (on the metric of speed) out-performs it. By articulating *goals* rather than *outcomes*, optimization-based workflows let us productively explore wider swathes of parameter space. Intermediate steps can then select optimal parameters that we would not normally tune our systems to — for example using hotter nozzle temperatures in large parts and colder temperatures for small parts, trading off layer cooling time against flowrate increases. Combining heuristics *with* models also allows us to extend those heuristics across a range of underlying physical constraints, reducing the total number of parameters that must be tuned to generate successful outcomes (Section 5.10.3). This means that the values we *do* tune can be tuned against the underlying physics they represent: we can pick parameters *against* constraints, rather than trying to estimate where those constraints are and tuning parameters blindly (Section 5.11.2). I also (accidentally) show that model-based printing workflows can correct for human errors, i.e. mislabelling a machine component (Section 5.11.6).

In Section 6.2 I develop model-based tools for machine designers for the purposes of motor selection and motion control parameter selection. In Section 6.4.2 I demonstrate the use of these tools (in combination with tacit knowledge) as I commission a CNC milling machine. I also show how the model-based 3D printer can provide useful feedback to machine users and designers in Section 5.11.1 and Section 5.11.3.

In all of these exercises, machines build their own models. The models that I use for motors and for motion are not novel, but the new control framework allows us to use the same models — literally the same code definitions — to represent *and* control our hardware.

2 Systems Integration for Modular Machine Control

Up and Running

2.1 Introduction

So, GCode splits our computational representation of machines across the "realtime boundary" (1.4.2). This means that overall machine operation is defined at many layers: by configurations and their associated algorithms in CAM tools, GCode interpreters, and lower level controllers like motor drivers. Those are normally set up by hand[30] and the underlying algorithms are often hidden and stateful, making it difficult to ascertain what will happen when we input new machine

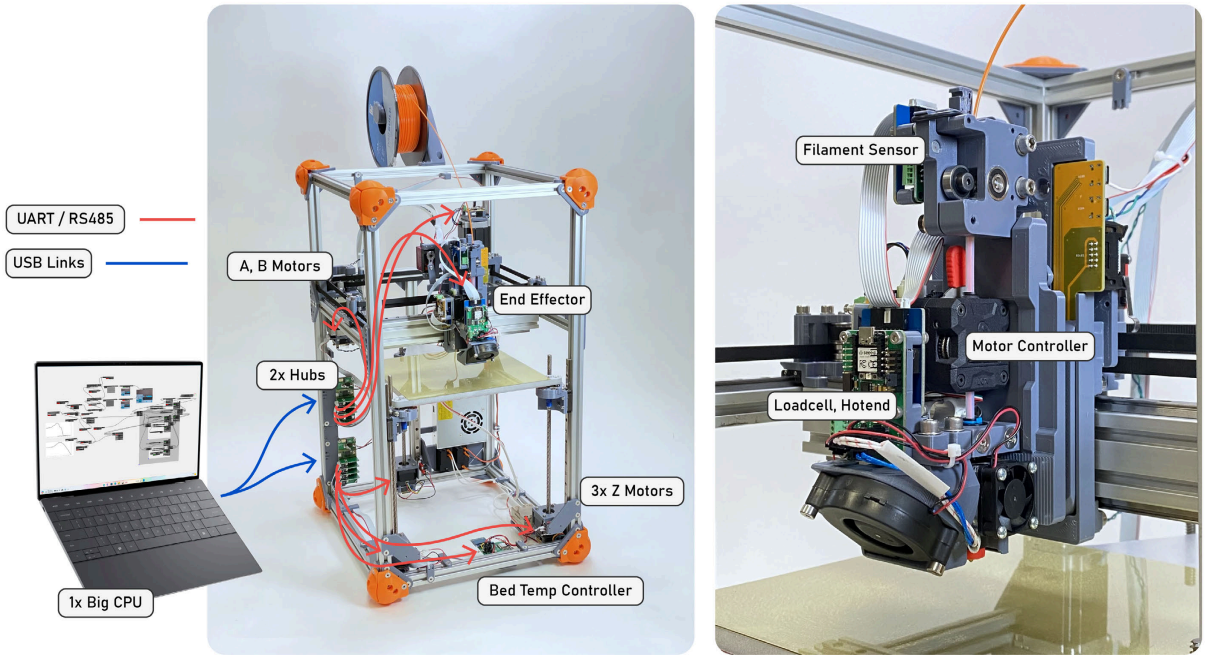


Figure 2.1: Here I show the FFF printer used in Chapter 5. The system is composed of multiple circuits, each with a particular task. The systems assembly challenge is to make the assembly of devices into a coherent controller; PIPES is the programming model used for this, and other machines in this thesis. MAXL is the motion control framework.

[30]When multiple configurations at multiple layers in the system are set up manually and cannot be easily checked, errors can easily emerge if any of those configurations disagree with one another due to human errors or misunderstandings.

instructions or make changes to isolated components; this is the black boxing that I discussed at the very start of the thesis in Section 1.3.2 and 1.3.3.

On the other hand, if we can connect across this boundary we stand to improve machine control overall; connecting real physical constraints to planning algorithms, sourcing configurations from hardware rather than aligning them manually, and integrating machine controllers directly into new workflows to enable more interactive operation of hardware and tighter couplings between motion and process controllers.

Taking a cue from software design, we also want to build machine systems from a reusable set of components across hardware (motor drivers, sensors etc.) and software (motion planners and modelling routines). Doing so lets us re-use modules across projects to quickly compose new systems from the parts bin rather than re-engineering circuits, drivers, and code.

So, the focus of this chapter is to develop a set of systems integration tools that "span" the realtime gap, connecting big and small modules in a distributed runtime and developing a programming model that can configure and coordinate global operation of devices within that runtime. This will focus first on networking technologies, and then on the development of a programming model to suit.

2.1.1 Machine Building Needs Flexible Network Architectures

In practice machine-scale modular hardware systems are deployed on a heterogeneity of different network links and transport layers [60] whereas the internet is dominated by only a few (TCP, Ethernet, WiFi, etc). This is true for the same reason that many different types of machines exist: the physical world is itself heterogeneous. Sometimes we want networks to be *cheap,* sometimes we want them to be *extremely performant,* sometimes noise immunity is critical, and other times it is not; this leads to a proliferation of different networking technologies. Because embedded devices are low on computing power, building clean interfaces between network layers is not top-of-mind for automation engineers[31] and because machines tend *not* to be required to integrate within larger networks there is little industrial pressure to develop interoperability *across* network architectures.[32]

The Open Systems Interconnect model (OSI [61]) developed by the international organization for standards (ISO) is meant to separate network layers from application-level concerns so that we could plug almost any device into almost any system. This follows the end-to-end principle [62], which proposes that systems be designed such that modifications to their *ends* (applications) not require updates or modifications to systems architectures themselves. The heterogeneity of machine control networks means that we really *need* the OSI model to work for automation perhaps more than we do for the internet. However, the OSI model is only "loosely" followed by most

[31] Layer interfaces and abstractions add computing and memory overhead.

[32] For example, a difference in network architecture from one machine to another is OK when those two machines don't need to be interconnected as part of a larger system as is the case for the internet, where connection to the network itself is the main activity. In fact, there is some industrial pressure *to maintain* architectural differences in networking technologies because these can form the basis of intellectual property "moats."

modern networks [63] and industrial networks in particular tend to collapse layers. CAN bus [64] (common in automobiles and simpler robotic systems) is perhaps the best example: we can only run CAN-type application layers on CAN busses, e.g. CANopen [65] and J1939 [66]. EtherCAT [67], [68] collapses the OSI layers for routing and transport: connecting devices on an EtherCAT network to nonlocal applications requires special *tunnelling* protocols. Time synchronization of one EtherCAT network to global time via the internet's precision time protocol (PTP, [69]) requires a special bridge.

This divergence in systems is not due to a simple lack of better standards; network performance constrains realtime control of hardware in important ways as I will discuss in more detail in 2.2.2.1; the through line is that networks for control should be performant and also deterministic. In fact EtherCAT's collapse of intermediate layers is done explicitly to remove overhead in those layers and thus increase performance, and CAN based networks are constrained in the application layer because CAN packet structures and application semantics are themselves combined in order to reduce packet overhead.

It is easy to imagine simply improving network systems' performance to the point where we could alleviate those constraints on architectures themselves (making room for network abstractions and layers), but the requirement for determinism bleeds more broadly into systems design. For example to build reliable networked controllers, accepting arbitrary network traffic from outside is often unwise: we would not want to expose those systems to DDoS[33] attacks or (for a more reasonable scenario) to be modified to the point where bandwidth consumed by new modifications saps the performance of previously reliable networks.

So, try as we may it is not really possible to separate the design of our networked applications from the design of networks themselves, or the design of the computing systems that run those applications. This is the *scheduling* problem, where we must take some set of tasks (an application) and run those in a constrained amount of time (available on network links in terms of bandwidth limits, and on computing devices in terms of clock cycles).[34] The main difference between purely informatic networks (like most of the internet) and networks that control machines and robots is that the former has much more relaxed time constraints, whereas in the latter failures to pass messages in time can make for unsafe behaviour. There are advanced algorithmic approaches to scheduling that provide good solutions[35] to the challenge, but none can completely guarantee optimal performance because scheduling is NP-Hard. Articulating network behaviour and programs themselves completely is also difficult, so in practice systems are often designed by hand against worst-case analysis [70] and then tested for safety — modifications to these systems are then difficult to make without re-assessing their schedules.

[33]In a DDoS (Distributed Denial-of-Service) attack, networks are flooded with malicious packets and as routers try to manage this surge in traffic, they collapse.

[34]The word *scheduling* in computer science broadly refers to this task: the problem first really became relevant when the first operating systems were developed (to share one CPU across many programs). It is NP-Hard.

[35]For example, good solutions to the scheduling problem can help to improve CPU performance substantially, so there is significant work on the topic.

The scheduling problem itself is well outside the scope of this thesis, but it is an important consideration as we design networks. Comprehension of the networked systems over which distributed programs are run is a critical step in solving the scheduling program either by hand or algorithmically; we are also motivated to build systems that have built-in mechanisms for feedback and measurement.

One final note on our motivations with regards to networking systems: embedded networks can fall under the purview of professionals whose main practice is not in information technology itself. The assembly of the internet (and its integration within businesses, homes etc.) is mostly carried out by network professionals, whereas machine design firms, factorys, or open source machine developers are less likely to be as completely versed in the nuances of network design. For this reason again we are motivated to make network systems that require minimal manual configuration and that provide some degree of flexibility and inspectability.

2.1.2 Machine Building Needs Flexible Software Architectures

Connecting devices over networks is not enough, we also need a programming model with which to use them.

While partitioning machine systems across the realtime boundary makes excellent sense from a reliability and determinism point of view, machine controllers are in practice a smaller part of much larger and more complex workflows; managing multiple layers of configuration and statefulness can end up limiting the development of those workflows. For example in the introduction I discussed how this partitioning prevents us from connecting high-level tasks like CAM from low-level constraints that can be seen by controllers; this makes it difficult to articulate machine control overall as a constrained optimization task or otherwise apply modern, intelligent controls strategies. In one of the background sections here (2.2.4.7) I show some examples of other research where machines are integrated into new workflows via GCode *wrappers;* software objects that expose machine functionality with an API but use GCode "under the hood" to control hardware. This approach is itself invaluable in the first pass, but then tasks like extracting real machine state, modifying motion parameters, configuring kinematics, or synchronizing control with motion trajectories becomes an issue. I make note of similar limits for state-of-the-art modelling of motion control systems themselves in Section 4.2.8 and I survey other improvements to controller partitioning in 2.2.4 and discuss difficulties in extending and reconfiguring GCode interpreters themselves in Section 2.2.1.

However, we cannot simply collapse the realtime boundary and e.g. use one monolithic program to operate the whole controller, so the architectural challenge is this: having partitioned our control systems, what kind of programming model can represent algorithms that *span* those partitions.

A machine programming model needs to work for two basic tasks: configuration and operation, aka to get our hardware *up and running.* The first means making sense of which devices are in the system and connecting them to one another according to some control logic. This includes configuration of software modules *throughout* the system, but we will want to be able to do this

without re-compiling and flashing firmwares which is time-consuming, cumbersome (especially on large and complex machines), and can easily lead to misalignments between firmware and software configurations. The second is to operate those machines, i.e. *task* them; deliver trajectories to machines, home and jog them, or describe higher level algorithms that control them etc. — broadly, tell them what to do. While the *operation* of our machines is partitioned, we would prefer to *configure and task* them from one place. With the systems I develop here we can develop application-specific scripts that can ascertain and modify the system's global state and then perform whichever tasks or control steps are required by the application.

Since we have multiple devices in these systems, representing networks and parallelism in the programming model is key; we will also need to develop a common *runtime* for software within each device such that the execution of local parts of a program is self-similar to operation of parts of the program that run globally.

2.1.3 Chapter Overview

This chapter presents OSAP in 2.3, it is an Open Systems Assembly Protocol and networking runtime that connects hardware and software modules together in one consistent distributed runtime over heterogeneous network links. To build and configure distributed programs in this runtime I build PIPES in 2.4 (for Programming In Piped EcoSystems). It is a programming model that combines dataflow with scripting and includes *tools for looking at and configuring partitioned systems,* making sense of their global structure. Finally, MAXL (2.5) is set of PIPES modules that

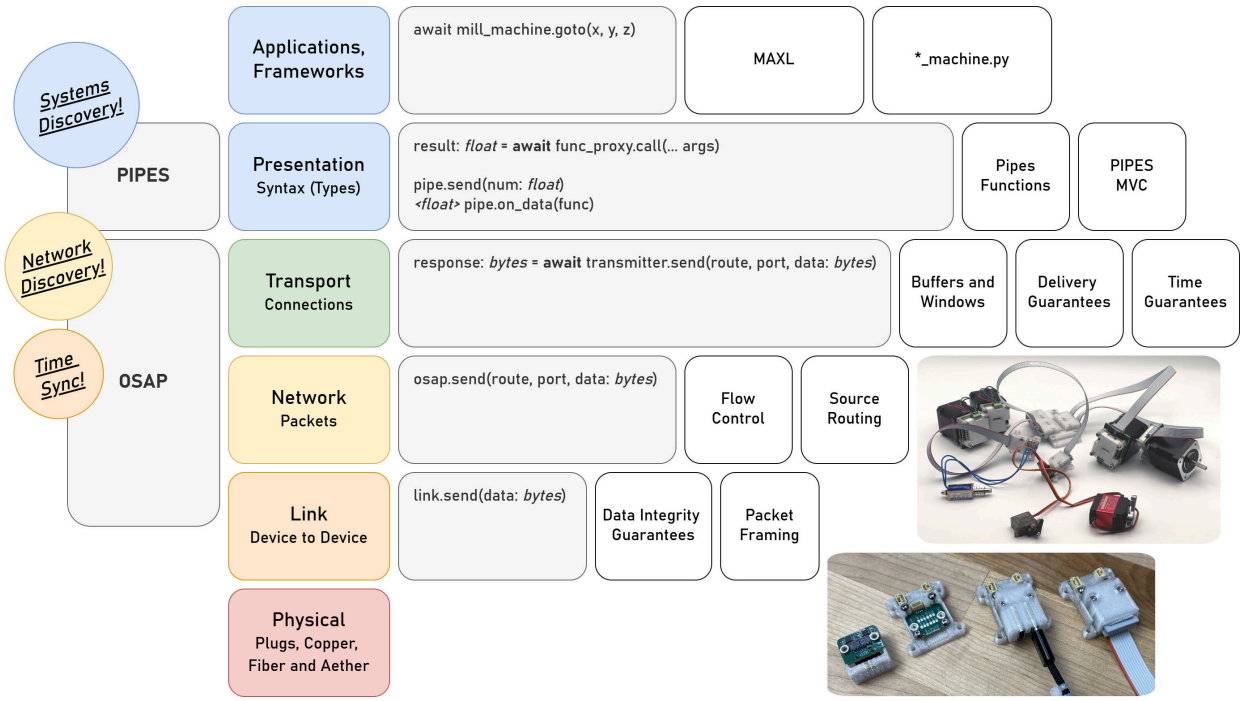


Figure 2.2: A simplified OSI layering, including OSAP (network layers), PIPES (presentation layer) and MAXL (an application built on top of PIPES). Each layer presents well-defined software APIs to surrounding layers in an effort to make components of the system easy to interchange.

express motion control configurations *using dataflow* so that controllers can be *flexibly partitioned.* These map to the OSI model for systems interconnect in Figure 2.2.

2.1.3.1 Key Questions and Goals

At its core this chapter is about learning new ways to solve the partitioning problem that I described in Section 1.4, which unpacks into a few more focused questions.

Partitioned systems span two basic representations: networks and the programs that run across them. In the background section on state-of-the-art distributed systems (2.2.3) I show that most of them represent distributed programs using shared data layers that abstract network operation even though understanding network operation is critical to understanding how these systems run. Their integration of truly low-level embedded devices is also limited and indirect even though the operation of those devices is essential for tight control of mechatronic systems. So, how should we formulate an architecture that adds embedded devices to distributed systems as first-class citizens and also extends up into high level computing? Another common pitfall of existing distributed systems is that they require manual alignments across program and network configurations, can we use design patterns from other complex distributed systems like the internet itself and datacenters to resolve this issue, even in much smaller computing environments?

An overarching goal of the thesis is to develop model-based control for machines in order to simplify and improve machine operation and to expose the hidden optimization that I described in Section 1.3.3 so that operators can better understand their hardware and so that planning algorithms and modelling tools can relate more directly to low-level machine states. Doing so involves running substantial parts of the controller in off-the-shelf operating systems where high performance computing is available and systems development is straightforward, but also requires tight integration with low-level embedded controllers. This introduces another series of questions and goals: in this new architecture how should we synchronize motion control outputs written in non-deterministic operating systems with execution across modular hardware, how do we collect data in those systems, and how can we minimize indeterminacy and latency across the networks that span them? Using the same programming model and runtime in operating systems *and* in embedded devices is likely to limit the capability of high-level compute while introducing overhead in low-level devices, so what is the right balance between complexity and descriptiveness and is there a way to enable simple, low-level operation within devices with complex high-level global configurations?

And what about the programming model itself? Machine operation requires high-level configuration *and* tasking, but also fast low-level operation, what kind of systems description tool can combine those without introducing undue overhead? Implicit in all of these questions is the goal of building *flexible* systems that span variable machine kinematics and application, so in each case we have the additional challenge and goal of accomplishing these tasks using modules and learning e.g. how big those modules should be, which subroutines they should contain, how they can be combined across applications and how they should be authored.

In a really compressed framing: the GCode partition lies right in the gap between high-level / operating-systems scale computing and low-level embedded computing. To build better machines we want to connect these layers, so this chapter considers how we should span the gap computationally and contributes new architectural models to do so.

2.1.3.2 Background Overview

The background section (2.2) starts with an overview of the longer arc of machine systems research from my own local environment at the Center for Bits and Atoms (2.2) and then takes a practical look at the problem, showing how off-the-shelf GCode interpreters are partitioned and modified in Section 2.2.1. With that broader context in mind I take a step back and look at relevant research and themes in network organization and network-based control in 2.2.2, and then the same for distributed systems and programming models in 2.2.3, and finally a closer look at machine control specific partitions and architectures in 2.2.4. Each of these aspects of the broader problem are interlinked. Overall the background is about understanding how the partitioning problem from Section 1.4.2 is solved or managed in the state-of-the-art, not just for machine systems but also in broader contexts where excellent design patterns have been developed for similar problems e.g. in the internet itself, and datacenter architectures and massively parallel computing and in other mechatronic systems like spacecraft and robotics.

2.1.3.3 Methods, Contributions and Results

I present this chapter's architectural contributions in three sections, OSAP (Section 2.3) establishes a common network-oriented runtime within and across distributed devices, routes messages over heterogeneous link layers using a stateless networking scheme, provides synchronization and configuration discovery services and basic performance measurements, and does so within a simplified OSI model. PIPES (Section 2.4) develops a distributed programming model that combines dataflow with scripting to configure and task machine systems using a unified Systems Object Model that combines software and networking. It includes tools for discovery and modification of systems-wide configurations and schemas, and for device API discovery and authorship. MAXL (Section 2.5) develops modular motion control as a series of reusable dataflow blocks for kinematic transforms, offsets, and other reusable control components.

In Section 2.6 I show how these contributions differ and improve on state-of-the-art practice; how they represent of data (2.6.1.1), operation (2.6.1.2), networks (2.6.1.3), and configurations (2.6.1.4) across distributed systems, how they enable tighter integration between model-based controllers and machine networks in a comparison to other state-of-the-art model-based control researchers' work (2.6.2), and how they add important capabilities to other machine control specific architectures (2.6.3) while simplifying their configurations.

In the start of the results section I summarize key architectural deltas over the state-of-the-art and explain how those enabled other work in this chapter and thesis (2.7.1) which also serves to introduce the results from the chapter. I show how they turn GCodes into code (2.7.3) and that they allow "soft" reconfiguration of hardware for different realtime tasks (2.7.4) and reconfiguration of hardware across different machines (2.7.5) including a litany of kinematics (2.7.5.1). To improve control and feedback, I will show how they expose the hidden velocity optimization simply be

moving it from hardware into software (2.7.6) and how they overlay sensor data with controller data for online modelling (2.7.7).

Overall the systems that I develop represent a new *combination* of design patterns from state-of-the-art practice in a litany of other fields. This includes lightweight networking strategies from early IoT and spaceborne networks, discovery and configuration patterns from datacenters and the web, and composable systems insights from visual dataflow systems. In this thesis they are newly applied to machine control. In doing so we can invent new design patterns for distributed systems that combine high-level and low-level computing while maintaining that low-level devices are first-class citizens, new methods for the operation of machine controllers across the realtime gap in that let us blend non-deterministic but computationally powerful computing with deterministic low-level computing, and new strategies for the rapid reconfiguration of hardware modules and software modules — especially for software-defined motion control using dataflow. We also learn what the key limitations to these architectural deltas are and what types of systems we should focus on developing in the future.

Broadly I would say that the SI tools I develop in this chapter (more so than in other contributions in the thesis) are combinations of other systems and methods. I borrow patterns from datacenter / distributed software systems, from web development, and from other network-based controllers. The main differentiator across these contributions and my own is that these patterns have seldom been applied to machine control: here we have more direct constraints on the time-based performance of these systems and smaller, more heterogeneous computing.

2.2 Background in Systems Integration

Systems integration spans many domains: programming and compute models, API design, networking, etc. A complete discussion of each topic is outside of the scope of this thesis, but I want to cover the relevant constraints and considerations and then look more closely at other researchers' efforts to improve machine controllers from this perspective: building new ways to configure and program digital fabrication machines and workflows.

So, I will organize this background section from low- to high levels: on networks (which constrain our partitions), on distributed programming models (to describe, use and modify partitioned systems), on various partitionings of *motion* control systems and machine programming tools, and finally on research efforts to make machine *workflows* more organizable.

Before all of that background, I would like to note that the SI tools that I have developed here continue a research program that has been ongoing at the Center for Bits and Atoms (CBA) for some time. The core idea is that we can build *Object Oriented Hardware (OOH):* if we develop machine controllers as assemblies of *virtual* software modules that mirror real hardware modules we can more easily modify, extend and understand them.[36] This was first developed by Ilan Moyer[37] [71] and Nadya Peek[38] [20]. In my efforts I extended this architecture across a more flexible networking subsystem (in OSAP, Section 2.3) and developed a dataflow-based programming model and motion controller (in PIPES 2.4 and MAXL 2.5) that modularizes lower levels of our machine controllers. I also improved tooling for module and systems authors that reduces configuration misalignments between "real" and "virtual" representations of modules (in 2.4.2, 2.4.4.1) and extended the paradigm's application in model-based machine control (Chapter 4, 5, 6).

With regards to networking itself, the CBA also has a history of developing small networks for inter-device internetworking [72], and OSAP's message-passing architecture is modelled roughly on this idea and more directly on a pattern proposed by my advisor Neil Gershenfeld[39] for "Asynchronous Packet Automata (APA, [73])" to provide the key services of *naming, routing and flow control* using a stateless and source-routed[40] scheme.

The notion that we should apply dataflow in machine control applications was introduced to me via the `mods` project [74], which organizes machine CAM workflows into a reusable set of computational blocks for common path planning tasks.[41] The impetus to apply the approach for control itself emerged because `mods` lacks a clean way to then apply those modular workflows onto hardware;

[36] In the introduction I wrote a section on how machines are *poorly represented (1.3.2),* Object Oriented Hardware is the clearest way to begin solving this problem.

[37] Who appears in this thesis' acknowledgements, and who I have been lucky enough to spend time and share research notes with during especially the latter half of my time at MIT.

[38] Who is one of my co-advisors on this thesis, and whose guidance as I have developed these tools has been invaluable.

[39] How we might build simpler and more *stateless* networks and machine controllers has been a long-standing question posed by my advisor Neil Gershenfeld.

[40] See more on *source routing* in Section 2.2.2.2.

[41] I discuss `mods` in more detail alongside other tools like it in Section 2.2.4.7.

the value in applying the same computational model to both layers of machine workflows was discussed in Section 2.1.2 and in many other sections in the thesis.

2.2.1 Steps Required to Understand and Extend a GCode Interpreter

To compare against how systems are configured and then tasked (or modified) in my own systems, I want to take a quick look at all the steps required to reconfigure a GCode interpreter. In Section 2.7.3, I provide a similar work-up of the same process as developed using the tools in this chapter.

For context, much reconfiguration of machines can be accomplished *above* GCode, e.g. by simply wrapping interpreters with programming interfaces (that expose a Python[42] API, but write GCodes to hardware under the hood). The background Section 2.2.4.7 shows a few examples of this practice. It is an excellent way to quickly turn a machine into a programmable tool, but they have limited recourse to machines' realtime states, as I discussed also in 1.3.3.

I will also note that not all interpreters are open source, and in those cases modifying them in this manner is out of the question. This is why most machine workflow researchers choose open source interpreters, in particular the Duet ecosystem [75] which runs RepRapFirmware [76] is often chosen because the authors make simple reconfiguration a clear goal in their development of the ecosystem, which is also partially modular — I discuss those in Section 2.2.4.2. Difficult-to-modify interpreters have motivated a litany of research and the development of other new architectures, some of which I have already just mentioned, the rest are in the subsequent subsections of this review, mostly starting in Section 2.2.4.

Even with open source interpreters, understanding how GCode interpreters work can be challenging because code paths trace through compile-time configuration options, which you will see in the next few listings, e.g. the block between `#if HAS_CUTTER ... #endif` in Listing 2.2 will only compile if a `#DEFINE HAS_CUTTER` flag has been set elsewhere in the source code. This is a common strategy used by firmware developers,[43] and actually I still rely on this method in some cases within the architectures that I develop — although I do aim to eliminate these steps, too, they are at a low enough level to enable the work in this thesis.

The larger problem overall with GCode interpreters is that GCodes themselves are documented externally in tables that are maintained by hand. So for example the block of GCodes below uses `Gxx` and `Mxx` codes to define program operations:

Machine programmers then use resources like RepRap's GCode Table [77] to track what individual codes (like the `M3` at Line 07 above) should do (and whether they are supported on the particular firmware they are using). Machine vendors with closed controllers provide similar tables, normally

[42]Python is a popular scripting-based programming language that has been widely adopted by scientists, engineers, backend developers etc.

[43]It is common for GCode interpreters to be configured for different tasks using a firmware sourcecode file called i.e. `config.h` where system settings are changed. To update settings, we modify this file and recompile the firmware. More modern firmwares allow these configurations to be read and written remotely, normally through an auxiliary interface but sometimes through custom GCodes.

Listing 2.1: This line of GCode from Listing 1.1 tells the machine to "turn the spindle on," but how do we actually trace that code path to its implementation?

```
L07   M3  S5000            ; turn the spindle on, at 5000 RPM
```

in machine operating manuals. There are some standards for GCode, but they are not strictly adhered to because machine hardware is highly heterogeneous. In order to interface with CAM companies, they must collaboratively develop "post processors" that are applied by CAM tools to ensure that the toolpaths they output align with the "flavor" of GCode that the machine accepts. Those "posts" also define machine kinematic configurations, so if a machine owner wants to modify their machine's motion systems, they must re-engineer the post. And there is a different post processor format for each CAM tool.

Each of those codes is implemented in firmware, below I show the snippet of C++ that defines the switch for any "M" code — here I am looking at the Marlin firmware (a derivative of GRBL [78]) that is popular among hobbyist FFF printers. This snippet shows the compile flag `HAS_CUTTER` to define motion so even if we can find this snippet of code, we still won't know if the firmware that is currently running on our controller has it set — we would need to look at the configuration file as

M3: Spindle On, Clockwise (CNC specific)

Support	**Marlin**	**RepRapFirmware**	**Klipper**	**Prusa**	**Buddy**	**Repetier**	**Smoothie**	**Druid**	**MK4duo**
	Yes	Yes	No	No	No	Yes	Yes	No	No
	grbl	**Sprinter**	**BFB**	**FiveD**	**Machinekit**	**Redeem**	**Teacup**	**Yaskawa**	**MakerBot**
	Yes	No	No	No	No	No	Yes	Yes	No

Parameters

`Snnn` Spindle RPM

Example

```
M3 S4000
```

The spindle is turned on with a speed of 4000 RPM.

Teacup firmware turn extruder on (same as `M101`).

RepRapFirmware interprets this code only if in CNC mode (`M453`), in laser mode (`M452`) or if a Roland mill has been configured. You must always provide an S parameter with this command to specify the required spindle speed pr laser power. In RepRapFirmware 2.05RC2 and later, and RepRapFirmware 3.0beta13 and later, in laser mode (M452) the laser will only fire during G1/G2/G2 moves.

In Repetier-Firmware in laser mode you need `S0` .. `S255` to set laser intensity. Normally you use `S255` to turn it on full power for moves. Laser will only fire during `G1` / `G2` / `G3` moves and in laser mode (`M452`).

Figure 2.3: The specification that defines what the `M3` GCode will do and what its expected parameters are, from the RepRap GCode Table — a community resource that is maintained by hand by volunteers.

Listing 2.2: Finding the entry point for the GCode that we are trying to modify or extend.

```
// in Marlin/src/GCode/GCode.cpp
// line 477
  case 'M': switch (parser.codenum) {
      // ...
      #if HAS_CUTTER
          case 3: M3_M4(false); break; // M3: Turn ON Laser | Spindle (clockwise), set Power
| Speed
         case 4: M3_M4(true ); break; // M4: Turn ON Laser | Spindle (counter-clockwise), set
Power | Speed
        case 5: M5(); break; // M5: Turn OFF Laser | Spindle
      #endif
      // ...
  }
// exit line 1172
```

well. This is not an issue when the value is properly represented in the reference table (Figure 2.3), but if we or someone else has modified the controller we would need to see the source code that was compiled and loaded onto the controller.

So, the `M3` from earlier went to `case 3` to call `M3_M4(false).` The actual implementation of that function can be found deeper in the firmware source, with another selection of compiler flags (`ENABLED(LASER_FEATURE)`). The complete definition of that function is in Listing 2.3.

Firmware for GCode interpreters can be particularly difficult to understand because they contain functions like `planner.synchronize()` and `planner.buffer_sync_block()` which both relate to the machine's acceleration controller; it is using a different time encoding for motion than what a pure interpolation across GCodes would generate. As I mentioned in Section 1.3.3, GCodes themselves are not encoded in time, but machines of course are. This means that interpreters are always managing multiple representations for actions they should take: one which is skewed according to the velocity optimization, and another which is not (yet). Understanding *when* something will actually be scheduled vs. when it is *queued* into the interpreter is difficult, because their internal representations in *time* are varying depending on what stage of the planning step they are in. For a more complete overview of velocity planning, see Section 4.2.4. In Section 2.2.4.8, I show some researchers who used an estimation of what this velocity planning step would do to their input GCodes to then encode their custom controls on top of that time encoding, to align the two representations. In MAXL, we also use time-based encodings to improve how modular code can interact with trajectories.

This leads to another point, that GCode interpreters are buffered. In order to optimize over velocities, they need to look ahead. Whichever other system is sending them GCodes (sometimes a serial port, sometimes they are read from e.g. an SD card) is not always guaranteed to have exceptional bandwidth, and so interpreters must keep a number of moves in their queues, a bit larger than they may need to plan over. This makes it difficult to build interpreters that are responsive to new inputs: we can only insert new GCodes at the end of this queue, and must wait

Listing 2.3: This is continued in Listing 2.4. Where interpreters use compile-time configurations to adapt across machines, it can be difficult to ascertain what our machines will do for each different code — even when we can see the source code, as in this case where many potential code paths are available, and are switched using compiler flags like `#if ENABLED()` etc.

```
// in Marlin/src/GCode/control/M3-M5.cpp
void GCodeSuite::M3_M4(const bool is_M4) {
  #if LASER_SAFETY_TIMEOUT_MS > 0
    // Reset timeout to allow subsequent G-code to power the laser (imm.)
    reset_stepper_timeout();
  #endif

  if (cutter.cutter_mode == CUTTER_MODE_STANDARD)
    // Wait for previous movement commands
    // (G0/G1/G2/G3) to complete before changing power
    planner.synchronize();

  #if ENABLED(LASER_FEATURE)
    if (parser.seen_test('I')) {
      cutter.cutter_mode = is_M4 ? CUTTER_MODE_DYNAMIC : CUTTER_MODE_CONTINUOUS;
      cutter.inline_power(0);
      cutter.set_enabled(true);
    }
  #endif

  auto get_s_power = [] {
    if (parser.seenval('S')) {
      const float v = parser.value_float();
      cutter.menuPower = cutter.unitPower = TERN(
          LASER_POWER_TRAP, constrain( v, 0, CUTTER_POWER_MAX),
          cutter.power_to_range(v)
      );
    }
    else if (parser.seenval('O')) { // pwr in PWM units
      const float v = parser.value_float();
      cutter.menuPower = cutter.unitPower = CUTTER_PWM_TO_SPWR(
          constrain(v, 0, 255)
      );
    }
    else if (cutter.cutter_mode == CUTTER_MODE_STANDARD)
      cutter.menuPower = cutter.unitPower =
          cutter.cpwr_to_upwr(SPEED_POWER_STARTUP);

    // PWM not implied, power converted to OCR
    // from unit definition and on/off if not PWM.
    cutter.power = TERN(SPINDLE_LASER_USE_PWM,
                        cutter.upower_to_ocr(cutter.unitPower),
                        cutter.unitPower > 0 ? 255 : 0);
  };
```

for previous inputs to be "flushed" (actually run) before the new command does anything. Many interpreters have simpler handles to (for example) change the velocity of the machine without

Listing 2.4: Continued from Listing 2.3.

```
  if (cutter.cutter_mode == CUTTER_MODE_CONTINUOUS ||
        cutter.cutter_mode == CUTTER_MODE_DYNAMIC)
      {  // Laser power in inline mode
      #if ENABLED(LASER_FEATURE)
      // M3 or M4 is powered either way
      planner.laser_inline.status.isPowered = true;
      get_s_power(); // Update cutter.power if seen
      #if ENABLED(LASER_POWER_SYNC)
        // With power sync we only set power so it does not effect
        // queued inline power sets
        // Send the flag, queueing inline power
        planner.buffer_sync_block(BLOCK_BIT_LASER_PWR);
      #else
        planner.synchronize();
        cutter.inline_power(cutter.power);
      #endif
    #endif
  }
  else {
    cutter.set_enabled(true);
    get_s_power();
    cutter.apply_power(
      #if ENABLED(SPINDLE_SERVO)
        cutter.unitPower
      #elif ENABLED(SPINDLE_LASER_USE_PWM)
        cutter.upower_to_ocr(cutter.unitPower)
      #else
        cutter.unitPower > 0 ? 255 : 0
      #endif
    );
    TERN_(SPINDLE_CHANGE_DIR, cutter.set_reverse(is_M4));
  }
}
```

waiting for a GCode to move through the queue, but we cannot change the spatial trajectory itself without halting and restarting the machine.

Based on this look at how a *standard* GCode is implemented, we can see what steps would be required to write a *new* GCode that extends machine operation:

1. First, write the low-level firmware driver that runs the new piece of hardware. This may involve modifying an existing GCode interpreting control board (re-purposing some of its outputs and inputs), in which case the source code needs to be available. If the appropriate low-level drivers are not available on any of these boards, we need to develop a new circuit and firmware.
2. Define a new GCode, picking something like `Mxx` and ensure that the instruction does not collide with others in publicly available tables.
    a. Write down what that GCode means,
    b. Integrate the new firmware with the existing firmware's interpreter and planner. In some cases the additional compute power required by this new functionality may introduce issues at runtime.

3. Integrate that new GCode with whichever program is being used to write machine instructions, either a CAM post processor or perhaps a custom Python code.

There are many better approaches to this problem in research communities that I will cover in the next sections (after a broader look at systems architectures more generally), but GCode interpreters remain extremely common in practice.

2.2.2 Background on Networks

2.2.2.1 Network Constraints and Benefits in Control

In Section 1.4.2 I mentioned that *communication* between control layers is a key constraint on our ability to partition systems in the ways we might like; controllers have bandwidth and timing requirements, but networks have bandwidth limits and delay. Operation over networks also requires some extra computing power to serialize, transmit, receive, route, and deserialize messages. This means that no matter how much engineering we do in our network layers, distributed controllers will always be less performant and less *deterministic* than monolithic controllers where all elements are in the same CPU.

For a simple example imagine a control loop that drives a motor according to readings from an encoder. We could build this using a distributed system with i.e. two $48MHz$ microcontrollers (one CPU for the encoder and one for the motor) or in a monolithic system with one $94MHz$ microcontroller performing both tasks. The second option will have superior performance because moving data from one process to another in the same CPU is always faster and more reliable than sending it over a network; network delay is directly related to controller bandwidth. Networks are also variable: if a link is congested its performance will decrease non-linearly (i.e. slowly at first and then all at once)[44]. Out of band disturbances i.e. noise in the surrounding electromagnetic environment can cause packet loss[45] and networks can be degraded by unrelated computing loads: if a microcontroller is busy computing new control values it has less time to process messages. So, network variability is directly related to controller determinism: moving data between two processes in one CPU is *much* more reliable than moving data across a network link. Each of these constraints is discussed in [79] and [80], and network performance is evaluated with regard to controller performance in [81]. These show that networks must be designed *alongside* controllers to understand total system performance.

On the other hand, the distributed controller in this simple example is more modular: we could re-use the encoder module with different motor drivers, or vice versa. This is sometimes a more

[44]Networks tend to *collapse* when they approach their maximum utilization. The typical pattern is that increasing congestion starts to cause packet loss, after which transport layer algorithms begin queuing extra messages (retransmits), further increasing congestion. Once everyone starts doing this, links are quickly saturated and performance bottlenecks. Much work has gone into developing transport algorithms that intelligently avoid and recover from these scenarios, i.e. TCP New Vegas (which uses packet delay, rather than packet loss, as a flow-control signal).

[45]I was once debugging a packet loss issue for nearly half of a day before I realized that the cable (containing UART over RS485) was lying on top of a switching power supply that was emitting noise in around the same frequency of the link's bitrate. I moved the cable and the performance was restored. Wireless links are the same: too many cellphones in a room and your bluetooth headphones or WiFi performance will degrade.

valuable property than raw performance. In larger systems distributed controllers can provide pure performance benefits as well. For example if a controller needs to sample e.g. six different sensors before generating a new output signal it may be faster to distribute each of those sensor readings to a standalone microcontroller for each and then collect readings over a network. This is especially true where the total compute required by a control program exceeds the power that is available on any single microcontroller.[46]

The exact "break-even" point for these partitioning outcomes is dependent on constraints from the networks, the available CPUs, and the particular control task. The problem of optimal partitioning for distributed systems is handled directly by [82] for the general case (distributed computing tasks) and for realtime controllers in particular by [83].

Solutions to these problems also extend into our controllers' *mathematic* architecture as well; we can "trade computation for bandwidth" by distributing predictive models throughout a system [84], relying on those models for intermediate estimates of distributed system states. Similar patterns appear when partitioning parallel algorithms over compute cores (using optimal partitioning to minimize memory contention [85]) and in edge computing for mobile device networks [86].

These ideas are most relevant in this work where we want to align motion control trajectories generated in one device with their execution in other devices. If they were fast enough, we could rely on network speed alone to synchronize devices. However, the links between path planners and motor controllers and sensors introduce real delay and indeterminacy and so we need to develop a strategy that will work well despite those limits. Each of the strategies that I cover in Section 2.2.4 manages this differently.

These constraints do not only apply to classically "networked" controllers: even monolithic GCode interpreters that receive new GCodes over i.e. a USB cable face the same issues. I expand on that in Section 2.5.2 alongside notes on how this background informs MAXL's design.

Developing networks for real-time systems is a common challenge and so there is well established practice. For example OSAP borrows the "earliest deadline" scheduling pattern from [87] (see 2.3.1), and I use some key insights on clock synchronization from Network Time Protocol (NTP [88]), its high-performance counterpart Precision Time Protocol (PTP [69]) and other simple approaches for clock sync [89], [90]: using simple diffusion and control laws for clock discipline[47] (see 2.3.2.2).

2.2.2.2 IP vs. FieldBus vs. Source Routing Networks

Much of our networks' complexity comes from the routing layer. IP networks use "destination routing" where packets contain a destination address that is globally resolvable and the network itself resolves the route at runtime. To do this they keep tables that map ports to addresses and

[46]For an example from this thesis, the motor controllers (Section 4.4) are using *all* of their available 200MHz to servo the motors around. If we tried to stick the motion controller in there as well (and more motors), things would explode. This is additionally true for compute in general: datacenters and supercomputers rely on networks to expand compute volume beyond what is available on a single die.

[47]"Clock Discipline:" a term of art meaning the strategy with which we actually skew our local clock against the network clock.

learn which ports point towards which destinations over time as they receive packets from those destinations. To forward messages routers read recipient addresses in packet headers, lookup which port those recipients are available on, and then copy the message onto the appropriate port [91]. Performing these steps efficiently can be complex and routing tables can consume large amounts of memory [92], [93] but in well-developed network stacks (i.e. the internet) the problem has been exceptionally well managed and is done one dedicated hardware [94], [95], [96].

Some efforts have been made to compress these algorithms for smaller devices [97] but many embedded networks instead use simpler *busses* where the routing step is skipped entirely, in networked control these are called FieldBusses, for example I mentioned CAN bus [64] as a primary example in the introduction to this chapter (2.1.2). On a bus, all devices receive every message and they simply ignore packets that are not intended for them. Some busses specify that each device on the bus has a unique address, whereas others ignore individual addressing entirely and delineate packets purely based on the data that they contain — i.e. the network model and data model are completely collapsed. Busses have a scalability downside because all devices need to share the same amount of bus bandwidth (they share one transmission "medium") and a computing downside because each device needs to listen to, delineate and read each message even if those messages are not intended for them whereas point-to-point networks can be organized into subtrees of local traffic. Busses can also be less deterministic for this reason: if a number of devices try to transmit a packet at the same time their packets would collide, so each bus must develop a Media Access Control (MAC) strategy for collision avoidance. EtherCAT's innovation was to use a bus network topology with a point-to-point "ring" link topology: packets travel *along* the ring where they are read by each device, and each device forwards them while inserting new data into the same packet [67]. They use standard Ethernet PHYs (Physical Layers) for this, but specialized Network Interface Controllers (NICs) to manage the low-latency packet forwarding / data insertion step. See Figure 3.12 and the surrounding text in Section 2.6.1.2 for more on that step. Busses have the advantage that devices can receive broadcast (aka multicast) messages in sync which is especially useful in control where we may want to synchronously send i.e. all motors in a controller new control values.

I should also note that switched Ethernet networks have also been adopted for networked control [98]. Switched Ethernet is an interesting example of the bumpy application of the OSI model and "sticky" / high inertia adoption of working technologies; Ethernet was originally developed in the 1970s for shared-media links over coaxial cables, and so it specifies that any device have a globally (as in, worldwide) unique MAC address. In large systems this led to the same scaling issues for shared-media busses that I mentioned above, and so switched Ethernet was developed where transparent switching devices are added between individually point-to-point Ethernet links. Like IP routers, these also learn and store routing tables in order to forward messages along individual links. This means that like EtherCAT, these systems *look* like a bus (and include multicast capability) but their physical topology is point-to-point, eliminating most of those scaling issues. Because switched Ethernet hardware was developed for wide use across datacenters and consumer networking, they are cheaper to integrate than EtherCAT and only slightly less performant. For a final note, Ethernet hardware is still challenging for a small microcontroller to operate and

only high end μc's have this capability, and even then require external physical layer hardware to actually drive the signal. Integration on switched Ethernet networks also requires that each device has a MAC address, and again not every microcontroller is given one - they are allocated to device manufacturers in address space blocks that must be licensed by the IEEE Registration Authority to maintain their global uniqueness.

One alternative to both is *source routing* [99]. This was first introduced to me by my advisor Neil Gershenfeld who developed a scheme called APA (Asynchronous Packet Automata), a description of which is in Nadya Peek's thesis ([20] Section 3.1). In source routed networks, intermediate devices are mostly stateless; packets themselves contain routing instructions rather than destination addresses. This makes for simpler routers and networks overall and enables more precise network configuration, but requires that devices know where their recipients are because *they* write the routing instructions. This requires careful configuration and a deep understanding of the network, whereas destination-routed networking is simpler from the device programming standpoint; if you know your intended recipient's address you just write that into the packet header and the job is done.

Source routing is used today where networks designers want explicit control over routing paths, simple and more robust router designs, and high performance. For example SpaceWire [100] source routes data between sensors, processing units and telemetry for on-orbit scientific instruments and is maintained as a standard by the European Space Agency. Source routing is especially important in space because it allows for the definition multiple routes between the same two endpoints such that if a device along one route flips a bit due to stray radiation the change can be detected and either corrected or retransmitted. The simplicity also makes it easier to deploy on robust and deterministic computing systems like FPGAs. SpaceWire is overall the most similar architecturally to OSAP among currently practiced network strategies, I make a more complete comparison between the two in Section 2.6.1.3. Source routing is also used in supercomputers to connect partitioned CPU cores [101] and in "network on a chip" systems for high performance interconnect of local processes [102], [103].

OSAP uses source routing for most of the same reasons: machine networks are local and so security is less of a concern, routers should be simple, performance is important, and direct configuration of routes is a valuable tool for the design of networked controllers as I've just explained in 2.2.2.1. In trading away complexity *in* the network, it does add some complexity to network configuration because routes must be configured using a view of the network topology that includes the transmitter and receiver (so that a router can be mapped between them). In OSAP, that goes hand-in-hand with the automatic detection of network topologies that is also useful to ascertain hardware configurations. I discuss that step in Section 2.3.2.1 and show how it is used in PIPES to connect software modules together in 2.4.4.3, where data routes for the program are written directly on top of source routes — this lets us directly configure network traffic at a program level.

Source routing is also related to strategies for larger scale network optimizations that I will discuss under the next heading (2.2.2.3), but it has fundamental security issues in open networks because

transmitters have authority over where their packet lands; this vulnerability is what makes it uncommon for everyday networking.

2.2.2.3 Network Design as Optimal Partitioning

We can think of network design itself as an optimal partitioning problem, in Chiang's excellent paper [104] network operation is explicitly framed as a distributed control problem that can be subjected to optimization-based design and operation. In this case, the problem is to choose where (i.e. in which protocol layer or in which device) different parts of the networking algorithms should go: for transport control, flow control, routing, etc. This is the framing that is adopted by the Next Generation Protocol working group [63].

In another view we also have the problem of optimizing traffic within a network, i.e. routing data flows over a constrained set of available paths. Source routing is one of the choice tools in this domain, it has seen a resurgence in carrier-scale network optimization where packets are *partially* source routed in a scheme called "segment routing" [105]. Network optimizations are also common in datacenters or in large local networks (on i.e. a university campus). This is where Software Defined Networking (SDN) is most common. In SDN, network topologies (maps!) are used by centralized planners to optimize flows within networks. This was originally introduced with OpenFlow [106], a tool that allowed network administrators to remotely configure network hardware by hand for efficiency experiments, but that capability quickly led to the development of "policy based routing" [107], [108] where optimal network flows are described declaratively by administrators and then configured with software that uses realtime data on network traffic to measure flows - a fairly explicit form of constraint-based optimization. These approaches are used extensively by i.e. Google [109] and Microsoft, whose overall datacenter architecture is described very well in [110] using policies from [111] and remotely reconfigurable FPGA routing hardware from [112].

OSAP and PIPES do not come anywhere near this level of complexity, but I think it is interesting to keep in mind as we design networks: again, we see a relationship between *global system oversight* for configuration and remote reconfiguration of lower level hardware for realtime operation - and of course the idea that constraint-based optimization of even the *networking* components in our systems is interesting.

2.2.3 Background on Distributed Programming Models

Next I want to look briefly at distributed programming models in a broader window, before focussing down in the next section on machine-specific architectures and programming models in 2.2.4.

There is a huge amount of work in this domain. I will try to show that most approaches here are suited *either* for high-level systems *or* for embedded systems; i.e. at the same realtime boundary where GCode sits. In these distributed systems, bridging across that gap is largely still done using *middlewares.*

The umbrella keyword for systems integration for hardware is Cyber Physical Systems (CPS), an up-to-date review of which is in [113]; this gives a sense of the breadth of concerns that are relevant for systems design. A major tension in the field is around determinacy, [114] notes that most of these systems are combined in a way that determinacy is not preserved, but shows that more complete descriptions of CPS could enable rigorous evaluation of their determinism. That means modelling networks, data flows, and computations within a system's components - a fairly heterogeneous set of descriptors. A key challenge in this regard is ascertaining overall state in any given CPS. On the networking side this is related to the prior two sections where we looked at the importance of modelling networks themselves. At the application layer the focus is on "plug-and-produce" systems that propose automatic discovery of device descriptors and data models [115], [116].

2.2.3.1 Message Passing Middlewares

So, the key capability of these systems is *message passing.* This seems like it should be simple but it requires that messages are routed according to a system designer's wish (which requires discovering network configurations and naming devices and messages) and that messages are encoded in a manner that each device can read and write them. There are four systems that are worth talking about in this regard, each with some subtle and some more substantial differences. These are each known as *middlewares* (see [117], Section 1.1[48] and the original description in [118]).

MQTT [119] is the simplest. Devices publish and/or subscribe (pub/sub) to "topics" that have human-readable names and payloads are routed using centralized "brokers," it is most common in IoT networks and most payloads are serialized in JSON, but any binary format is possible. Network configuration is simple: each device is configured to point to the broker which normally has a static IP address.

ZeroMQ [120] has pub/sub semantics but is more akin to "sockets on steroids" (their wording), and is decentralized by default; devices can discover one another using multicast packets. ZeroMQ does not specify data formats, that is left up to the programmer.

OMG-DDS (Object Management Group's Distributed Data System [121]) is most similar to ZeroMQ but *does* enforce data formats, which enables automatic discovery. In this system, programmers specify which data they will pub/sub to under a schema for typed topics (in the OMG-IDL, for Interface Definition Language) and then devices discover one another at runtime and send messages according to the schemas found in the local network (i.e. sending data to neighbors who specify that they would like to subscribe to their stream).

The OPC Unified Architecture (OPC-UA [122]) was developed specifically for industrial equipment and is most similar programmatically to OMG-DDS but includes built-in data schemas that are

[48] A clear explanation of *middleware* from the Distributed Systems textbook [117]: *"To assist in the development of distributed applications, distributed systems are often organized to have a separate layer of software that is logically placed on top of the respective operating systems of the computers that are part of the system [...] In a sense, middleware is the same to a distributed system as what an operating system is to a computer: a manager of resources offering its applications to efficiently share and deploy those resources across a network."*

maintained by OPC. OPC-UA systems engineers *can* still define their own types, but the point of the specification is to enable interoperability using shared and standardized descriptors.

A key note is that each of these systems relies on IP-based network backbones and their devices' local operating systems to operate those networks. To interact with hardware directly via embedded processors, *bridges* are developed that pass data over simpler link layers into operating systems that run these more advanced middlewares. See [123], [124], [125] and [126], and a direct comparison to OSAP's inclusion of embedded devices in Section 2.6.1.2.

For a better look at these systems, this evaluation paper [127] summarizes their differences in more detail and measures relative performance (MQTT is the slowest overall, likely due to its reliance on the centralized broker). I should also note that DDS also has an extensive interface to configure Quality of Service (QoS) for each topic, i.e. specifying transmission deadlines, priorities and delivery guarantees - these are essentially transport layer configurations.

2.2.3.2 ROS: Application Layer Middleware via Message Passing

Of course we also have to talk about ROS, the Robot Operating System [128] - specifically the new implementation ROS2 [129]. It is a "a set of software libraries and tools" for robot applications; another middleware but its purpose extends beyond messaging alone to help robot systems authors integrate sensing, simulation, and planning softwares. In ROS, DDS is used for message passing but individual ROS "nodes" (blocks of software) expose pre-defined APIs using community standard[49] message formats. Software packages in the ecosystem connect to one another through this framework and systems authors build custom applications by composing these packages, authoring their own middlewares to connect them. Representations of physical robots are built using REPs (for ROS Enhancement Proposals), which are more formal standards to describe robot kinematics and i.e. units of measurement. To connect to hardware, ROS also provides a litany of *drivers,* these are ROS nodes that wrap existing OS-level interfaces to devices and exposes them as ROS topics, services and parameters.

In many ways, ROS is similar to GCode: it is a middle layer between hardware and software that is built using convention but is not strictly typed or inspectable. Using ROS is a *practice* and requires users have some tacit knowledge of its internal architecture, principles and conventions [130]. One-third of ROS bugs are dependency errors [131] that arise from hidden misconfigurations where node or user-code is set up according to varying or out-of-date conventions [132], discovery of configurations is not itself built into ROS, but some work adds additional tooling for this [133], [134].

2.2.3.3 Serialization Layers

Each of these middlewares must serialize messages: this means taking data out of memory in the transmitter's CPU and writing it into a format "on the wire" that can be read by the reader in a standard format.

[49] Message formats in ROS are not *strictly* standardized, they are more akin to naming conventions, for example `/tf` for robot transforms, `/scan` for laser scans, `/joint_states` for joint positions. Software packages are set up by default to look in these pseudo-standard locations, but can be configured otherwise.

The most common serialization format is JSON (JavaScript Object Notation), which serializes structured data into human-readable strings. It is commonly used in MQTT and is extremely flexible, but serializing strings is computationally expensive and also memory intensive [135]. More performant systems use compiled data protocols like ProtoBuf [136] or Capn' Proto, both of which are compared in [137], [138]. These are exceptionally performant *but* rely on shared schemas that are built before systems are compiled: programmers must specify data types ahead of their use and without access to these schema (i.e. in a `.proto` file) it is difficult to inspect them remotely, although they do provide tools for automatic documentation and some extensions allow for runtime reflection on types.

2.2.3.4 Dataflow for Hardware

There are three systems worth discussing that each implement dataflow programming models for hardware.

LabVIEW is the longest standing and most widely used dataflow tool for hardware. It was developed by National Instruments for the development of new laboratory equipment and allows scientists and engineers to compose programs using a mixture of software components with "virtual instruments" [139]. Virtual instruments are implemented in much the same way as other embedded devices described in the subsection above on middlewares: OS-level drivers communicate with hardware via bridges over USB, serialport or custom communication layers and expose those devices within the LabVIEW runtime itself via an "agent" that virtualizes them. For this reason it, like others, is limited in computational scalability by the one standalone dataflow runtime: messages queued between two hardware instruments must pass through this domain through those bridges.

Node-RED is used in IoT [140], [141] to compose software for "collecting, transforming and visualizing data." It ingests messages over MQTT or HTTP and can broadcast transformed data over the same protocols. Systems are composed in flows where each block is a wrapper on a JavaScript function or class. While it connects to distributed devices, it is not itself distributed: blocks operate in a single runtime. Connecting two node-red runtimes into one system requires that each manually configures MQTT inputs and outputs, and that those be configured to communicate with one another, and discovery of the global system from some other location is not possible.

MsgFlo [142] also runs on MQTT (or AMQP: Advanced Message Queuing Protocol [143]) but is more distributed: each process / device is independent and can receive or transmit message flows, capabilities of remote devices can be automatically discovered, and dataflow graphs can be composed across those devices. MsgFlo is most similar to PIPES in this regard, but differs in a few key ways that I will explain in Section 2.6.1.5.

2.2.3.5 Patterns from the Web

Each of these more modern approaches to distributed systems borrow heavily from design patterns that were originally developed for the Web, the largest and messiest distributed system of all time! There are three relevant ideas that I would like to cover.

The first is organized around REST for Representational State Transfer, introduce by Fielding [144], [145]. This is a widely-used architectural *style* for web APIs that encourages simpler

interoperation of online services. It encourages a few key principles: statelessness in the interface, layering of server functions, and unified interfaces using Unique Resource IDs (URIs) and a core set of operations. The style itself doesn't specify anything about the particular implementation, but it is most often followed for HTTP messaging on the web where the core operations are: GET (read), POST/PUT (create or update), PATCH (partial update) and DELETE. Each request is made to a resource *endpoint* (a URI) alongside the operation keyword, a header (metadata and context, i.e. state) and a body (the payload). In many ways, REST APIs are the middleware of the web.

A common programming framework for RESTful APIs is the Model-View Controller (MVC) originally developed for the smalltalk language [146] they are widely adopted in web applications [147]. Here the *model* is whichever underlying representation is being interacted with: a database or user session on the web or (in the case of PIPES) a machine configuration. The *controller* mediates between the user and the model i.e. allowing and implementing or denying requests to make changes to that model, and the *view* is what is rendered for the user.[50] OSAP and PIPES both use this framework; controllers in each system return views of their *local* models for network state (via OSAP, 2.3.2.1) and for program state (via PIPES, 2.4.4.1). These are then combined into a global view of the PIPES system model (Section 2.4.4.1), which is the main interface for programming in PIPES systems.

RESTful APIs are often used by *microservice* style server architectures, where requests made to datacenter-scale systems are handled using modular systems composed of smaller programs, each of which provides some subset of the contents of the reply [148]. These allow more rapid development of server programs because components can be replaced piecewise and i.e. authored in different languages: in fast `cpp` for complex tasks or in scripted languages like JavaScript or Python for simpler or rapidly developed features. The same is true for updating large distributed systems: rather than updating the entirety of a system's code at once, individual instances in the stack can be updated one at a time or i.e. tested partially before deploying completely. Microservices *also* allow for systems-scale optimization of compute resources: rather than loading one monolithic piece of code in each server, services can be spun-up depending on which are needed most often according to current loads. This level of configuration connects to network-scale optimizations discussed in Section 2.2.2.3 where data flows are optimized between processes, at this layer the actual layout of those processes can be orchestrated. So, again, we see this correlation between the optimization of program *and* network configurations in distributed systems and their coupling.

For one final note in relation to the *economic* partitioning of systems and how that relates to technical partitionings (from Section 1.4), microservices also serve to bundle *organizational* capabilities into standalone software components; teams that are responsible for a particular component of

[50]For an example of an MVC interaction: you (the user) request to trade $1000 USD of virtual money in your bank's ledger for cash at an ATM. The ATM sends a request to the controller in the bank's backend software. It checks if you have this many virtual dollars in the ledger and if you do (and you have authenticated yourself), it subtracts those from the ledger (updating its model) and returns a new view to the ATM: your updated bank balance, and then a subsequent *request* to the ATM's controller to dispense the dollar bills. The ATM dispenses the cash and then updates *its* internal model of its cash stores.

a firm's technical systems are also responsible for its integration into their server's backend via a microservice (see [114], Section 3.1). This simplifies organizational complexity within the firm: that team does not have to explain their system's internal organization to their collaborators - they just have to specify the interface. This idea is carried into OSAP / PIPES and MAXL: in Section 8.3 I relate the architectural choices made in these systems to the same organizational complexity tasks faced in open source hardware.

2.2.4 Background on Machine Control Architectures

Continuing on a path away from pure architectural background through real systems, in this section I want to discuss a few common machine control architectures from industrial control to hobby 3D printing and in related research. Under these headings I will discuss how each is arranged computationally (i.e. which controller function is located in which computing device) and how those arrangements relate to the myriad ways that programmers and users configure and operate machine systems.

2.2.4.1 Centralized, Monolithic Control

The simplest and most common arrangement is the monolithic / centralized GCode interpreter. One of these is shown in Figure 3.2 on a Prusa FFF Printer [149]. These boards run a single firmware that consumes GCodes via a serialport, WiFi or an SD Card. They interpret those codes, queue them into buffers for velocity planning (see Section 4.2.4 for a longer explanation of the velocity planning step), and continuously update their queues and velocity plans as they simultaneously operate hardware like stepper motor drivers, heaters, and spindle controllers via low-level drivers.

2.2.4.2 Centralized Timing and Control with Remote Devices

In most state-of-the-art industrial machines a central microcontroller or PC running a realtime operating system performs computational tasks like GCode interpretation, velocity planning, and some components of servo feedback control and sends commands to motor driver hardware over simple network links or fieldbusses like EtherCAT.

Drive commands can be sent in a variety of forms. Splines are sometimes used because they cleanly encode position, velocity, and acceleration signals and can be interpolated at rates that exceed the network's bandwidth. I rely on splines in MAXL for the same reason (see Section 2.5.3). Some systems transmit piecewise linear segments for motor interpolation, and others send simpler sample-and-hold reference signals for motor velocity. In each of these cases, a faster controller in the servo drivers themselves operates the motor hardware at much faster bandwidths, normally between 500 and 2000Hz depending on the system dynamics.

These networked controllers are very simple distributed systems but rarely deploy the types of middleware that I surveyed in Section 2.2.3, instead they write simpler communications layers between GCode interpreters and fieldbus protocols.

This pattern is being adopted in open source, especially by the Duet ecosystem [75] which runs RepRapFirmware [76] and uses a CAN fieldbus to connect interpreters to auxiliary drivers; the Duet mainboard is still mostly a monolithic controller but can be extended under this pattern.

2.2.4.3 Object Oriented Hardware

I briefly introduced OOH from Peek [20] and Moyer [71] in the very beginning of this section (2.2). These controllers virtualize control and configuration by moving core logic into an OS where machines can be programmed using virtual representations of hardware and machine control. It also eschews GCode interpretation and instead presents a more modular machine API that can be used to develop machines as realtime-interactive devices and to better integrate motion control with user applications.

OOH's main innovation is in the way machine configurations are expressed as modular software APIs, overlaying object oriented programming with modular hardware objects. Peek shows in [20] how this framework can be used by new machine designers to configure and control hardware of their own design using more flexible representations of machine control and interacting with controllers dynamically via Python scripts.

In terms of computational partitioning is would also fall in the class above: it centralizes program interpretation and velocity planning and sends lower-level motion segments to motors over a fieldbus. In this case the fieldbus is a custom implementation called FabNET.

2.2.4.4 Klipper

Klipper [58] is an interesting hybrid system that follows most closely in-line with Object Oriented Hardware controllers.

Klipper's main logic (GCode interpreting, velocity planning and optimization of stepper motor pulse timing) runs in the Linux operating system and is normally deployed on small single-board computers (SBCs). It's use of higher power computing available in SBCs enables it to build more advanced controllers; Klipper was the first open source system to implement *input shaping,* a filter-based velocity planning step that minimizes excitations of machines' resonant modes (see Section 4.2.5 for background on input shaping).

Klipper is fundamentally based on step-and-direction based control of stepper motors. Precise and high bandwidth timing of these step pulses is required to smoothly operate stepper motors, the likes of which are difficult to generate within an operating system. Klipper offloads this control component (and control of other auxiliary hardware) to embedded devices and communicates with them over USB links using a custom and very low level protocol. This allows Klipper systems to operate stepper motors' step and direction pins at extremely high frequency,[51] 223 kHz on SAMD21 microcontrollers (with a 48 MHz clock) and up to 885 kHz on RP2350 microcontrollers (a more modern device with a 150 MHz clock) [150].

[51] High speed stepper pulse timing is an important performance metric for these systems, as I discuss in some more detail in Section 7.4.2.

While it is authored in Python and so easier to edit, Klipper is still a GCode interpreter and so extending it to modify control logic itself is nontrivial. Configuration is similar programmatically to other interpreters (a static `config.py` file containing controller parameters and options is modified) but is simpler to update because it does not require that the interpreter be re-compiled and re-flashed into firmware.

2.2.4.5 Urumbu

Urumbu was developed by my collaborator Quentin Bolsée and my advisor Neil Gershenfeld [57]. Like Klipper and OOH, it centralizes program state, configuration and velocity control into a Python program running on an operating system. Urumbu uses Python threads to increase performance of this software and in doing so can perform even very low-level timing of stepper pulses there without a realtime OS; one of the threads runs a tight loop that uses the OS clock to coordinate steps. It connects to devices over USB links using USB hubs and sends single-byte instructions to motors and sensors.

This makes embedded device development incredibly simple: each only has to translate single bytes into hardware outputs. For stepper motors, this amounts to one step per instruction using a bitmask where the 7th and 8th bits in the byte correspond to the stepper driver's step and direction pins. Devices can optionally send one return byte in each cycle to read i.e. switch states. However, it limits performance according to the operating system's USB driver performance. While this can be surprisingly fast (at 2000 Hz) it is not deterministic and the low-level representation means that stepper motors' stepping rate is limited to this frequency whereas high performance stepper drivers operate well above 100 kHz .

2.2.4.6 StepDance

Ilan Moyer's StepDance [151] is a motion control framework that aims to help machine designers develop new automated and real-time interactive machine tools, with a particular focus on craft-aligned interactions.

StepDance is similar to MAXL in that it describes machine controllers as *flows* of data streams that can be declaratively mapped and that controller configuration is based mostly on the arrangements of these flows. It also includes a library of dataflow blocks for kinematic transforms and function generators, a-la MAXL's own set (in 2.5.4). Rather than encoding motion in basis splines, it uses pure step and direction streams. It runs entirely in firmware at a core step tick rate of 25 kHz but multiple modules can be connected together using a custom physical link that encodes multiple axes of step and direction using pulse-width multiplexing over four-conductor audio cables. In earlier work by the same authors [152], a similar system based on step and direction signals is mixed with a classical GCode interpreter and velocity planner (Duet3D [75]).

StepDance is architecturally novel, it partitions motion control over multiple devices at extremely low latency by eschewing networks entirely. Instead they build a custom physical link that is packet-free; each digital pulse on this link encodes step and direction directly, although they do use pulse width *multiplexing* to encode multiple channels per physical link. The architectural through-line is

to align the whole system around stepper motor drivers' core representation for motion (discrete increments… steps) and broadcast those through a new network and software architecture.

That is perhaps also its weakness: these links cannot transmit arbitrary network data that would enable remote software control of kinematic blocks via i.e. remote procedure calls or interfacing with more traditional user interfaces or remote configuration tools. Control of inputs and outputs that are not step-based is also requires some extra steps, but they are well managed using i.e. a software block that converts analog inputs to steps using a velocity model and another that can convert step and direction integrated positions to a hobby servo's reference signal.

In Section 7.4 I discuss the relative benefits and trade-offs that emerge when we use splines as a core representation, including the ability to use the spline's natural derivatives for velocity and acceleration for improved closed-loop tracking, how spline representations allow for improved stepper interpolation, and their natural relationship to underlying machine physics.

Another difference between StepDance and the network-based controllers in this chapter is that we can re-create the control graph's state remotely by inspection over the network to produce a global system model (2.4.4.1). Understanding global configuration of a StepDance system would require understanding each module's source code and physical link mappings.

However, StepDance's realtime performance surpasses that of the systems in this chapter, it's 25 kHz is equivalent to only $40\mu s$ delay. In Section 2.7.9 I show that in the stackup of MAXL, PIPES and OSAP the equivalent delay between two embedded devices is about $400\mu s$ or a 2.5 kHz update rate, see Section 2.7.9 for a complete workup of timing results from this work. However, this is in the case where we are updating *basis spline* control points, so it represents an update rate but not an interpolation (stepping) rate, whose performance would be dependent on the interpolating module's performance. In practice, I run the stepper motors (from 4.4) *closed loop* update at 15 kHz which requires substantially more computing power than just stepping. But I also push most control elements into the operating system, and MAXL is based on a deterministic timing gap (see Section 2.5.2) between the OS and path execution. I discuss this also in 2.7.9: real-world delays due to this gap are typically $64ms$, this is only (15.625 Hz) and above the threshold for human-perceptible delay for interactive computer systems i.e. drawing on a tablet with a stylus [153], but under the threshold for closed-loop teleoperation of robotic systems through the human visuomotor loop which is around $250ms$ [154].

2.2.4.7 Computational Control of Machines via GCode Wrappers

Many researchers choose simply to wrap GCode interpreters in improved, dynamic APIs. These are normally Python codes that connect to interpreters over a serial port, they expose a computational API on one side and write GCodes into the interpreter on the other. Frikk Fossdal in [155] and [156] develops interactive machine interfaces in Grasshopper (a dataflow programming tool that is integrated with CAD) using a Python script as an intermediary to send GCodes to an off-the-shelf machine controller, it also reads machine configuration states from the controller using a separate link available in the Duet3D ecosystem.

Hannah Twigg-Smith's Dynamic Toolchains [157] and Peek and Gershenfeld's `mods` [74] systems also use dataflow to control machines, but more indirectly: they are designed to write toolpath plans in a more modular fashion, but then render those toolpaths as GCode (mods and Dynamic Toolchains) or machine knitting instructions (Dynamic Toolchains) and transmit them to interpreters in off-the-shelf machines.

In Jasper Tran's "Imprimer" system [158], computational notebooks are used as an interface for machine workflows: their system also implements an intermediary software object that communicates with off-the-shelf controllers using GCode, but presents a more useful API to the notebook and uses an additional intermediary representation to keep track of machine state.

The Jubilee project [159], [160] is a machine platform that implements a modular tool-changer, and has been successfully deployed by researchers to automate duckweed studies (a popular model organism) [13] and to study nanoparticles [14]. Jubilee *also* uses an intermediary Python object to interface with an off-the-shelf GCode controller.

Each of these works shows the value of integrating motion systems with application-layer scripting languages, but they are sometimes limited by the integration strategy. At the configuration level, each Python interface in these examples must mirror the GCode interpreters configuration in its own local state, this process is done by hand and misalignments can cause difficult to diagnose errors. Consolidating configuration state was a topic discussed during and NSF sponsored workshop that I attended on open source lab automation tools [161] where we used Jubilee machines to build new workflows. In terms of motion control, it is difficult for these tools to interact directly with as-planned machine trajectories because the GCode interpreters do not expose those. Besides the configuration challenges that can be resolved with careful engineering, this is not a real issue for the duckweed and nanoparticle studies or in any other case where moving to a position and then performing the new automation while the machine is stopped is sufficient, but presents challenges for researchers who integrate new process physics like those in the next subheading here and introduces delay and possible errors in shared digital representations of these machines. Each of these studies is also motivated by the idea that machines should be easier to modify for non-experts, so improving the configuration step remains important especially if we would like to apply them on a broader set of individual machines, each of which has a unique configuration.

2.2.4.8 T-Codes and FullControl

Two groups of researchers working on control of fluidic 3D printers [162], [163] have developed time-encoding (T-Codes) based machine control layers in direct response to the issues that I discussed in Section 1.3.3, which is that GCode interpreters' velocity planners make it very difficult to synchronize additional axes of control with motion without modifying those planners directly.

These researchers want to control their custom gel extrusion printers using a system of their own design because off-the-shelf GCode interpreters for 3D printers make assumptions about flow dynamics that their systems do not contain. To work around this, they follow a few steps:

1. Write a standard GCode for multi-material 3D printing.
2. Strip that GCode of extrusion information, leaving only positioning commands.

3. Use their machine's acceleration and velocity parameters to calculate an exact time mapping between the original GCodes and the machine's actual operation.
4. Use that time mapping and the original GCode instruction to generate TCodes (time-encoded instructions) that operate their custom extrusion system.

This is an interesting approach and solves a classic *type* of problem that I have described already: of integrating new physics into existing motion systems. In Chapter 5, I show how the motion controller from Chapter 4 is more fundamentally modified to encode and then optimize both flow and motion physics together. Their insight is that machine operation is basically time-based, not position based and so external control should be orchestrated as such. This is present in MAXL's use of fixed time encodings for basis splines, and in [164] I even showed a very similar example in synchronizing LEDs with velocity-controlled motion paths using time encodings of an "event" track type, the output from that experiment is in Figure 2.13.

Other researchers simply write computational APIs that can compose GCodes directly. One popular tool for this practice is called FullControl [165] and Blair Subbaraman extends the p5.js ecosystem in a similar tool [166]. This approach is excellent for more direct operation of hardware, but is mostly suited to experimentation, education and creative practice. However, it does show that while GCodes present certain integration issues, they still expose a relatively straightforward "API" that can be extended in simple and productive ways.

2.3 OSAP: Machine Systems Interconnect

Open Systems Assembly Protocol

OSAP (for Open Systems Assembly Protocol) is a relatively lightweight piece of software that I have authored in C++ for embedded devices and in Python for high-level system components. It includes a runtime where messages are queued and handled (Section 2.3.1) and software interfaces that connect that runtime *up* into software and *down* to network drivers (2.3.1.2).

Because it mediates between hardware and software and does so *across* networks (where the real "underlying hardware" is the network itself), it is akin to a distributed operating system. In contrast to most distributed systems architecture from the state-of-the-art, embedded devices are first-class citizens in this scheme.

Another main differentiator is that OSAP's networking is *link agnostic,* meaning that it can be extended across many types of network technologies with relatively little overhead. Links are added to OSAP's runtime in the same way that software is: via *interface classes* in Python (through composition) or in embedded (via class inheritance); to improve performance of networking hardware software interfaces are often missing between OSI layers, but OSAP includes them to improve systems-level composability. Links are all point-to-point at the moment but the protocol could be extended to include fieldbusses and includes reservations for this.

Because OSAP is a *protocol* and *design spec,* it should be easy to author in other languages when i.e. we want to build versions that provide handles to software written in Rust or JavaScript, or even into hardware design languages for custom silicon or FPGAs. At the moment I run it in Python and in embedded C++, but earlier versions were also written in JavaScript.

OSAP's main task is to get serialized messages from any port in the system to any other port in a timely manner. It does so using *source routing,* a simple networking scheme that I discussed in Section 2.2.2.2 - OSAP's implementation is described in Section 2.3.1.3.

It also provides two valuable services. Section 2.3.2.1 describes the discovery service which allows any device to retrieve a network map of locally connected devices. This allows us to inspect networks and determine i.e. if the motor drivers that our machine needs are, in fact, connected, how to reach them, and how to describe the network route between *two other* remote devices if we would like to remotely configure dataflows. Section 2.3.2.2 describes the *time synchronization* service, which keeps device clocks in step with one another. This is a critical building block for mechatronic systems since it lets us synchronize motion, measure real network performance, and collect coherent time series data from networks of sensors and actuators.

OSAP is the connectivity layer, programs are assembled within OSAP networks using PIPES (Section 2.4), and motion controllers are composed in PIPES using MAXL (Section 2.5).

In my description of OSAP in this section I first cover how it operates under the Runtime header 2.3.1; its structure in 2.3.1.1 and main components: ports and link gateways, and the software interfaces that are used to integrate them to the runtime (in 2.3.1.2), and then the networking scheme in 2.3.1.3: this describes how packets move through the network. Memory allocation, flow control and scheduling are all closely related and are covered in Sections 2.3.1.4, 2.3.1.5 and 2.3.1.6. We can then understand OSAP's main loop in Section 2.3.1.7 and finally discuss the transport layer in 2.3.1.8. Network discovery (Section 2.3.2.1) and time synchronization (2.3.2.2) are both under the Services header, Section 2.3.2.

2.3.1 Runtime

At the core of OSAP is a runtime that manages parallel operation of network links and message passing and software modules that are attached to its ports. In embedded devices where no operating system is present, it resembles an RTOS (realtime operating system) because it allocates CPU time across software modules that each run sub-programs (this is scheduling). In workstation computing it runs within Python's existing `asyncio` loop that itself runs within the existing OS. Below I describe its core data structure, its operating principle across scheduling, flow control and memory allocation, and the software interfaces that are used to connect it "down" into network segments and "up" into software.

2.3.1.1 Structure, Setup, and Loop

OSAP's internal structure has three key elements:

1. A list of *ports,* which are interfaces to software.
2. A list of *link gateways,* which are interfaces to network segment drivers.
3. A stack of *packets* to handle.

These are the software handles used by the runtime to schedule operation, receive and process packets, and collect information about itself during network discovery steps. Each is explained in more detail in subsequent sections. The runtime itself also has a few properties:

1. A *name,* which can be remotely re-written, and is saved into non-volatile memory.
2. A *type name.*
3. A *runtime version number* and *build type.*
4. A *netresponder* class on the 0th port (see 2.3.2.1).

OSAP is deployed as a software library that connects to device-specific drivers for i.e. link layers and other hardware through it's own API. Those interfaces are defined in the next section on ports and links. I show how the runtime object itself is instantiated in embedded devices in Listing 2.5 (in the Arduino style) and in Python scripts in Listing 2.6. The notable difference is that the embedded runtime is currently limited in it's ability to dynamically instantiate link gateways and ports, although I did implement that functionality on microcontrollers in an earlier predecessor to the system in my master's thesis [167].

Listing 2.5: This is boilerplate code to instantiate the OSAP runtime in an embedded device in an Arduino build system. An example of link gateway instantiation is in Listing 2.9. Ports are similar, but are most often instantiated via Pipes Function software objects, an example of which is in Listing 2.34. The runtime's main loop routine is described in Listing 2.13.

```
// in 'main.cpp'
OSAP_Runtime osap( // instantiate the runtime,
    "loadcell_cs5530_d21", // device type
    DEVICE_NAME // device name
);

// ... link instantiations
// ... port / pipes function instantiations

void setup() {
    osap.begin(); // initialize the runtime,
}

void loop() {
    osap.loop(); // call the runtime's loop repeatedly,
}
```

Listing 2.6: A boilerplate scripting utility to establish an OSAP runtime in Python. Here, the loop operates alongside Python's native concurrency framework `asyncio`. Link gateways can be instantiated dynamically or set up at the beginning of the program, which is what I do here: `AutoUSBPorts()` scans for and opens any USB-CDC devices that present with device IDs that are likely to contain OSAP runtimes. It does not matter which `COM` / `tty` number they associate with in the OS because systems configurations will be automatically laid on top of these network addresses later on. This utility also instantiates the `MetaManager,` which builds the PIPES Systems Object Model (Section 2.4.4.1) via network scan.

```
async def network_tools_setup(runtime_type_name, runtime_name):
    osap = OSAP(runtime_type_name, runtime_name) # runtime software object
    loop = asyncio.get_event_loop() # hook to the script's asyncio loop,
    loop.create_task(osap.runtime.run()) # attach osap's loop as a task

    auto_ports = AutoUSBPorts() # discover and start ports
    gateways = await auto_ports.start_ports()
    for gateway in gateways:
        osap.link(gateway) # attach

    manager = MetaManager(osap) # the PIPES SOM generator
    await manager.refresh_system() # perform the first scan

    return osap, manager
```

2.3.1.2 Ports and Links

I think of OSAP's ports as facing *up* into software[52] and link gateways as facing *down* or *out* into network segments and Figure 2.4 and other diagrams in this section reflect that. Both are attached to OSAP's runtime using software defined interfaces. In C++ these are based class inheritance and in Python via composition. I show base classes in C++ for each in Listings 2.7 and 2.8 below; functions marked with `virtual` are implemented by the inheriting class and others are defined within OSAP.

Both interfaces have functional equivalents to network transmit and receive; for ports this means passing data across network *routes* to another *port* in some other runtime or, where the route is empty, to another port in the same runtime. For link gateways this simply means framing and sending data over its connected segment, or ingesting data from the same and adding it to OSAP's internal message stack, which is described in the next subsection.

Almost any link technology can be added to OSAP, but each needs to perform a few key functions. The first is to encapsulate outgoing packets and delineate incoming packets from raw byte streams (this is "packet framing"), and the second is to ensure that packets written into OSAP's message stack (see the next section) do not contain errors: OSAP's internal operation is not robust to corrupted data. This is a common pattern across many network architectures.

These are not complicated tasks, for example in most of the systems that I used in this work, I promote simple UART drivers (which are ubiquitous and simple) into robust network links using Consistent Overhead Byte Stuffing (COBS) [168] to frame packets and also adding Cyclic Redundancy Checking (CRC) [169] for error detection. Specifically I use `CRC16 CCITT` with the `0x1021` polynomial as specified by [170] (a standard). COBS is designed for deterministic performance and CRCs can be calculated quickly using pre-computed tables. Doing both in firmware *does* introduce some computational burden, but the simplicity over using external network controllers or other ASICs for the task is advantageous, in Section 3.4 I expand on why that is the case.

[52] Programmers will normally think of "ports" as the interfaces they use to communicate *into* a network, so when we are talking about OSAP's internals, this can become confused with "links," which are what point out of *OSAP* into network segments.

Listing 2.7: The interface between OSAP's runtime and software: each software module connects to a bi-directional Port.

```
class VPort {
  public:
    // Start / Run
    virtual void begin(void);
    virtual void loop(uint64_t now);

    // Software->OSAP (TX)
    boolean clearToSend(void);
    void send(
      uint8_t* data, size_t len,
      Route* destinationRoute,
      uint16_t destinationPort
    );

    // OSAP->Software (RX)
    boolean clearToReceive(void);
    virtual void onPacket(
      uint8_t* data, size_t len,
      Route* sourceRoute,
      uint16_t sourcePort
    ) = 0;

    // Constructor
    VPort(
      OSAP_Runtime* _runtime,
      const char* _typeName,
      const char* _name
    );

    // Properties
    char typeName[TYPENAMES_MAX_CHAR];
    char name[PROPERNAMES_MAX_CHAR];

    // States
    uint8_t currentPacketHold = 0;
    uint8_t maxPacketHold = 2;

    // Stash
    static uint8_t _payload[PCKT_MAX];

  private:
    OSAP_Runtime* runtime;
    uint16_t index;
};
```

Listing 2.8: The software interface between OSAP and a Link Layer, a "Link Gateway." Each connects to an external network segment (the link) using a software driver that extends this interface.

```
class LinkGateway {
  public:
    // Start / Run / State
    virtual void begin(void);
    virtual void loop(void);
    virtual boolean isOpen(void);

    // OSAP->Link (TX)
    virtual boolean clearToSend(void);
    virtual void send(
      uint8_t* data,
      size_t len
    ) = 0;

    // Link->OSAP (RX)
    void clearToReceive(void);
    void ingestPacket(VPacket* pck);

    // Constructor
    LinkGateway(
      OSAP_Runtime* _runtime,
      const char* _typeName,
      const char* _name
    );

    // Properties
    char typeName[TYPENAMES_MAX_CHAR];
    char name[PROPERNAMES_MAX_CHAR];

    // ------------ States
    uint8_t currentPacketHold = 0;
    uint8_t maxPacketHold = 4;

    private:
      OSAP_Runtime* runtime;
      uint16_t index;
};
```

Listing 2.9: Links are added to embedded runtimes by instantiating link gateways, each of which wraps a network segment (the underlying link). Each of these examples is a class that is built on the base gateway class in Listing 2.8. In this example I use firmware configurations i.e. `USE_UART_LINK` to define which gateways the runtime will load.

```
// instantiate the USB-CDC gateway,
#ifdef USE_USB_CDC_LINK
OSAP_Gateway_USBCDC_COBS<decltype(PA_USB)> serLink(
    &PA_USB, // link driver
    "usb" // link type
);
#endif

// or instantiate the UART gateway,
#ifdef USE_UART_LINK
OSAP_Gateway_UART_COBS_CRC16<decltype(PA_SERIAL), true> uartPA(
    &PA_SERIAL, // link driver
    UART_BACKPACK_BAUD, // data rate
    "uart_rs485_pa", // link type
    PA_PIN_GP1, // link hardware configuration
    PA_PIN_GP0,
    PA_PIN_CS
);
#endif
```

In many cases OSAP uses this link interface to virtualize existing network segments. For example we can establish OS-level software sockets and encapsulate those as link gateways in a runtime, the same is true of USB-CDC connections. In these cases and others, the underlying links already perform both or some of these tasks and so we do not need to re-define them in the link gateway's driver. Sockets are already packetized and error corrected, whereas USB-CDC streams are unencapsulated but error corrected; in the underlying *USB* drivers they are also encapsulated, but these lower layer drivers are more difficult to write whereas USB-CDC drivers work "out of the box" on most operating systems. So you can see how the OSI model becomes somewhat strange here: we are taking IP-based sockets (a full network stack) and exposing them to OSAP at the *link layer*.

This is a design choice based on the *availability* of these technologies to a systems integrator. OSAP is a kind of melting pot and these interfaces homogenize messy systems to make its internal system smoother; at the runtime layer each link gateway or port is like any other. This means that configurations for software-defined links — i.e. the `COM` number for wrapped USB-CDC connections or addresses for IP-based sockets — are external to OSAP. This is a design choice that increases simplicity of the networking scheme, but it limits our ability to configure networks remotely. There is future work planned to add these capabilities in a manner that doesn't add complexity to the runtime itself (in Section 2.9.1.1), or to instantiate embedded link gateways based on the Knuckles Backpack ID Pin to update software configurations based on hardware state — see Section 3.3.

2.3.1.3 Networking and Packet Structure

OSAP is based on source routing. I covered a wider background on the strategy in Section 2.2.2.2; the basic scheme is that data sources write network routes directly into the packet header rather than specifying a destination address that must (1) be globally unique in the network and (2) be resolvable by intermediate network devices. Source routing is much simpler because intermediate devices simply read routing instructions from the packet (no thought required), but requires an outer loop of route configuration. In this work, that is done globally by way of OSAP's discovery service (explained in Section 2.3.2.1) which is used at a programming level in PIPES to overlay software configurations on networks, see Section 2.4.4.3 on connection configuration and Section 2.4.4.1 for the Systems Object Model that represents those configurations (and underyling networks).

Figures 2.4 and 2.5 diagram two OSAP systems and examples of some source routes between ports. Operation of these routes is described below.

Packets headers contain small bytecodes that are interpreted by the runtime in order to forward messages, process them internally to provide network services like time synchronization, or send them on into ports.

Because link gateways and ports in a runtime are stored in ordered lists, each has an index — these are used in packet arguments. For example one of these instructions is `link_forward,` it has one argument for the link index that the message should be sent along.

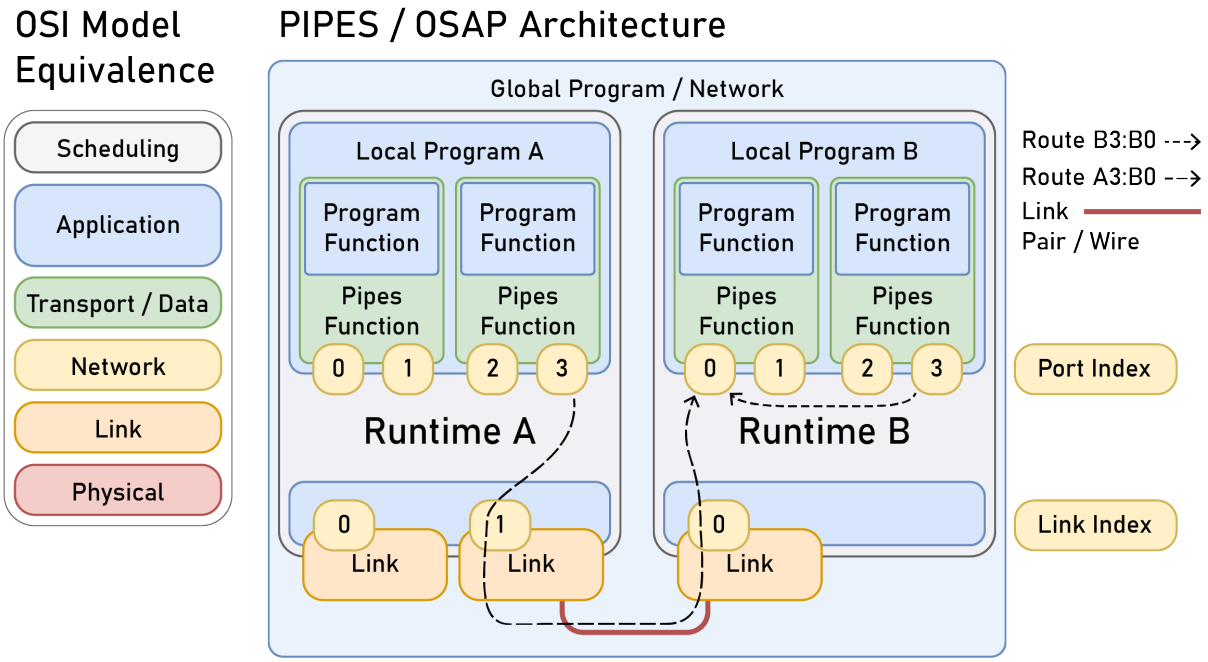


Figure 2.4: Two OSAP Runtimes `A` and `B` each containing a local program and program functions that are wrapped as Pipes Functions. Link gateways and their indices are "below" the runtime, with functions attached via ports "above." Each of these have indices, which are used in packet routing. Two routes are shown and routing instructions for each are shown in the text below.

Listing 2.10: Packet instruction pointers are read from the packet's first byte, and operation types are in the first two bits of the instruction byte with arguments packed behind.

```
// the instruction pointer is in the first byte of the packet,
uint8_t ptr = pck->data[0];
// the opcode occupies only the first two bits of any byte in the header,
uint8_t opcode = pck->data[ptr] >> 6;
```

Route	1st Bytecode	2nd Bytecode
B3:B0	`port_handle(src: 3, dst: 0)`	
A3:B0	`link_forward(1)`	`port_handle(src: 3, dst: 0)`
B0:A3′	`link_forward(0)`	`port_handle(src: 0, dst: 3)`

Table 2.1: Bytecodes for the routes in Figure 2.4, including the reversed B0:A3′ route.

The first byte in the packet is the instruction pointer, it indicates the in-packet location of the next instruction. The instruction type is indicated by the first two bits of this byte, so to get the opcode we simply do:

Some other source routing schemes require that we search for the instruction pointer in the packet, which both increases the number of compute cycles required to find the opcode and requires that packet instructions are delineated from the pointer itself which consumes valuable bits in the header. Others *delete* processed codes as the packet traverses the network, in these cases the route is either completely removed from the packet by the time it arrives at the destination (and so it cannot be reversed) or it is inserted at the end of the packet, which requires shifting the packet bytes in-memory. The pointer lets us simply modify instructions in-place to reverse them, which we do.

Link gateways *transmit* packets without modifying the packet, meaning that the recipient gateway can recover the index of its partner in *its* runtime, which represents an important piece of network state for later graph traversal routines. When they ingest packets, they increment the pointer value so that the runtime will process the next instruction and insert their *own* index in place of the forwarding instruction, which then becomes one step along the reversed route. Leaving this information in the packet means that we can return messages to their sender at any time, i.e. ports can reply to senders without having to ascertain network state. Tables 2.1 and 2.2 include (in the third row) reversals of routes from the second row.

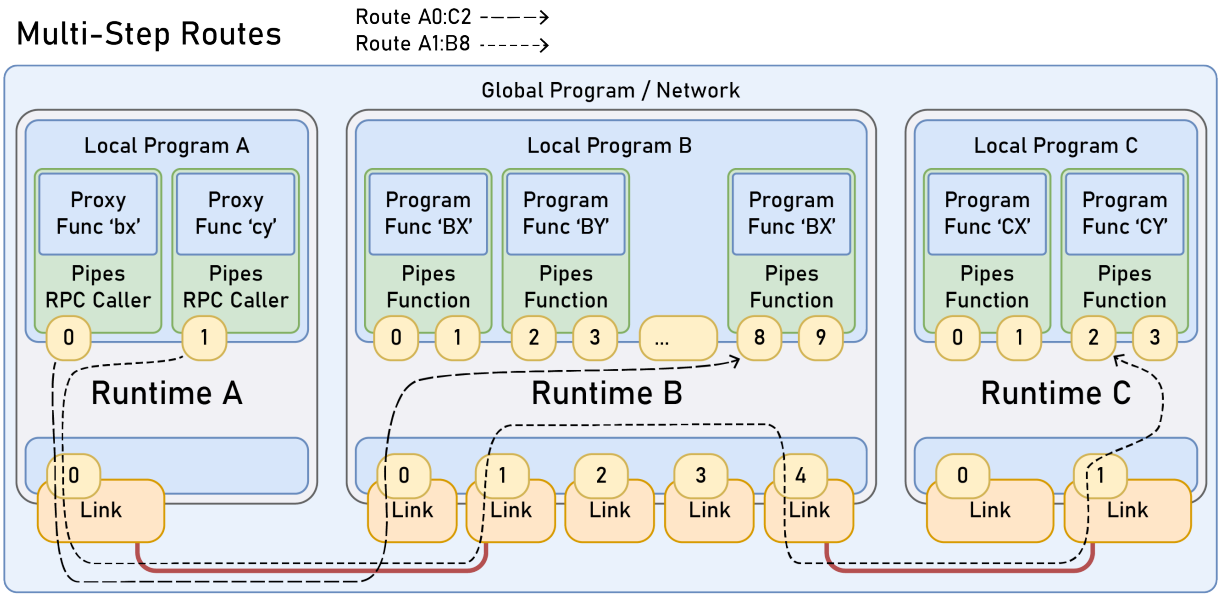


Figure 2.5: Here I show three runtimes with two routes, one of which uses an intermediate runtime being used purely as a message passing device. A unique property of OSAP is that runtimes and "routers" are not distinct devices.

Route	1st Bytecode	2nd Bytecode	3rd Bytecode
A1:B8	`link_forward(0)`	`port_handle(src: 0, dst: 8)`	
A0:C2	`link_forward(1)`	`link_forward(4)`	`port_handle(src: 1, dst: 2)`
C2:A0′	`link_forward(1)`	`link_forward(1)`	`port_handle(src: 2, dst: 1)`

Table 2.2: Bytecodes for the routes in Figure 2.5, including the reversed C2:A0 route.

Routes can be extended across potentially many links by adding instructions, Figure 2.5 shows one example of this (it corresponds to Table 2.2). `link_forward` instructions are only one byte (encoding up to 64 ports per runtime, the first two bits contain the opcode and the last six are used for the opcode itself), the provision for `bus_forward` is two bytes (for 64 fieldbusses per runtime with 255 drops on each bus).

The `port_handle` opcode instructs the runtime that the packet should be delivered to a software module. It has two arguments: destination and source port indices. These are used by the runtime to deliver the packet to the appropriate port, that port also receives the source port and source route (which is reversed), both of which are required to generate a reply. This instruction is three bytes in length, source and destination indices are 11 bits each for a limit of 2048 unique ports per runtime.[53]

The final opcode is a `system_message,` at the moment those are used only for time synchronization

[53]This seemed like a lot when the project started, but it should likely be expanded with an additional byte for 15 bit port addresses (32 768 ports).

Code Key	Bytes	Arg	Arg
`link_forward`	1	Link Index `[2:8]` (64 Gateways)	
`bus_forward`	2	Bus Index `[2:8]` (64 Fieldbusses)	Bus Drop `[8:16]` (255 Drops, 1 Broadcast)
`port_handle`	3	Source Port `[2:13]` (2048 Ports)	Destination Port `[13:24]` (2048)
`system_msg`	1	Message Code `[2:7]` (64 Keys)	

Table 2.3: Opcodes and arguments at the runtime level in OSAP. Codes have variable length, code keys take up the first two bits `[0:1]` of any given code, and arguments are inserted piecewise into the succeeding bits, `[inclusive, exclusive]` and zero-indexed.

(see Section 2.3.2.2), but up to 64 of these codes could be implemented for i.e. trace packets.[54] There is allocation for a `bus_forward` opcode as well, with an argument for the bus gateway index and an additional argument for the *network address* on the bus of the receiving runtime. This is not implemented in the current version of OSAP but Section 2.9.1.1 discusses their inclusion, as fieldbusses are very common in industrial systems.

Packet headers contain two additional values, both of which are 16-bit. The first is the route's Maximum Transmission Unit (MTU) which defines the underyling link layers' largest data packet size in bytes. Where routes hop across multiple links, the smallest MTU of all links in the route is used. This is an important consideration because MTUs can vary widely in different link layers, for example CAN bus is often limited to 64 bytes only whereas Ethernet frames can be as large as 1500 bytes, some IP links allow up to a full 65 536 bytes per frame, and inter-process sockets in unix operating systems have variable underlying MTUs that are mostly based on buffer sizes. Encoding the MTU in *every* packet does consume considerable packet header space, but it ensures that low-level operation is stateless and the trade is that even very large MTUs can then be used whenever they are available, which improves efficiency.

The other header value is a *time to live* (TTL) stamp in microseconds, which is the basis for OSAP's earliest deadline first scheduling - more discussion on that to come. A complete packet header that specifies one routing step (a link forward) between two ports in two different runtimes is shown in Table 2.4 below.

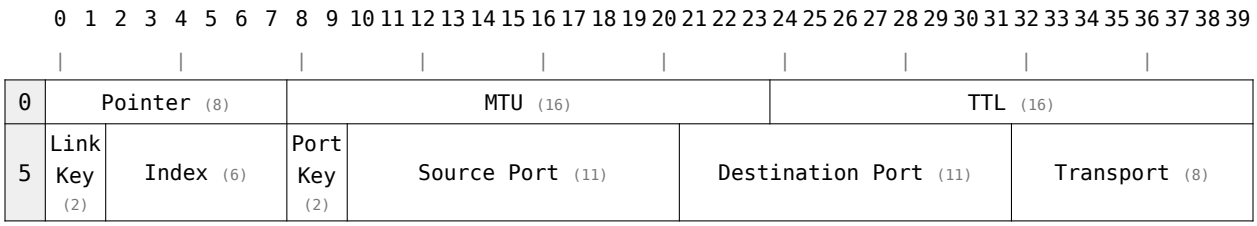


Table 2.4: An OSAP packet header for a message that is routed between two ports in two runtimes across one link forwarding step.

[54] Trace packets can be used in some networks to measure performance. As they move through networks, they append timestamps into their datagrams and information about the location of the timestamp — see Section 2.9.1.3.

Listing 2.11: The "Virtual Packet" structure used in OSAP: runtimes keep a stack of these in a linked list, which allows them to be insertion-sorted by their service deadlines under "earliest deadline first" scheduling. This struct also stores the time of the packet's arrival in the runtime, and a pointer to the system component that owns it: packets can be passed internally via pointer ownership without relocating them in memory.

```
typedef struct VPacket {
  // data and it's size,
  uint8_t data[PCKT_MAX];
  size_t len = 0;

  // the scheduling deadline and time of arrival,
  uint32_t serviceDeadline = 0;
  uint32_t timeOfArrival = 0;

  // pointers to packet owners,
  // only one of which is valid at any given time
  VPort* vport = nullptr;
  LinkGateway* linkGateway = nullptr;
  OSAP_Runtime* runtime = nullptr;

  // for linked list operation,
  VPacket* next = nullptr;
  VPacket* previous = nullptr;
} VPacket;
```

2.3.1.4 Memory Allocation and the Message Stack

The central data structure in the runtime is the "Virtual Packet," see Listing 2.11 for the definition of the `VPacket` structure in the embedded runtime. These store raw packet data, the packet's arrival time and service deadline, and pointers to the runtime component that owns them.

The stack is at the very core of OSAP's runtime: service deadlines are used to schedule overall operation and packet allocation and deallocation is a key component of the flowcontrol system. Packets move between components in the runtime via pointer ownership, so they are also a kind of memory bus.

In workstation scale computing the depth of this stack is virtually unlimited relative to the average packet size but on embedded devices allocating memory manually requires extreme care.[55] Microcontroller memories are also limited and so robust embedded designs prefer static allocation. I use static allocation of packets in both cases (Python and embedded) to match operation throughout but in embedded the scheme is slightly more involved, as I will explain.

Ports and link gateways may try to generate new packets at any time and write them into this stack, but memory and CPU time is limited. To allocate access to the stack fairly, each port and link can only own a finite number of stack slots at any given time. This limit is configurable on

[55]Memory allocation is one of the main functions that operating systems perform and doing it poorly can lead to severe and difficult to trace bugs.

Listing 2.12: Embedded link gateways and ports use these functions to check for, claim and relinquish space on the shared packet stack.

```
// in `packets.h`
boolean claimPacketCheck(VPort* port); // check if a stack slot is available,
VPacket* claimPacketFromStack(VPort* port); // claim a packet,
void relinquishPacketToStack(VPacket* pck); // release the packet
```

a per-port and per-link level, i.e. you can see in the link gateway (Listing 2.8) and port (2.7) a `maxPacketHold` value. When implementing simple link layers that do not contain their own internal buffers - or very small ones - the stack can effectively extend these buffers.

In Python it is easy to expand the stack size such that all ports or link gateways can be "full" at any given time, but in embedded devices we sometimes need to limit stack use globally as well as on a per-component basis. There, each component uses the functions in Listing 2.12 to check for, claim, and relinquish stack space.

2.3.1.5 Flow Control

Flow control is a common problem in networking. We have the fundamental problem that data sources will often want to produce data at a rate that data sinks cannot manage. This takes form in a few different ways, the first of which was just mentioned: memory is not unlimited and so packets cannot be written by ports or pushed from links into the runtime when the stack is completely allocated. On the other side, ports need time to handle data and links are not always "clear to send" new data: their transmit buffers may still be full with prior data or their downstream partners may be equally busy. This extends end-to-end in the case where very fast devices transmit very many packets that must be consumed and processed by very small devices.

OSAP manages flow control on these two ends (sources and sinks): the stack interface controls flow into the stack as I described above, and messages will not be cleared from the stack before links or ports that will handle them are busy. You can see the software interfaces for this in Listing 2.8 and 2.7 (`clearToSend, clearToReceive`). The runtime's main loop routine in Listing 2.13 also reflects this operation. Application layer flowcontrol is additionally performed at the transport layer, as is done in IP networks.

2.3.1.6 Scheduling and QoS via Earliest Deadline

Flow control and scheduling are related because both are related to how we operate networks under resource constraints: limited bandwidth and limited CPU time. I covered a broader background on networking constraints that includes some discussion of scheduling in Section 2.2.2.1.

As I've mentioned, operation in OSAP is scheduled around packets' service deadlines: packets that are due to expire soonest are handled first, using the "earliest deadline" method that I have already mentioned which is based on [87] who note that this method out performs priority-based scheduling under heterogeneous loading scenarios.

Deadlines are calculated based on their arrival time within the runtime (which is also stored) and their packet-encoded time-to-live. Rather than sorting the packets according to this value in

each runtime loop, packets can be inserted into the stack (a linked list) *based on their deadline* so that in the next loop cycle new packets are already sorted into the order in which they should be processed.

Time-to-live is also the main software handle for configuring network Quality of Service (QoS) and is first written into packets at their source ports: for example Pipes Functions (2.4.2) expose this value for individual connections. Tighter deadlines will cause packets to be serviced earlier than others, so they are a proxy for network priority and also encode - well - deadlines: if we have a control loop that must happen within a certain interval, we can transmit its data with a time-to-live that corresponds.[56]

When a link gateway transmits a packet it updates the time-to-live encoded in the packet header (see Section 2.3.1.3 for a table of packet encodings) according to the transmitting runtime's measurement of that deadline, i.e. removing the interval of time that has elapsed[57] since the packet was written into that runtime. This way the time-to-live value decreases as the packet traverses the network and overall timing can be maintained. At the moment this *does not* account for the wire time (the interval that the packet spends actually being transmitted) which can be substantial. Future work could use coordination between the synchronization service and this layer to ameliorate this. This type of interaction is one of the many ways that network layers - while we may try to separate them - remain intimately related to one another.

2.3.1.7 OSAP's Main Loop

Considering each of these steps, we can now see how the runtime's main loop runs - the core routine contains these four steps:

This tight loop is run repeatedly and so blocking code at any of these steps can cause e.g. a number of packet deadlines to expire at once. This amounts to *cooperative* scheduling, which is a common strategy for simpler runtimes like OSAP's. In Section 2.8.3, I discuss how OSAP is likely to require

Listing 2.13: The main OSAP loop.

1. Run `loop` functions defined by each port and link gateway.
2. Traverse the message stack, which is sorted against packet deadlines:
    a. De-allocate packets whose deadline has passed.
    b. For packets addressed to Ports, reverse the route (2.3.1.3) and the source port index, and call the receiving Port's `onPacket` function if it is `clearToReceive,` de-allocate the packet if so, or continue (leaving it on the stack for the next cycle).
    c. For packets that should be forwarded through the network, read the next routing instruction, and forward it via the link's `send` function if it is `clearToSend,` deallocate the packet if so, or continue.
    d. Handle `system_msg` packets.
3. Run the time synchronization routine from Section 2.3.2.2; send clock queries to neighboring devices if the control interval has elapsed.

[56] This may too aggressively over simplify the scheduling problem, see Section 2.9.1 for a note on improvements.

[57] For simplicity and robustness this is based on the runtime's own accounting of time, not the system synchronized time.

preemptive scheduling (where ongoing tasks can be interrupted by the main loop) if it is to become a truly robust system. The cooperative strategy is exceptionally simple and does not lead to major issues if code is designed appropriately.

2.3.1.8 Transport

Software can be implemented directly on top of ports, which would be equivalent to the use of UDP packets in IP-based networks. However, we can also insert software-defined transport protocols. These are implemented between software (which may want e.g. delivery guarantees, timing guarantees, sequence guarantees, or to frame large messages over a series of packets) and ports (which are stateless). The first byte in any port-to-port message's data frame is a `transport_key,` which ID's the type of transport layer that the message is sent by. This is a provision to deconflict transport layers, so that messages that are improperly addressed can be handled appropriately.

So far I have only authored two of these: a very simple Sequential Transmitter and Receiver that provide order-of-delivery guarantees and allow us to easily *reply* to a particular message (not just a particular port) which helps software to ensure that responses to particular RPC (Remote Procedure Calls) (Section 2.4.4.2) are resolved against the correct call.

Transport layers in network systems are some of the most complex, and can introduce large overhead. The internet has annealed around TCP (Transport Control Protocol) but it is rarely used in networked control because of this overhead and state: messages can end up trapped in queues for long periods, etc. Transport control protocols are in fact controllers, i.e. the work that I discussed earlier in the background section on network design itself as an optimization problem made reference to Chiang's work [104], where transport control itself is considered as a distributed control problem across many layers.

In this work, streams of data basically fall into two categories: configuration steps which can be slow but can fail if delivery guarantees are not provided, and operation where missing a packet is permissible and transport control is not strictly necessary. This is typical of networked controllers, and is another reason that we want to have flexibility in this layer as well.

2.3.2 OSAP Services

2.3.2.1 Network Discovery

I mentioned in the background section on source routing 2.2.2.2 that the main drawback to the strategy is that it becomes difficult to configure routes themselves, because this must be done manually. This is OK in static systems like SpaceWire [171] because i.e. the James Webb Space Telescope's physical network configuration does not change very often and so routes can simply be written offline against system schematics and loaded into firmwares. For a reconfigurable, modular system it is obviously an issue.

So! OSAP also includes a kind of "distributed DNS" layer to allow discovery of network topologies. This effectively follows the Model-View Controller (MVC) architecture that I discussed in Section 2.2.3.5.

Listing 2.14: Pre-defined message types that can be sent to an OSAP Runtime's Model-View Controller

1. Get Runtime Information
2. Get Runtime Type Name
3. Get Runtime Name
4. Set Runtime Name
5. Get Link Information
6. Get Port Information
7. Get Debug Messages
8. Set Debug Verbosity
9. Get Time Controller Configuration
10. Set Time Controller Configuration

OSAP Runtimes each implement a *"netresponder"* class on their 0th port, which corresponds to the controller in the MVC framing. These reply to queries made by *"netrunner"* instances in other devices, which are responsible for ascertaining global network state - the *view.* The runtime's internal structure is the *model,* in this case the total *view* spans multiple runtimes and involves also using information in source routes themselves to ascertain the structure of the network that connects them.

Each query and response uses data serializations that are pre-determined and written at a low level. While it would be possible to implement these using Pipes Functions (since they are basically RPCs), that would collapse the network and application layers and so goes against the design ethos of the stack. These are the queries:

To collect network state, the netrunner code does a breadth-first search over runtimes. It begins by querying its own runtime to ascertain how many link gateways it contains and whether or not they are open, and then writes a packet to traverse each of those gateways and query the runtimes connected to each, etc. Source routes are useful again here because they can be built sequentially in each recursion of the traverse and because they themselves represent network topologies.

OSAP allows that there can be multiple paths to the same devices, which may be beneficial in control systems where we would want to send data on one link and receive it on another, or otherwise use i.e. multiple routes between two devices to increase overall throughput and reduce latency, perhaps sending traffic that requires determinism over one link and burst traffic over another. This means that networks can contain loops and a simple recursive breadth-first search would go on endlessly if they were not de-conflicted. To do so, the *Get Runtime Information* message cycles an incrementing UUID (unique to the scan) that is transmitted by the netrunner and stored by the netresponder, which replies with the UUID from the *last* time it received one of these messages. If the netrunner sees a UUID that it previously generated in another lookup, it knows that the device is connected in a loop - and it knows which device this is, to map that loop to the network map.

Operationally, addresses in OSAP are all based on source routes (which are sometimes called "network addresses"). Those are relative, not global; in Figure 2.5 the network address of runtime `C`

from the viewpoint of runtime `A` is different from the network address of runtime `C` from the runtime `B`. This in itself is not an issue when we are configuring a system from one viewpoint, but OSAP implements an additional human-readable naming scheme specifically because it is invaluable at the next layer up, when we start to write programs on these networks - which is what networks are normally made for in any case. The internet's DNS is the same: to visit a website one does not type in the IP address of its web server.

I mentioned in Section 2.3.1.1 that runtimes themselves each have *name, type name, version,* and *build* fields. The names are just strings, the version field is three bytes (so version numbers can span `0.0.0 - 255.255.255`), and the build field is keyed - it encodes whether the runtime is an embedded / C++ system or a Python runtime (etc).

Ports and link gateways each have *name* and *type name* fields as well. Their names can be semantically meaningful but are most often left blank, because the type name is what tells software (like PIPES) that i.e. this port contains a Pipes Function, which will contain its own semantically meaningful data when queried.

I originally designed OSAP itself to be completely *nameless* in the spirit of layer separation in the OSI model, but leaving "hooks" into the next layer at the edges of other layers is invaluable. The discovery of complete distributed programs would be infeasible without these names. In Section 2.8.4 I expand on this theme and reflect on some larger discourse around protocol design between the OSI model and the TCP/IP stack.

2.3.2.2 Distributed Time Synchronization

OSAP provides a clock synchronization service, which is used as a basis for motion control and for time series data collection. Since other clock sync algorithms are complex and consume large amounts of program memory, I developed a simple version that is loosely based on well understood principles from the well established Network Time Protocol (NTP, [88]) and Precision Time Protocol [69], but also on [89], where *diffusion* is used as a basis for synchronization.

System time is measured in microseconds using `uint64_t` to eliminate overflows. Runtimes request timestamps from neighboring devices at some configured interval (I use $50ms$ in most networks), inserting their own timestamp at the head of the packet when they do so. When the neighbor receives that packet, they add a timestamp of their own system clock estimate and their clock tier and then return the packet. Using these data, the runtime calculates its offset to the neighboring clock and runs a simple PI[58] controller to adjust its own clock skew, shimming it up to catch up to neighbors or down if it is overrunning. This is described in Listing 2.15. Parameters for the clock's PI controller and the offset estimation filter are tune-able via the netrunner / netresponder interface.

In the test above, we can see that clocks adjust to within 500μs with one another after $60s$ and within 250μs after $100s$. Devices running faster microcontrollers tend to have better sync performance. Two caveats, the first being the rather obvious disturbances on these plots, where it seems

[58]PI is proportional-integral control: to generate a control output, one control gain (for P) is multiplied by the error, and another (the I) is multiplied by an integrated error signal.

Listing 2.15: The clock synchronization routine that runs in each OSAP runtime.

1. On a configured interval, transmit timestamp query packets to the runtime's nearest neighbors over any open link gateway. Insert the runtime's own time estimate into the packet's first eight bytes.
    a. If a timestamp query is received, insert the local timestamp in the succeeding eight bytes and the local clock tier in the final byte, reverse the route and send the packet to its transmitter.
2. When a reply is received:
    a. Calculate the query's round trip time (RTT) using the timestamp made at packet generation and the current time and then estimate the (N) neighbor's *current* system time using $t_{\text{N,now}} = t_{\text{N,stamp}} + \frac{1}{2}$ RTT.
    b. Measure the offset between the local system time and the neighbor's system time.
    c. Apply an exponential weighted average to the offset estimate and store that for each neighbor, along with its clock tier.
3. Select the best neighboring clock:
    a. Pick the clock with the highest tier.
    b. Update our tier to be that clock's tier plus one.
    c. If multiple neighbors have the same tier, compute the average of our offset estimations to each.
4. Using the computed offset estimate to the best neighboring clock as an error signal, apply PI control to the local clock.

that the controllers are being *kicked* out of band momentarily - I discuss this in Section 2.9.1.3. The second is that $60s$ is a long time to wait for acceptable clock synchronization. To ameliorate

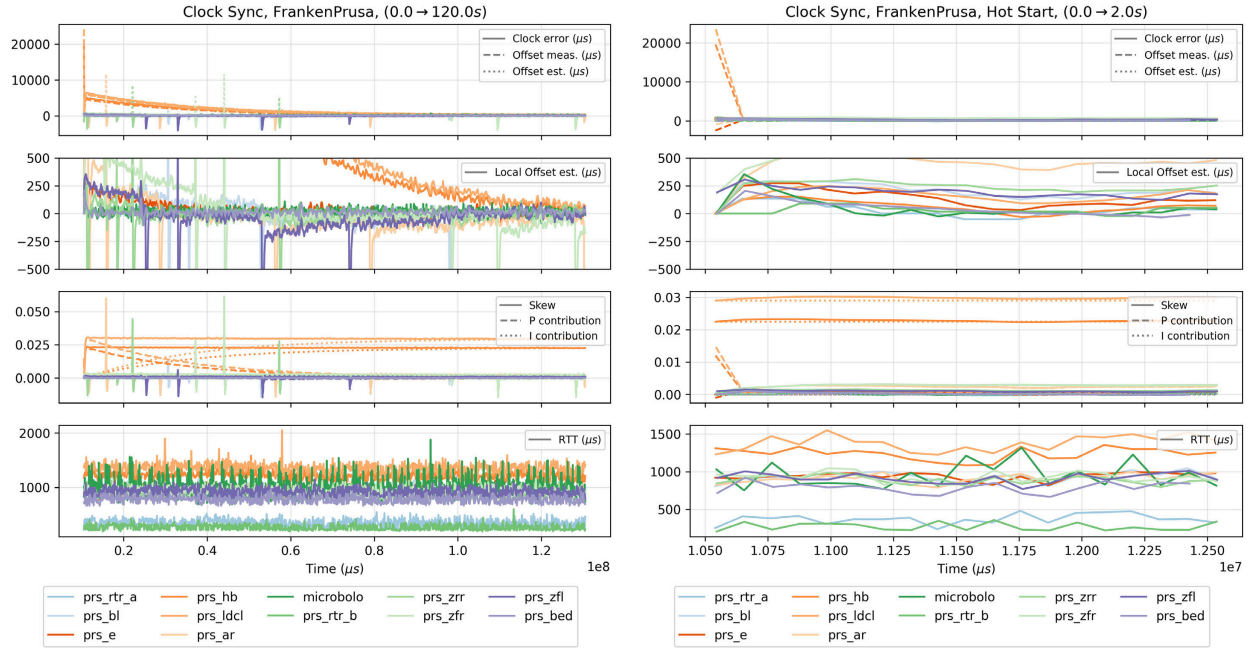


Figure 2.6: These plots show data from two clock synchronization tests on the FrankenPrusa (5.4.2). At left, 120 seconds of sync without hot-starting the clock skews. At right, the first two seconds of sync with hot-start. The first rows plots clock errors in μs, second rows are a y-axis zoom on the *devices'* own estimates of their clock errors, and the third plots skew control terms. The bottom row plots round trip times from the test script (in Python) out to devices.

this, we can *hot start* the synchronization controller, by storing clock skews after longer time synchronization runs and re-loading them into firmware when the system is initialized.

Time synchronization data also provides a quick proxy for network latency, in Section 2.7.9 I use some of these data for a simple evaluation of OSAP's own performance.

Overall, the synchronization in OSAP works well enough for me to complete all the tasks in this thesis. It is clearly not perfect, and high performance synchronization is a requirement of advanced control systems. I discuss future work on the topic in Section 2.9.1.3 and address those errors below.

2.3.2.2.1 Time Synchronization Errors

Before carrying on here, I would like to discuss the periodic errors in the clock synchronization test, shown in Figure 2.6 and also seen in the test data in Figure 2.7 below.

These spikes are not perceptible in any data sets that I saw prior to running these tests, and don't cause any audible or visual disturbances while machines are running, but they are clearly problematic. We can see that they only appear on the motor drivers. This is more clear in Figure 2.7, where the Knuckles hub (3.5) device (labelled `rig_rtr` here), is in blue and the motors are in red / orange hues.

The motor controllers use many control interrupts, which can get preempt the clock synchronization routine. It seems likely that that is causing this issue, via a race condition or by causing asymmetries in the round-trip time measurements (where e.g. one leg of the transmission takes longer than the other, because the μc is busy processing non-network tasks).

Luckily the clock controller recovers from these disturbances quickly, but this reveals is a clear example of the type of issue that emerges due to OSAP's cooperative scheduling, and of the scheduling problem more generally. To work well the timing system may need to preempt the controller, but who is to say which takes precedence?

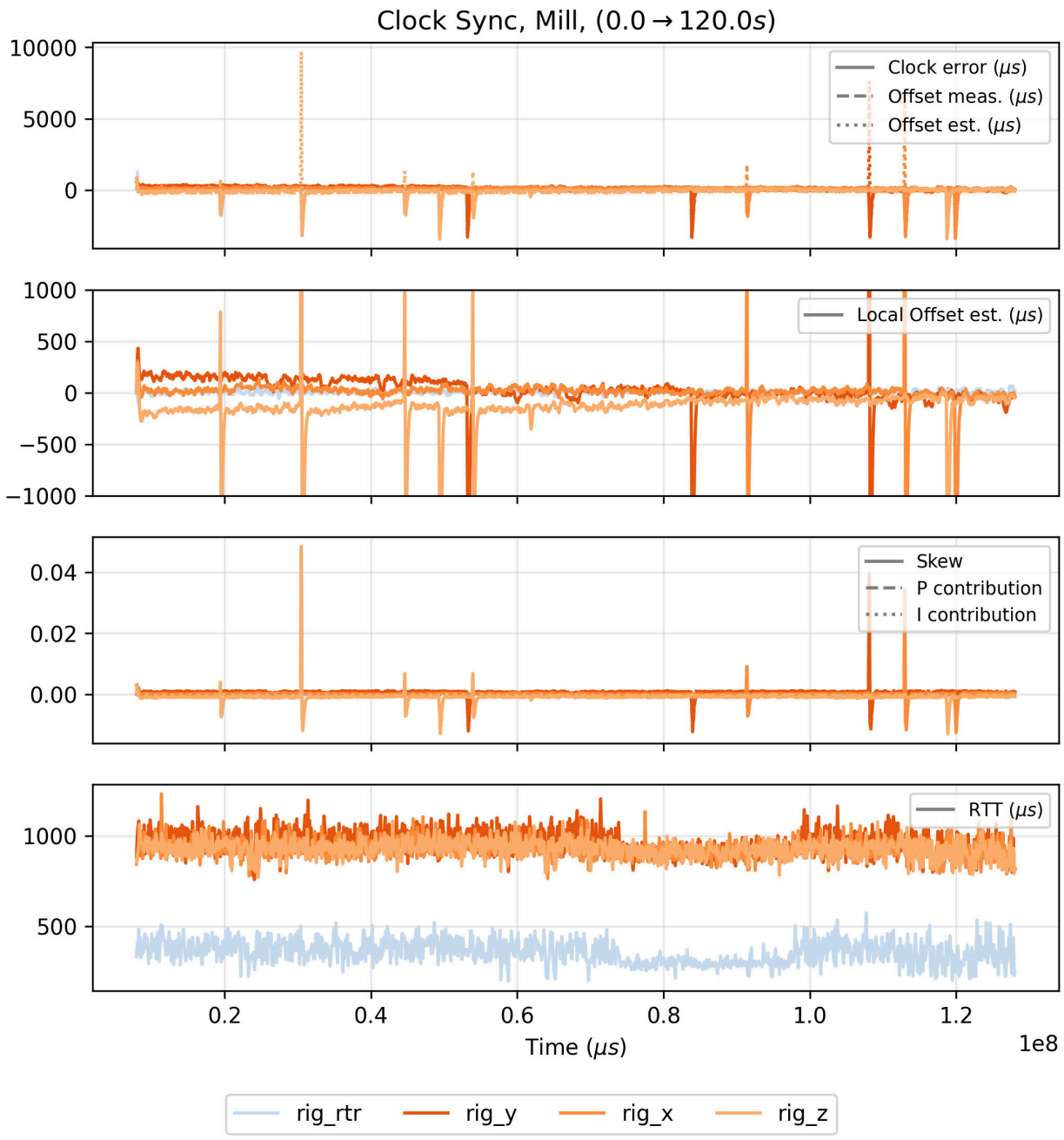


Figure 2.7: The same plot as in Figure 2.6, with data from the milling machine.

2.4 PIPES: a Programming Model for Partitioned Systems

Programming In Piped EcoSystems

PIPES runs on top of OSAP and is responsible for naming and describing software objects, building and modifying systems representations, configuring data flows between objects, and remotely calling software objects. PIPES uses a dataflow programming model to configure and represent controllers, but also mixes in scripting tools that make the *tasking* problem easier. PIPES' configuration tools are presented as a software API, so that scripts which run machines can sequentially configure them programmatically, and then run those configurations — i.e. get them *up and running.*

In this section I first discuss the core design principle in Section 2.4.1, which is to source data schemas from devices and then apply programs on top of those schemas. This is in contrast to most other distributed systems programming tools where systems-wide schemas are first developed and then components are each written against those schemas. I then cover how *functions* (their source code directly) are used as a source of those schemas, and are wrapped in a network interface that embeds them as dataflow blocks and as Remote Procedure Call (RPC) implementations (Section 2.4.2). This wrapper includes a small Model-View Controller (MVC) that allows us to extend OSAP's network discovery service from Section 2.3.2.1 to develop a programming model that combines dataflow with scripting, in Section 2.4.4. Programmer's interactions here are mediated by a `Manager` software object, which produces *proxies* (2.4.4.2) and allows remote configuration of dataflow connections (2.4.4.3), all of which are represented by the underlying Systems Object Model (the SOM, 2.4.4.1). For all of these steps I developed a standalone serialization and typing scheme that is described in Section 2.4.3.

2.4.1 Runtimes are Schema Sources, not Sinks

In Section 2.2.3 I surveyed a selection of state-of-the-art distributed programming systems. Each of those uses an Interface Description Language (IDL) as a shared representation of how system components should interact with the system; IDLs are written by hand and they are the surface against which systems integrators and component developers work.

For example to integrate a new module or program with one of these systems, a programmer writes their device's internal program, imports the IDL, and then authors "glue code" that connects their program to the IDL, defining new functions to call when certain data are received, and which functions will deliver data to certain topics (which are labelled), at which intervals and under which QoS parameters.

PIPES takes a different approach, which is to allow each device to describe its own interface and then expose those in a way that is discoverable so that global maps of devices and their interfaces

Listing 2.16: PIPES allows device programmers to define which functions in an OSAP runtime will be exposed as network interfaces that can be interacted with at a systems level as Remote Procedure Calls (RPCs) or as dataflow elements. These two functions from the Knuckles Loadcell module (from Section 3.2 and used in the 3D printer Section 5.4) are authored in the device firmware, rather than adding a separate layer of glue code to expose them against an IDL, they are wrapped with the `PIPES_FUNC` macro to generate interfaces.

```
// in knuckles_loadcell `main.cpp`
float _lower_thresh = 0.0F;
float _upper_thresh = 0.0F;

// normally used as an RPC,
void setupComparator(float lower_threshold, float upper_threshold){
  _lower_thresh = lower_threshold;
  _upper_thresh = upper_threshold;
}

// normally configured to transmit at an interval through a Pipe
// returns a timestamp *and* the comparator's output value for
// resynchronization with other motion control states upstream
auto getComparatorReading(void){
  auto [time, reading] = getCalibratedReading();
  if(_lower_thresh < reading && reading < _upper_thresh){
    return std::make_tuple(time, true);
  } else {
    return std::make_tuple(time, false);
  }
}

PIPES_FUNC( // the Pipes Function wrapper generator, a C++ template
  setupComparator, // a reference to the implementation,
  "lower_threshold, upper_threshold", // provided argument names,
  "" // names for return values (empty)
);

PIPES_FUNC(
  getComparatorReading,
  "", // no input arguments
  "time, in_bounds" // output value names,
);
```

can be discovered. In Section 2.4.4.1 I describe how that process works; we build a Systems Object Model which effectively replaces the IDL on a per-system basis. For a more in-depth comparison see Section 2.6.1.1 and Figure 2.25 in Section 2.6.1.2.

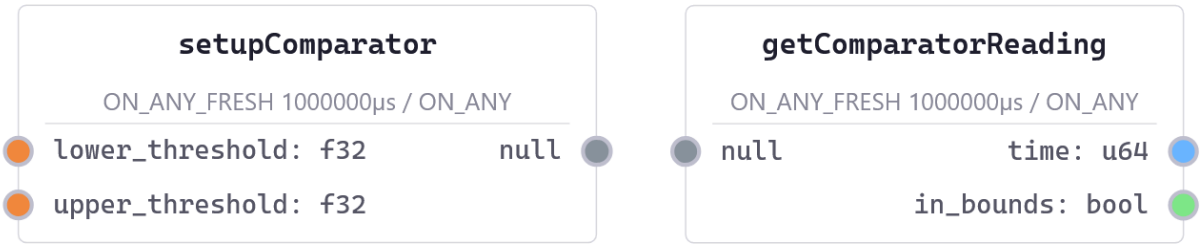


Figure 2.8: Functions tagged with `PIPES_FUNC` in Listing 2.16 are wrapped by PIPES and source their data schema and network interactions, to produce systems-wide programming models.

This could involve writing a smaller interface within each device and then collecting each of them piecewise into a larger, global IDL. This is what MQTT [119], MSGFlo [142], and some other embedded bridges into distributed systems do; XRCE-DDS [123], Micro-ROS [124], and [125], [126].

In PIPES the approach is to use function definitions directly, rolling those up into interfaces by way of modern programming patterns like template programming in C++ and function "decorators" in Python. These are "code introspection" tools that allow programs to analyze themselves, and the insight is that programming languages' own internal typing and naming schemes are descriptive enough to be used as interface descriptors directly. I cover some more background on this type of strategy (and how my use of it differs from the state-of-the-art) in Section 2.6.1.1.

The major benefit to this strategy is that it means that the schema we use to assemble systems has been built from a "single source of truth," which is the schema's implementation itself. Here, we extend that to mean not only that *schemas* are consistent, but that their representation as *functions* is consistent. In applying the same *operational* scheme across these functions, we can make sense of systems-wide operation by way of coordinating networks of functions.

The tool that I use for this is the `PIPES_FUNC` compiler macro in C++ and the `@pipes_function` decorator in Python. Both of them ingest function *implementations* and produce what I call *Pipes Functions,* both using interface classes that mediate between the implementation and the system, which I will describe below.

2.4.2 Functions as the Basic Building Block

Using functions directly allows module developers to reflect their native program representations directly to systems programmers, but it requires that we develop a network interface that wraps functions, attaches them to OSAP networks (on top of ports), and mediates between serialized network data and their actual implementations. This happens in two steps: first we build a state machine around the function (Figure 2.9) that will call the native function (the implementation) according to system-defined rules and based on network flows and messages, and then we build a controller for that state machine that can be queried remotely to generate a remote view of its configuration — another piecewise MVC layer (Section 2.4.2.6).

I call these interfaces Pipes Functions, in a sense they are like the Link Gateways from OSAP: they mediate between programmer-defined implementations (of functions themselves or of link drivers) and systems-defined operation of the same.

2.4.2.1 A State Machine to Embed Pipes Functions in PIPES Systems

The key figure here is 2.9, which describes the state machine and MVC that wraps each function implementation. I will describe each part of this figure in more detail in the next subsections, but for a quick map: we have the actual implementation at the core in *orange,* and see that runtime-native code can still call this function directly - so can other functions within a class instance (in the case where the function is a member of that class). The *green* blocks are inputs and outputs to the function - these are captured as the function's argument list and return values. Items in *blue* are network configuration and discovery handles, and *yellow* blocks represent various network transactions: the RPC call and interactions with other Pipes Functions as dataflow components (see Sections 2.4.2.3 and 2.4.2.4). *Grey* blocks represent external data sources and sinks.

2.4.2.2 The Function Signature

A function's "signature" is just its native interface: the function's human-readable name, the names of its arguments (and their types) and their output type. In many cases functions only return one value (or none), but in some languages and practices returning *tuples* (typed lists) is common. I found this to be an invaluable design pattern in PIPES and MAXL, as will become clear. Those are also typed, but it is not common for values *returned* by a function to have names; implicitly the function name itself is equivalent to the return value name. I added some tooling

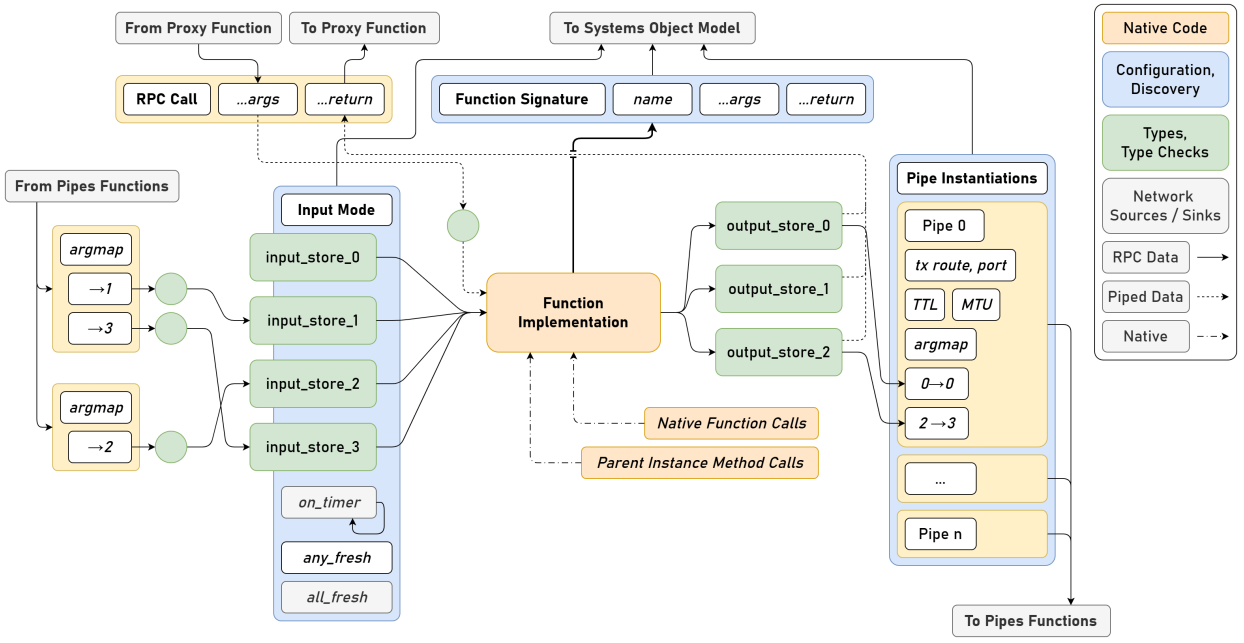


Figure 2.9: Diagram of the Pipes Function state machine with its user-defined implementation (center), typed input and output buffers and control logic. These embed the function into networks, allowing it to be called directly using simple RPC semantics or used as a component in a dataflow compute graph (or both).

that allows function authors to specify what names should be applied to tuples, which you can see in Listing 2.16.

The function signatures for our two loadcell functions are in the listing below alongside serialized configuration values for their input mode - this is how they appear in a serialized Systems Object Model. You will see that these also include a "localPort" value, which maps the function to the OSAP port where it is attached.

Listing 2.17: The function signature of `setupComparator` from Listing 2.16.

```
{
  "name": "setupComparator",
  "inputs": [
    {
      "name": "lower_threshold",
      "typeName": "f32",
    },
    {
      "name": "upper_threshold",
      "typeName": "f32",
    }
  ],
  "outputs": [
    {
      "name": "",
      "typeName": "null",
    }
  ],
  "inputMode": "ON_ANY_FRESH",
  "inputInterval": 1000000,
  "localPort": 4
},
```

Listing 2.18: The function signature of `getComparatorReading` from Listing 2.16.

```
{
  "name": "getComparatorReading",
  "inputs": [
    {
      "name": "",
      "typeName": "null",
      "value": null
    }
  ],
  "outputs": [
    {
      "name": "time",
      "typeName": "u64",
      "value": null
    },
    {
      "name": "in_bounds",
      "typeName": "bool",
      "value": null
    }
  ],
  "inputMode": "ON_ANY_FRESH",
  "inputInterval": 1000000,
  "localPort": 5
}
```

Not *any* function can be rolled up as a Pipes Function. Input and output types must be atomic values in the core PIPES type set (which includes most ANSI C types (plus strings), see Section 2.4.3), and there are practical limits to the *length* of type names and type values that are not clearly articulated to programmers; for example this architectural stack does not yet implement the transmission of multi-packet datagrams, so it is implicit that serialized data structures need to fit within a single packet.

It is also the case that Python functions are not always strictly typed, and so Pipes Functions can only be applied to functions that implement complete *type hints,* a somewhat recent syntax addition to Python.

2.4.2.3 Interactions with Remote Procedure Calls

Remote Procedure Calls (RPCs) are a common design pattern in distributed systems ([117] Section 4.2). They are atomic operations (one request, one response) and simply mirror the function call in a remote device. In Figure 2.9 the RPC is in the top-left, and data flows in that case are represented with a dashed line.

The RPC call packet is generated by a function proxy (see Section 2.4.4.2) and contains a message type key (denoting that it is a function call), and the input arguments that are serialized against the function signature. Serialized argument types are checked against the local signature and conversions are applied if they are available (again see the Pipes Types section 2.4.3), and then serializes the return values and replies to the caller.

So far this is just a classic RPC implementation, but in PIPES there is the added side-effect that the RPC call's operation of the implementation generates new output values (green blocks to the right of the implementation in 2.9). If Pipes (see the next section) have been instantiated on this function, those new values will be transmitted according to their configurations.

2.4.2.4 Function Interactions with Dataflow Graphs

To Pipes Functions we can attach *Pipes,* the atomic object (not the system itself PIPES). In Figure 2.9, these are diagrammed in a column (blue border, yellow Pipes) to the right of the function's output values.

A Pipe is a dataflow connection that collapses network and program operation. We can attach any number of Pipes to a function, but are again limited in embedded systems by memory (and everywhere by compute and network bandwidth) requirements for these objects.[59]

Each Pipe instantiates a new OSAP Port so that messages can be generated in parallel for each pipe and so that transport flow-control from one does not impede the other (etc). Each has a configuration, which is below:

Listing 2.19: Configuration values for a PIPES Pipe.

1. A *transmit route* and *port* that specifies how the data should be routed through the OSAP network, and to which Pipes Function port. This is configured against the Systems Object Model, and also includes the route's Maximum Transmission Unit (MTU, see Section 2.3.1.3).
2. An *argmap* that specifies which function outputs should be transmitted to which of the recipient's inputs. This amounts to two lists:
    a. A list of Output Arguments i.e. `0, 2` in the first Pipes instance shown in Figure 2.9.
    b. A list of Input Arguments i.e. `0, 3`.

[59] In embedded systems *Pipes* are allocated from a fixed-size pool that is instantiated in the runtime's memory (its size can be configured at compile time), a common design pattern and similar in nature to the one that I described for OSAP's packet access scheme in Section 2.3.1.4.

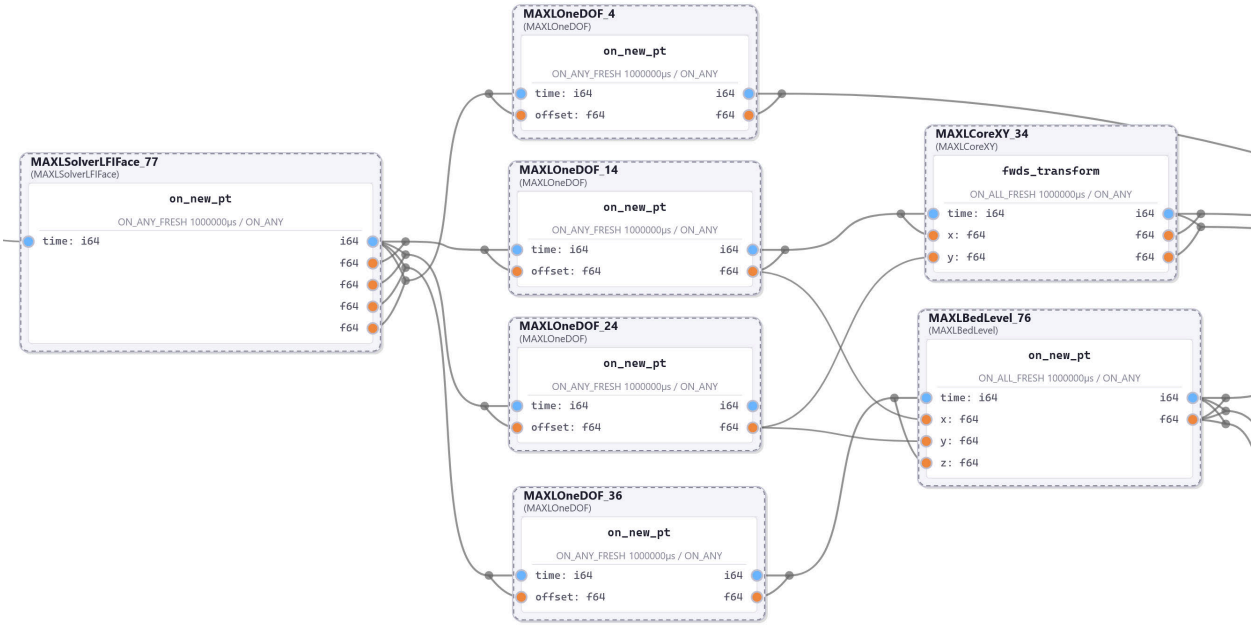


Figure 2.10: A view of seven Pipes Functions whose inputs and outputs are "woven" together using Pipe `argmap` configurations. In this example a motion control planner's output from `on_new_pt` (at left) is a tuple of `(time, x, y, z, e)` control points. That is mapped into a series of OneDOF MAXL blocks (see Section 2.5.4.2), each of which is applied to one axis only: the maps are e.g. `(0, 1) -> (0, 1)` for the second OneDOF (in x) and `(0, 3) -> (0, 1)` for the third OneDOF (the z axis).

Configuring Pipes such that they can "weave" output and input indices together in this manner was another key design pattern that I developed alongside MAXL, for example in Figure 2.10 below, where one example of this pattern is shown (and explained in the figure caption).

Pipes are responsible for serializing outputs, and do so whenever new outputs are available *and* their downstream transport layers are clear *and* a new packet can be allocated, if not they wait until these conditions are met. However they do not contain internal buffers and so when i.e. two new outputs are generated during an interval when the Pipe has been awaiting clearance to send data downstream, one of those outputs is missed.[60]

Woven inputs are applied to Pipes Functions at the left-hand side of Figure 2.9. When a Pipes Function receives data from another function, it deserializes these values and loads them into the input stores here, marking them as "fresh," meaning that they contain new data. They are additionally type checked / converted at this point if the configuration and types allow it.

The *input mode* configuration defines when the function will be called according to these values' freshness. I have developed three modes, `on_any_fresh` and `on_all_fresh`, which are explained by their names: one is used to eagerly call the function on any new data and the other is used to

[60]This represents a current limit in the PIPES system. In the work in this thesis it did not become problematic, but a more principled design may include provisions to configure functions with improved *flow control* rules at the PIPES level, e.g. not consuming new data from inputs until all Pipes are cleared — or configuring these rules on a per-Pipe basis.

wait for each to be updated before operating the function. In dataflow programming models this is known as "input resynchronization."[61]

It is worth noting that in this scheme we can easily write Pipes that transmit to other functions in the same runtime, or even that transmit data from the function back to itself. This is because PIPES is based on a networking model, not on direct graph analysis and execution as is the case in some other dataflow programming models like Grasshopper [172], where entire graphs are analyzed before they are run and loops cause infinite analysis cycles.

2.4.2.5 Functions as Class Members

I extended the PIPES system to include classes as groups of functions, but only completed this exercise in Python where code introspection is more straightforward.[62]

This has the main purpose of allowing us to deploy groups of functions that can change one another's output via the classes' internal state. Class functions can call one another in their local scope, and can update states that affect functions which are part of dataflows.

This side-effect means that we can *operate* dataflow objects via function calls, and this is a key capability for the merging of dataflow with scripting that PIPES implements. The MAXL design pattern emerged on this basis: each MAXL block is a Pipes Class that has a core function that samples timestamps (and other streamed control points), and then operates on those to generate new outputs. Other methods in the same class are used to update that core function's behaviour according to the current task being performed by a global control script.

Pipes Classes extend the Pipes Function; the constructor's function signature is captured in the same manner, and the Pipes Class MVC is extended such that we can recover which Pipes Functions are bundled within that class. Functions within classes can be marked with a leading underscore to indicate that they should not be included in the Pipes Class.

Finally, Pipes Classes can each implement a *loop* function; PIPES will call this function once on each cycle of its OSAP runtime loop. This can be used by the class to generate periodic outputs according to its own internal logic.

2.4.2.6 The Function's Model-View Controller

I have discussed the Model-View Controller (MVC) design pattern in a few places: an overview of the pattern and my implementation of a kind of *network* MVC in OSAP (Sections Section 2.2.3.5, 2.3.2.1).

Each Pipes Function also implements an MVC: the *model* being the function and the Pipes state machine (which encompasses the function signature, its input mode configuration and its output Pipe instantiations), the controller being a part of the network wrapper for that function, and the *view* being a piecewise component of the larger Systems Object Model.

[61] For example in MAXL blocks where it is important that we generate only one new output on each tick of the timer, but we have woven inputs to one of these blocks, the `on_all_fresh` input mode will hold operation until each input has received new data from upstream blocks. The CoreXY and BedLevel blocks in Figure 2.10 use this input mode. MAXL's operating logic is described in Section 2.5.1.

[62] I discuss how we might add this capability to embedded systems in Section 2.9.2.1.

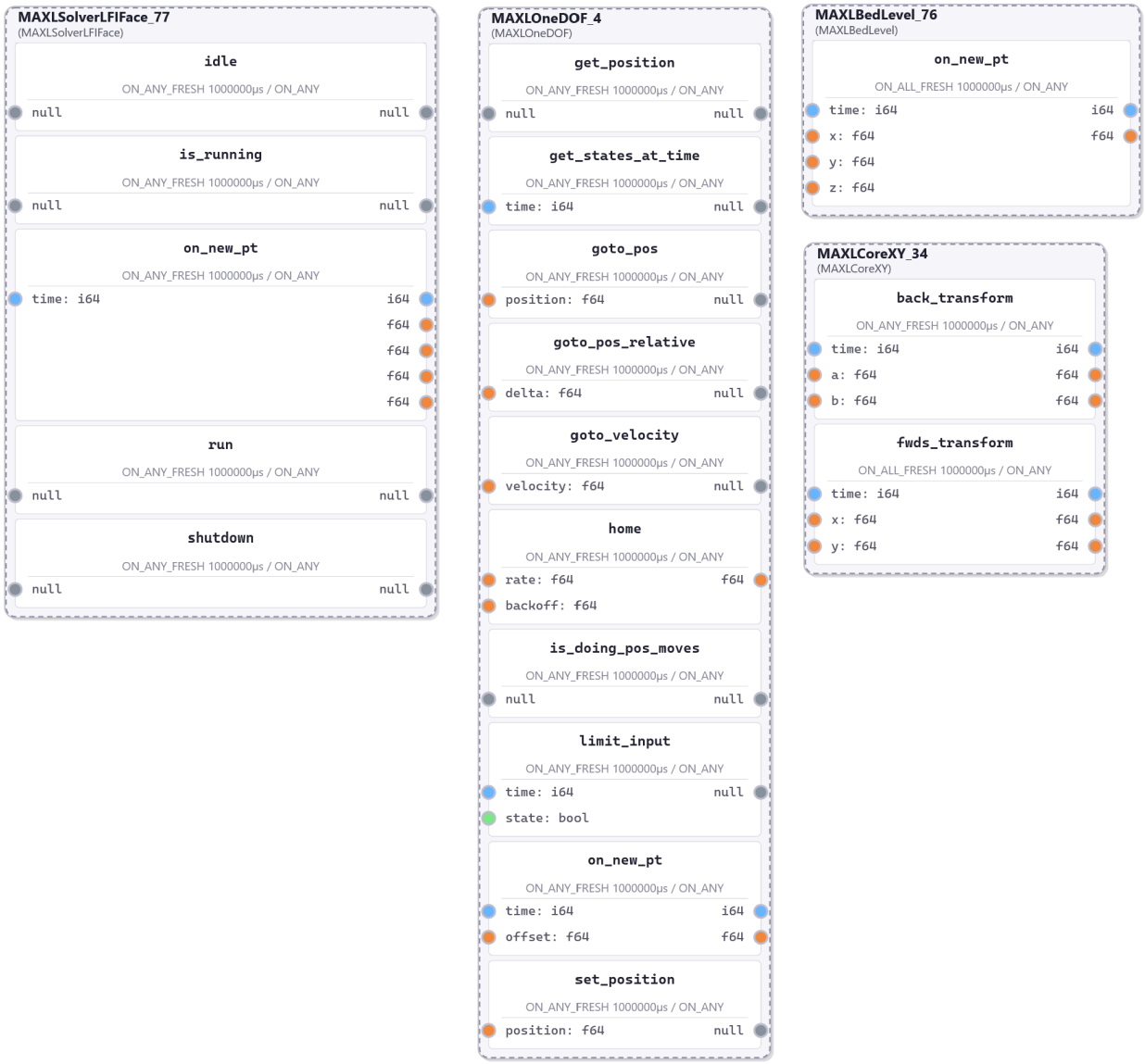


Figure 2.11: Four MAXL blocks, each of which is implemented as a Pipes Class containing Pipes Functions. Only some of these functions are used directly in dataflow graphs while others are used in scripts via remote procedure call, although any function can be used in both ways.

This pattern appeared in OSAP as well: each component (in that case the OSAP runtime) contains its own small MVC, and we run a larger cycle over a collection of these to create a global view at the systems level.

While we add a new OSAP port for each configured Pipe, each Pipes Function is initialized with just one port, and messages to that port are switched according to a pre-determined key code; this includes messages that

It is perhaps worth noting that a Pipes Function's output configurations (to wire data to other Pipes Functions) is useless without the surrounding network map and definitions of the recipient functions, this is the property of source routed networks that I have discussed in a few places. In

Listing 2.20: Declaring that this Python class (the OneDOF block 2.5.4.2) should be transformed into a PIPES Class using a class decorator. Only the class constructor is shown.

```
@pipes_class_implementer
class MAXLOneDOF:
    def __init__(self, max_vel: float, max_accel: float, output_scalar: float):
        self.max_vel = max_vel
        self.max_accel = max_accel
        self.output_scalar = output_scalar
        self.output_offset = 0.0
        # delta-tee tracker
        self.most_recent_timestamp = -1
        # states tracker
        self.position = 0.0
        self.velocity = 0.0
        self.velocity_target = 0.0
        # for offsets
        self._pos_offset = 0.0
        # stores latest limit states
        self.limit_time = 0
        self.limit_state = False
        # stores pts for history lookup, and blocks for motion
        self.control_points: Deque[MAXLOneDOFControlPoint] = deque(maxlen = 4096)
        self.segments: List[MAXLQueueSegment]
```

Section 2.4.4.1, I will explain how we use a global search to do systems-scale reconciliation of these piecewise descriptions.

2.4.3 Types and Serialization

PIPES uses a core and internally consistent serialization scheme that is based on the ANSI C set of datatypes; each type gets a systems-wide keycode which is used in the leading byte of every serialized value and also to serialize function signatures.

Listing 2.21: Pre-defined message types that can arrive at a Pipes Function port, to be handled either by the Pipes Function's Model-View Controller, or processed as inputs to the function itself.

1. Get Function Signature
2. Remote Procedure Call
3. Pipe Datagram (as a Dataflow Component)
4. Get Pipe Count (Dataflow Output Wires)
5. Get Pipe (retrieve one Pipe configuration)
6. Set Pipe (add one Pipe configuration)
7. Wipe Pipe (remove one Pipe Configuration)
8. Get Input Mode
9. Set Input Mode

Listing 2.22: Type Keys used in PIPES to annotate function signatures and runtime data streams. Quentin Bolsée and I designed this scheme together to be extensible in the future against array- and tensors of the same underlying values. Keys at left, numeric values, and behind comments `//` are names and bitwise representations.

```
#define TYPEKEY_NULL    0   // null   0b00000000
#define TYPEKEY_BOOL    1   // bool   0b00000001
#define TYPEKEY_UNKNWN  3   // unknwn 0b00000011

#define TYPEKEY_I8      16  // i8     0b00010000
#define TYPEKEY_I16     17  // i16    0b00010001
#define TYPEKEY_I32     18  // i32    0b00010010
#define TYPEKEY_I64     19  // i64    0b00010011
#define TYPEKEY_I128    20  // i128   0b00010100

#define TYPEKEY_U8      24  // u8     0b00011000
#define TYPEKEY_U16     25  // u16    0b00011001
#define TYPEKEY_U32     26  // u32    0b00011010
#define TYPEKEY_U64     27  // u64    0b00011011
#define TYPEKEY_U128    28  // u128   0b00011100

#define TYPEKEY_F16     33  // f16    0b00100001
#define TYPEKEY_F32     34  // f32    0b00100010
#define TYPEKEY_F64     35  // f64    0b00100011
#define TYPEKEY_F128    36  // f128   0b00100100

#define TYPEKEY_ASCII   48  // ascii  0b00110000
#define TYPEKEY_UTF8    49  // utf8   0b00110001

#define TYPEKEY_ARRY    64  // arry   0b01000000
#define TYPEKEY_TNSR    128 // tnsr   0b10000000
```

Listing 2.22 shows these keycodes, each PIPES system includes these in their source files - Quentin Bolsée and I designed this together in an attempt to make it extensible across compounding types, for example the bitmask of the Array type can be or'd with other key bitmasks to produce a compound. Table 2.5 shows how a Pipe is serialized, this matches the Pipe configuration that is rendered in Figure 2.9.

PIPES' serialization scheme is again designed to be more stateless than other schemes. It does so at the cost of packet size: each data stream contains full-width data and *also* type keys — where we are transmitting hundreds of thousands of these Pipes, adding a full byte per data object is hardly efficient. However, it avoids the state that would be required to establish stream type definitions ahead of their operation.

	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31
0	Pipe TX Key (8)	Argument Count (2) (8)	Arg Map 0 (0) (8)	Arg Map 1 (3) (8)
4	Type Key 0 (8)	Arg 0 (u64) (64)		
8	*…Arg 0 (u64)*			
12	*…Arg 0 (u64)*	Type Key 1 (8)	Arg 1 (bool) (8)	

Table 2.5: A serialized Pipe Output as described in Section 2.4.2.4. The succeeding packet header would encode the Pipe's transmit route and receiving function's port index (see Table 2.4 from the OSAP section on networking 2.3.1.3). I am using the argument mapping from Figure 2.9 and types from the load cell's `getComparatorReading` defined in Listing 2.16.

2.4.4 PIPES' Systems Programmer's Model: Scripted Dataflow

Configuring and tasking PIPES systems is done with two interfaces to one underlying representation, which is the Systems Object Model (SOM). Configuration and tasking is done only in Python at the moment, but it should be possible to develop similar design patterns in embedded devices. The complexity in these steps is much deeper though, and in many cases we prefer to run configuration and tasking steps from a more easily edited and more inspectable language like Python.

I developed a `MetaManager` class (it first appeared in Listing 2.6) to collect and manage the SOM. It is attached to an OSAP port in the host runtime and available there as a software object. In collecting the SOM *from* devices, we are essentially sourcing the system's modular data schema from all the system participants themselves, and then using that unique SOM to program against.

The primary programming interfaces are remote procedure calls to Pipes Functions — which is done using *proxies* that are written based on the SOM (in Section 2.4.4.2) — and the `manager.connect()` function, which wires Pipes Functions together (see Section 2.4.4.3).

We cannot remotely instantiate new Pipes Classes or Functions, instead they are manually instantiated in each runtime. This is a major limit to overall systems reconfigurability and prevents us from more completely modifying lower-level controllers — e.g. adding PID code modules to existing hardware, that run in hardware — which could be an invaluable capability. In my Masters' thesis [167] I did develop a version where embedded classes and function *could* be instantiated (and deleted) remotely, and I cover some of the steps that would be required to recover this capability in Section 2.9 (on future work).

We can still instantiate Pipes Classes within the Python runtime where our configuration code and tasking script lives, and use the system to develop flows between those classes and remote runtimes. This is productive enough to accomplish the controls tasks in this thesis.

2.4.4.1 A Systems Object Model

The Systems Object Model is not unlike the browser's DOM, it is a hierarchical list of objects of predictable types but with unique names and properties. Objects in the SOM are represented by

Listing 2.23: An enumeration of the objects in the PIPES Systems Object Model.

1. A list of OSAP Runtimes, within each:
    a. A list of Link Gateways,
    b. A list of Ports,
    c. A list of Pipes Functions that have been globally instantiated there,
    d. A list of Pipes Classes that have been instantiated there.
2. A list of Links, each of which has:
    a. A source Runtime and Link Gateway,
    b. A destination Runtime and Link Gateway.
3. A list of Pipes, each of which has:
    a. A source:
        a. Runtime,
        b. Function Name,
        c. Class Instance Name (if the function is a member of a class).
    b. A destination:
        a. Runtime,
        b. Function Name,
        c. Class Instance Name (if the function is a member of a class).
    c. Output and Input indices to map the source function's return tuple items onto the destination function's arguments (the `argmap` from Listing 2.19)

their human-readable names, which are mapped to network addresses "under the hood" by the Manager but are still available on closer inspection of the structure.

The SOM is extended on top of OSAP's network map, which is also used as the basis for discovery of the current configuration. For example the network map contains runtimes, link gateways and links (which connect gateways), and ports - each port has a `type name` (see Section 2.3.2.1), in the case of Pipes Functions these are human-readable as `"pipes_function_rx"`, and in the case of Pipes (which transmit dataflows), as `"pipes_pipe_tx"` etc. These handles are used to generate additional Pipes Function MVC requests on those ports, collecting the software map.

The SOM is a flattened representation of a graph that may have loops, which simplifies its representation and allows us to serialize it as a `.json,` which I tasked an LLM to render as a visualized graph in a browser-based tool, that program is what generated figures like 2.12 based on SOMs that were saved from some of the systems that I deployed. That figure renders the FrankenPrusa's controller (from Section 5.4.2).

The flattened representation is also used by the `Manager` class for the slightly more difficult task of building network routes between two Pipes Functions, as I will describe in Section 2.4.4.3 and for writing *proxies,* which I will describe in Section 2.4.4.2.

2.4.4.2 Proxies for Remote Components and RPCs

The SOM stores the underlying systems representation, but it is not very useful to directly program against. Instead we use *proxies,* following the Object Oriented Hardware pattern; they are software classes that can be instantiated in Python within one OSAP runtime, and that represent remote

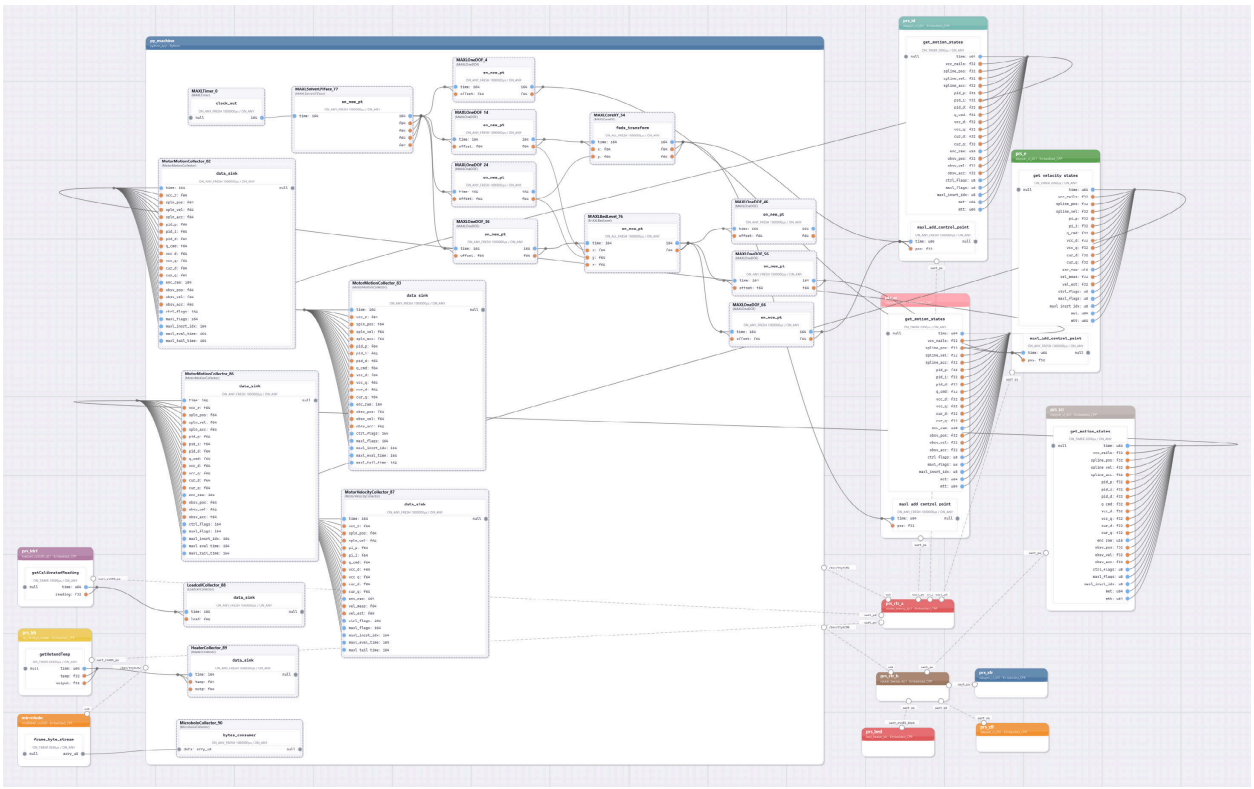


Figure 2.12: The PIPES / MAXL system that runs the FrankenPrusa (5.4.2). In this case the system is rendered in a configuration where only parts of the underlying system are in use, to perform a test.

hardware: when we make function calls on these classes they operate that remote functions' RPC and return the result. Like the PIPES manager, proxies are at the moment only developed for and deployed in Python.

I use the name *proxy* because I think that it more clearly articulates what the class is: an interface / stand-in object, rather than really "representing" the device itself. I used a templating language called Jinja [173] to write a "metaprogramming" tool that ingests the SOM and writes a series of `.py` files containing proxy definitions with one file and class definition per runtime / device, an example of which is in Listing 2.24.

Proxies provide a typed interface that can be called directly using RPC semantics *or* used as a handle to make connections, as I describe under the next header. Instantiating a proxy in your script implies to the manager that you expect that the device is present in the network at the time the script runs, when these instantiations are made the manager checks the proxy's configuration against the most up-to-date SOM and if types or names are mismatched produces an error. This effectively prevents us from running systems against mistaken configurations.

Because systems themselves are mostly stateless before our configuration tool runs, we can run this process *once* whenever we have a new set of hardware configured, write the proxies, and then import those into a new script that we use as the basis for systems development.

Listing 2.24: The proxy class definition in Python for the Load Cell Knuckle from Section 3.2.

```
class LoadcellCs5530D21:
    def __init__(self, manager: 'MetaManager', runtime_name: str):
        # a network-capable interface for each function definition
        # found in this device from the SOM,
        self._get_raw_loadcell_reading_proxy = PipesFunctionProxy(
            manager, # the system manager, which contains the loaded SOM
            runtime_name, # the system-unique runtime name to look for,
            "global", # class instance name, or 'global'
            "getRawLoadcellReading") # the function name
        # ...
        # ... omitting "_set_calibration_params_proxy"
        # ... omitting "_get_calibration_params_proxy"
        # ...
        self._setup_comparator_proxy = PipesFunctionProxy(
            manager, runtime_name, "global", "setupComparator")
        self._get_comparator_reading_proxy = PipesFunctionProxy(
            manager, runtime_name, "global", "getComparatorReading")

    # the programmer's interface is to these functions,
    # which are assigned type hints according to discovered function signatures
    # and adding """comments""" to roughly document the functions
    async def get_raw_loadcell_reading(
            self)-> Tuple[int, int] :
        """
        returns (time: int, reading: int)
        """
        time, reading = await self._get_raw_loadcell_reading_proxy()
        return cast(int, time), cast(int, reading)

    # ...
    # ... omitting "set_calibration_params" and "get_calibration_params"
    # ...

    # the "asyncio" interface releases the runtime while the network
    # retrieves the remote procedure call,
    async def setup_comparator(
            self, lower_threshold: float,  upper_threshold: float) -> None:
        await self._setup_comparator_proxy(lower_threshold, upper_threshold)
        return

    async def get_comparator_reading(self)-> Tuple[int, bool] :
        """
        returns (time: int, in_bounds: bool)
        """
        time, in_bounds = await self._get_comparator_reading_proxy()
        return cast(int, time), cast(bool, in_bounds)
```

2.4.4.3 Configuring Dataflows

To build Pipes Connections between functions, the manager implements a `.connect()` function. For arguments, it takes the *proxies* for the source and destination functions, and then argument lists (outputs, inputs) to map onto the pipe. For example Listing 2.25 writes the connections that result in the configuration shown in Figure 2.10 above.

When making these connections, the manager checks function signatures from the SOM to ensure that the connection's output and input types either match or will be easily converted in hardware, and then finds the appropriate network route between the two devices (by traversing the SOM) and each function's local port to complete the configuration. In `.connect` we can also specify the Pipe's time-to-live as a network quality of service handle, to prioritize certain Pipes performance over others. Once the Pipe is in place, it updates the local SOM to reflect this.

Finally, proxies are *also* the handle through which we configure remote functions' input modes, which I discussed in Section 2.4.2.4.

Listing 2.25: The function calls that connect a 3D printer's velocity planner (here it is named as the "solver," highlighted in 2.12) to a MAXL Timer (2.5.4.1), and then to MAXL OneDOF blocks (2.5.4.2)

```
# maxl timer pushes events to the solver,
await manager.connect(machine.timer.clock_out, solver.on_new_pt)

# the solver transmits points from new solutions one-by-one to OneDOF blocks,
await manager.connect(solver.on_new_pt, machine.dof_x.on_new_pt, "0, 1", "0, 1")
await manager.connect(solver.on_new_pt, machine.dof_y.on_new_pt, "0, 2", "0, 1")
await manager.connect(solver.on_new_pt, machine.dof_z.on_new_pt, "0, 3", "0, 1")
await manager.connect(solver.on_new_pt, machine.dof_e.on_new_pt, "0, 4", "0, 1")

# leveler gets x, y states to evaluate the level correction at that position,
# and z, which it offsets using the correction,
self.leveler.on_new_pt.set_input_mode('all_fresh')
await self.manager.connect(self.dof_z.on_new_pt, self.leveler.on_new_pt, "0, 1", "0, 3")
await self.manager.connect(self.dof_x.on_new_pt, self.leveler.on_new_pt, "1", "1")
await self.manager.connect(self.dof_y.on_new_pt, self.leveler.on_new_pt, "1", "2")
# the leveler transmits updated z-targets to each z-motor's OneDOF block;
# instantiating one of these per motor is useful because each needs
# to be homed and offset independently of the others
await self.manager.connect(self.leveler.on_new_pt, self.dof_zfl.on_new_pt, "0, 1", "0, 1")
await self.manager.connect(self.leveler.on_new_pt, self.dof_zfr.on_new_pt, "0, 1", "0, 1")
await self.manager.connect(self.leveler.on_new_pt, self.dof_zrr.on_new_pt, "0, 1", "0, 1")

# corexy transforms the solver's XY outputs into AB values to suit
# the machine's kinematics
self.corexy.fwds_transform.set_input_mode('all_fresh')
await self.manager.connect(self.dof_x.on_new_pt, self.corexy.fwds_transform, "0, 1", "0, 1")
await self.manager.connect(self.dof_y.on_new_pt, self.corexy.fwds_transform, "1", "2")
```

2.5 MAXL: Motion Control with Dataflow

Modular Acceleration planning and eXecution Library

The development of MAXL is motivated by two main goals that I described earlier in Section 2.1.3.1. The first is to build motion controllers that expose a better software interface to their *velocity planners.* They are key machine control components that cause machines to accelerate and decelerate in order to smooth motion and avoid exceeding kinematic and actuator constraints (Sections 4.2.4 and 4.6 both describe this step in more detail). Trajectories encoded in GCodes cannot be interpolated in time because velocity planners live *beneath* the GCode interface. As I have discussed, this causes issues for machine builders and process researchers — the instructions that we send to the machine are modified, and we cannot ascertain what was actually done when our instructions (GCodes) ran. The second is to develop motion control as a system of modules that can be easily reconfigured for a variety of machines.

The challenge in this regard is threefold: first, we need synchronized execution of a planned trajectory in our distributed modules. The basis for synchronization in MAXL is OSAP's clock sync service (Section 2.3.2.2), but we still need to write those trajectories and interpolate them. The second is to split operation of the controller across the realtime gap so that we can build smart controllers in high-level programming languages (which are not normally deterministic) but still smoothly run trajectories in embedded hardware, where we also do sensing of important machine states. Finally, we have the challenge of describing motion control in a modular fashion so that we can build controllers for a range of machines using a shared core of software components.

For these tasks I wrote MAXL (Modular Acceleration planning and eXecution Library); it develops motion controllers as dataflow graphs. We can then use PIPES configurations of MAXL blocks to combine kinematic transforms and corrections, look-ahead planners (4.2.4 and 4.6), and a host of other motion planning utilities (2.5.4) into new controllers.

2.5.1 MAXL's Operating Principle

MAXL works by sequentially transmitting trajectory segments between dataflow blocks. Most of these blocks (Section 2.5.4) are written in Python as Pipes Classes, and trajectory interpolators are written in C++ as Pipes Functions. I originally developed the basic premise in [164]: by encoding trajectories as some set of functions that can be interpolated on a time basis, we can simplify trajectory execution so long as we have synchronized device clocks. To do so we broadcast trajectories throughout our system — ensuring that time-encoded segments arrive *ahead* of the time when they will be used — and then at any time, firmwares can query trajectories against their synchronized clock to pick an appropriate action. For closed-loop motors this means updating their position reference at some interval, and for steppers it means issuing a step signal whenever



Figure 2.13: Using time-based representations for motion that *include* the results of lower level machine controllers (velocity planners) allows us to more easily add components to (or align data with) those trajectories. For example to synchronize the on/off states of the LEDs used for this light painting from the original MAXL paper, the LED controller needs to know exactly *when* it should flip those switches.

the interpolated value changes by more than one step, and so on. The scheme also makes the *reconstruction* of machine motion more straightforward computationally, because trajectories and sensor readings can all be reconciled as one time-series.

In that earlier paper, I developed a few types of interpolable functions: linear trapezoid segments to encode motion *event* tracks, which simply change value at set times and can be used as triggers for sensors or control set points for lower level systems. In this implementation I use only one: cubic basis splines (Section 2.5.3). I also extend the concept to use a dataflow configuration. The PIPES section already showed some examples of MAXL components, another is in Figure 2.14. This allows us to reconfigure motion control systems by re-wiring graph connections rather than re-writing software, and helps us to include motion control *as a component* of the larger distributed machine control system.

The premise in MAXL is that:

1. Device clocks are synchronized.
2. Timestamps *tick* along at a fixed interval, which is set by the Timer block 2.5.4.1. It outputs a new timestamp once in every interval.

3. Other blocks (a set of which are explained in Section 2.5.4) can be combined using dataflow configurations to modify and generate motion control trajectories.
    a. Blocks trigger on these timestamps, using whichever logic they would like to implement to generate a new output value at that time, passing those through the dataflow chain alongside the original timestamp. In many cases, blocks add, multiply, or otherwise combine their contributions to that value with inputs that were generated by other blocks i.e. adding offsets or transforming from cartesian to actuator spaces.
    b. Any component can save local trajectory history, so that sensor readings and events collected later on can be compared with those histories.
4. Besides operating or modifying these values, MAXL blocks can also *interpolate* them to produce reference signals that can be attached to hardware outputs (e.g. as position or velocity targets in a servo).

2.5.2 The MAXL Gap

In MAXL, components are distributed across networks and some motion components are computed in non-deterministic languages and operating systems. This means that we need a strategy to ensure that the time gap between trajectory generation and interpolation is *no shorter than* the maximum interval that it may take for the trajectory point to reach the interpolator.

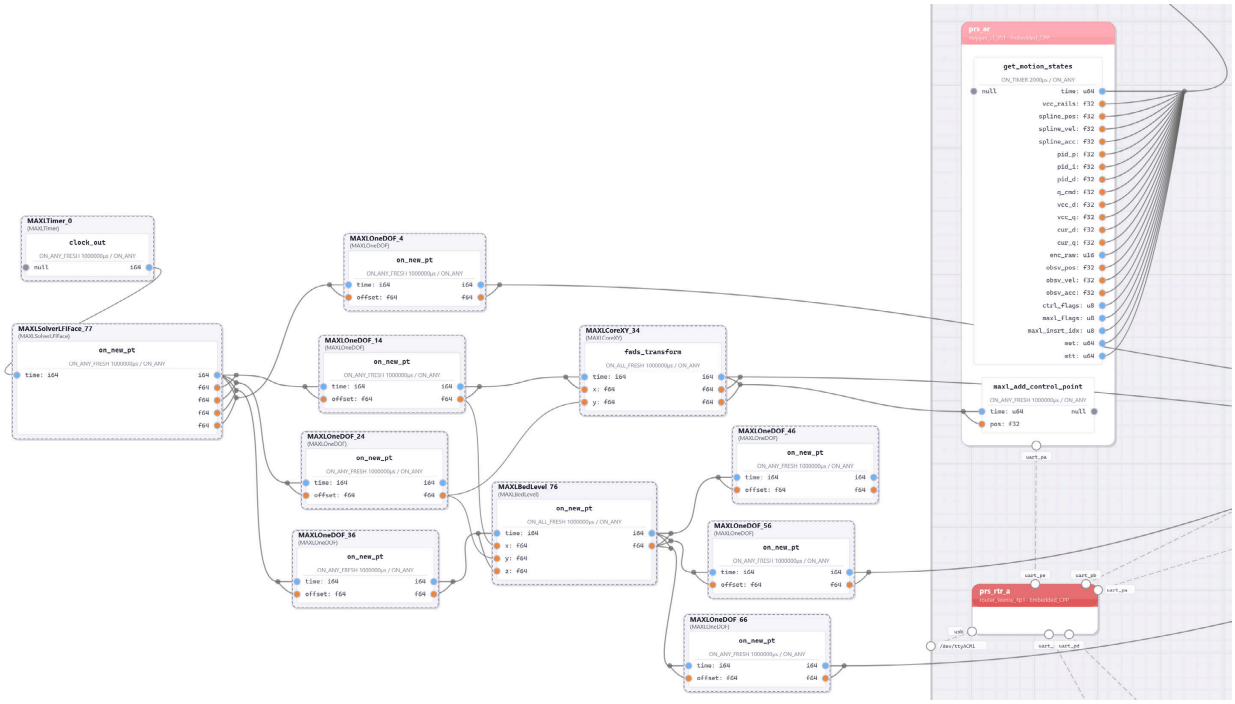


Figure 2.14: A MAXL dataflow graph, overlaid in PIPES. Top left is the timer, which issues time stamps that stream through motion components. Here we have a velocity planner (left, as "solver"), multiple OneDOF blocks, a CoreXY transform and a bed levelling utility. At right, one of the machine's motors is shown with the basis spline interpolator. A representation of the network is also shown, the `prs_rtr_a` is a message passing device, dashed lines represent network segments.

For simple linear segments this would mean that at any time interpolators have at least one control point in the future to interpolate towards. Basis splines require that they have two future control points in memory (and one in the past, see Section 2.5.3). At the same time, a smaller gap is better in many cases because it increases the high level bandwidth in our system.

So, in MAXL we configure a *gap;* the timer block generates a new timestamp whenever the current system time *plus* this gap exceeds the value of the *last* timestamp that it transmitted. The gap size can be configured against network and determinacy measurements such that it is minimized but is still larger than the minimally safe value.

In Figure 2.15 I diagram how this works; timestamp generation happens ahead of time and triggers blocks (in green) to compute new outputs. Those compute steps each add some delay, as do network segments in the graph. When the control points arrive at their interpolators they are stored in local queues where they are inserted with reference to their timestamp (so interpolators can also ascertain whether they have missed a point).

I have already mentioned why time-based encodings are valuable for the recreation of time series data and e.g. their combinations with sensor readings, but using a *fixed* interval also has some

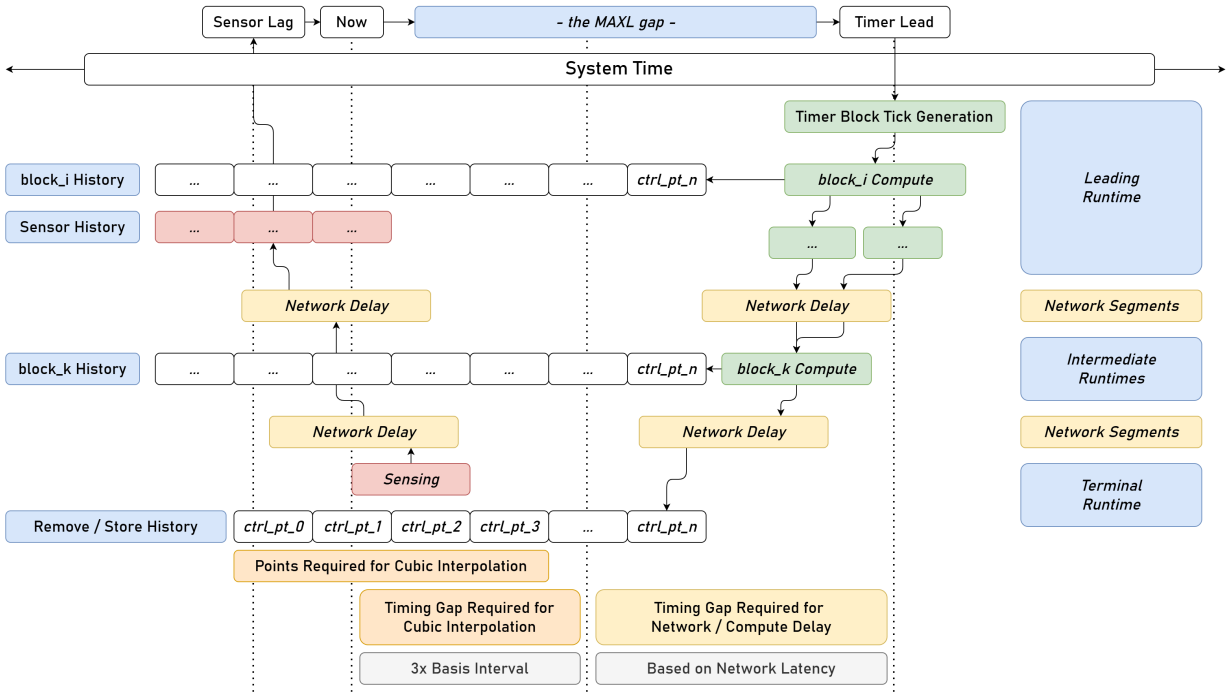


Figure 2.15: The MAXL *gap* is key to the scheme's operation. The timer block generates timestamps that lead the system clock by this gap, which leaves enough time for values that are computed on top of that timestamp to reach the devices that will interpolate them. This figure diagrams MAXL blocks in green, which are spread across three runtimes (in blue), and connected across two network links (in yellow). Buffers of control point history can be stored in time-series to recreate trajectories seen by whichever block chooses to store them. I also show sensor data flowing back through the same system and being added to a synchronized stream of samples.

benefits. The first is straightforward; blocks that operate as integrators can then do so using a repeating Δt, which can simply them numerically.

The second has to do with network loading. Transmitting at a fixed interval makes for deterministic network loading; the amount of data that our network has to transmit in order to operate the controller does not change dramatically during the course of operation. We can compare this to spatial encodings that are used in GCode (line and arc segments), Klipper (small velocity-planned line segments) and i.e. StepDance and Urumbu (steps are small spatial deltas). Spatial encodings require that we transmit more data when we make faster or more complicated moves, and less at other times.

For example CNC milling machines often make long, linear traverses and then *helical* entries into the succeeding cut. The traverse is one line of GCode that takes a very long time, but the helix is composed of very many small linear segments (or small arcs) that each take a very short amount of time. This means two things: first is that the bandwidth requirement between the GCode "sender" and the interpreter can vary greatly over time (slow, and then fast very suddenly), and that the GCode interpreter itself must balance the two tasks (planning prior segments, running control outputs — and reading new GCodes, rebuilding them as internal representations, and adding them to the queue) simultaneously. These issues have largely been solved by modern interpreters by increasing their performance across the board, but some can still stutter under stresses like this. The underlying phenomenon means that we have to select *queue sizes* — memory buffers, not space or time — for our interpreters against these constraints: too long of a queue and our machine becomes unresponsive to new inputs, too small of a queue and the planner may be starved where geometries are dense.

2.5.3 Synchronized Basis Splines

To describe motion in time, MAXL uses a *cubic basis spline* interpolation (often just called a B-spline) with control points (aka *knots*) at fixed time intervals. Basis splines are commonly used to representation motion because they have well-defined and smooth definitions of velocity and acceleration with step functions in jerk [174]. This matches particularly well to inertial systems controlled by electric motors because they have the same order. I will discuss this in Section 4.3.1; our actuators cannot instantaneously change the amount of torque (acceleration) they are exerting, since it takes time for an applied voltage on the motor stator to develop into current. This means that electric motors cannot instantaneously change accelerations, although they *can* make instantaneous changes to the rate of change in acceleration (i.e. voltage, jerk).

I discuss how they contribute to the overall *composability* of MAXL in Section 2.8.2. Interestingly they also relate well to constraints-based optimization in both the network optimization sense *and* in the motion control sense, I discuss that coupling in Section 7.4. They do add a small amount of interpolation error because they are not *direct interpolations,* the interpolated path does not pass directly through their control points. I quantify this error in Section 7.4.1.

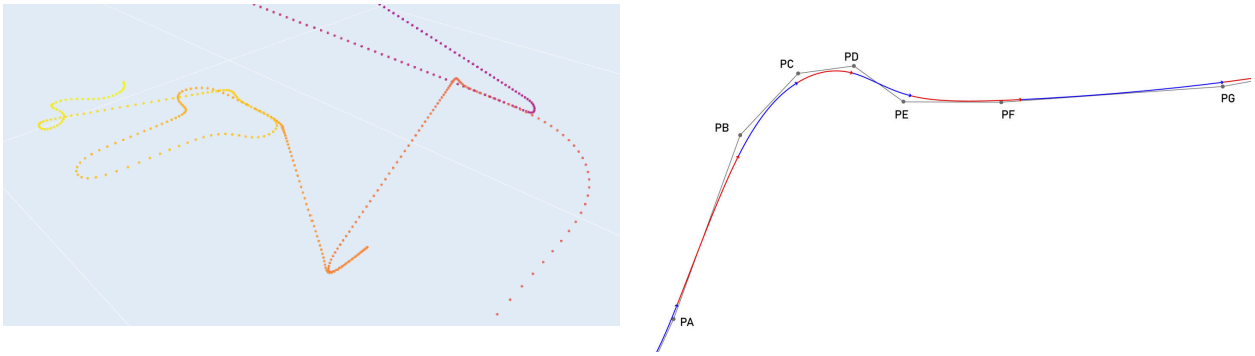


Figure 2.16: MAXL uses basis spline interpolation as a generic intermediate representation for motion. In this figure, I render a subset of spline control points that were generated by MAXL to stream to motors during a print job. At left is a velocity-planned trajectory segment in 3D rendered via its control points alone. At right: A 2D subsection of a basis spline, showing a stream of control points ($PA \rightarrow PG$) and the resulting curve.

$$\begin{bmatrix} P \\ V \\ A \\ J \end{bmatrix}(t) = \begin{bmatrix} 1 & t & t^2 & t^3 \\ 0 & 1 & 2t & 3t^2 \\ 0 & 0 & 2 & 6t \\ 0 & 0 & 0 & 6 \end{bmatrix} \frac{1}{6} \overbrace{\begin{bmatrix} 1 & 4 & 1 & 0 \\ -3 & 0 & 3 & 0 \\ 3 & -6 & 3 & 0 \\ -1 & 3 & -3 & 1 \end{bmatrix}}^{A} \begin{bmatrix} P0 \\ P1 \\ P2 \\ P3 \end{bmatrix} \tag{2.1}$$

Equation (2.1) is the cubic basis-spline form that MAXL uses, which defines position, velocity, acceleration, and jerk (P, V, A, J of (t) — not all need to be evaluated) at any time based on the points in A. Points in A can be vectors of any dimension; splines decompose well on a per-axis basis. The time t spans a fixed interval (Δt), and the interval is set at some integer value of microseconds that is a power of two, between 256us and 16384us. Using these intervals means

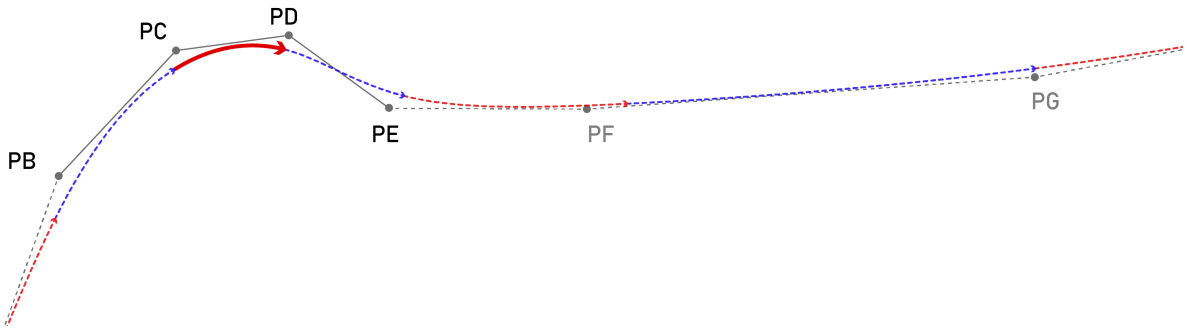


Figure 2.17: Basis splines can be interpolated through position, velocity, acceleration, and jerk values for the curve that they represent. Control points are defined by their position only, the other derivatives are implied by the spline formulation itself. For example the solid section of this curve (in red) lies primarily between $PC \rightarrow PD$, but is defined by points $PB \rightarrow PE$. All four points are combined to define the curve shape, its velocity, etc.; moving PE to the right would increase the velocity throughout this segment. New points can be freely added to the tail of these curves, which is a useful property for defining continuous motion. For example the next curve segment (in blue, dashed) is defined by $PC \rightarrow PF$ and so-on.

that the spline can more rapidly be evaluated using fixed point arithmetic in embedded devices. Basis splines have the helpful property that we can always add new points to the end of a stream, meaning that at each interval we only need to stream one new position whereas e.g. a velocity-planned linear segment of similar length would require much more information.

2.5.4 MAXL Blocks

MAXL Blocks are Pipes Classes that each perform a subset of the overall motion control and coordination task. I use them throughout this thesis to compose both complete motion systems and smaller controllers and calibrations. In these sections I provide a brief overview of how each functions and which roles they play.

2.5.4.1 Timer

The timer block leads graph, generating one new timestamp within each interval. These *ticks* then flow through downstream blocks to produce new outputs.

We only have one timer block in any MAXL graph and it is configured with the system's Δt and an `advance` parameter to span the appropriate *gap* (as in Section 2.5.2). For example if our interval is 1024μs and `advance=16,` the timer will generate a timestamp for $t = 16384\mu$s at $t = 100000\mu$s, effectively leading the system clock by those 16384μs.

To decide if a new timestamp should be issued the block uses its loop function to evaluate OSAP system time against the defined gap size; if the current system time plus the gap is beyond the

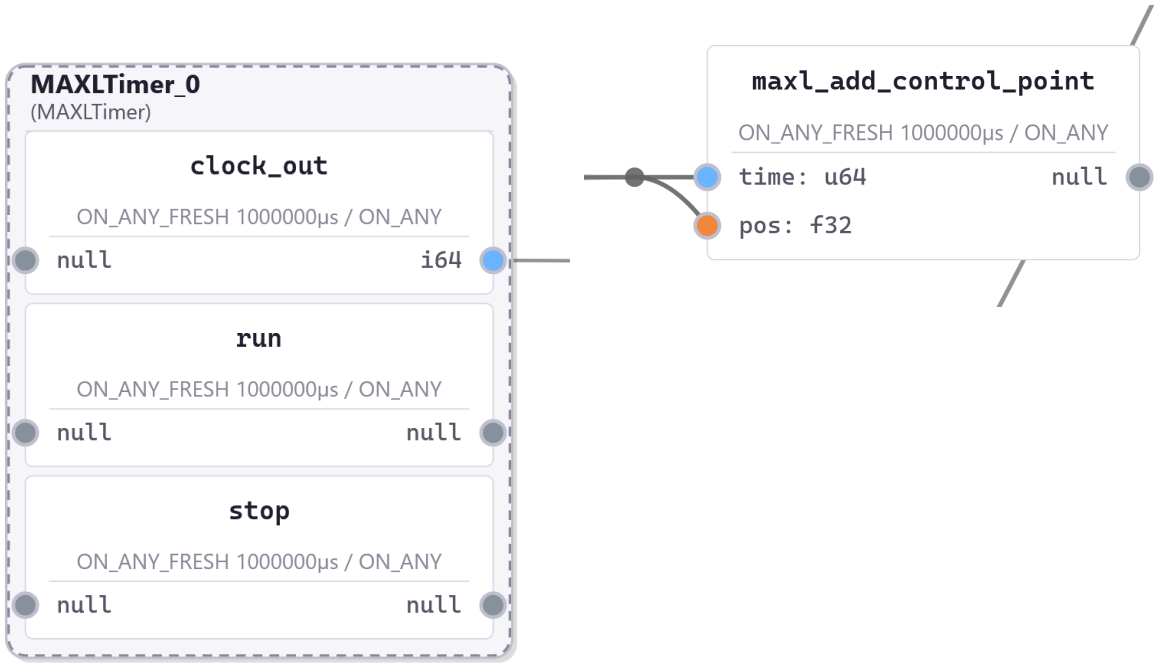


Figure 2.18: The timer block (left) is the single source of any MAXL graph, and the interpolator (right) is the final sink; there can be many interpolators.

most recently transmitted timestamp, it generates a new one. This is rather than relying on i.e. a perfectly scheduled call to the timer output. If the runtime is temporarily blocked for more than one interval the timer will output a series of points in succession in order to catch up; the *computation* of the MAXL graph does not need to happen according to precise timings so long as smooth interpolation of the control points is maintained.

2.5.4.2 OneDOF

The OneDOF is a utility for controlling single degrees of freedom (DOFs). It is used in-line with axes of motion to add general purpose motion functionality to those axes. It contains a single-segment trapezoid generator that can be used to move DOFs smoothly from point to point and a velocity control mode that applies acceleration control, and is included in Figure 2.11.

This block also implements a simple homing routine where the limit switch signal is exposed as a Piped input, meaning that we can source home signals from any module in the system. For example to level 3D printer beds, I use a comparator output from the load cell sensor to trigger this "switch" in combination with a OneDOF block that is chained into all three z-motors.

2.5.4.3 Chirp Generator

The chirp generator is another simple motion utility that I use in the generation of preliminary motion models. It writes chirp time-series using scipy [175] and interpolates through them to produce outputs. These excite motion systems across a range of frequencies, which is useful for systems analysis.

2.5.4.4 Trapezoidal Planner

This is a classical type of velocity planner that I describe in Section 4.2.4. It ingests line segments, queues them, and does look-ahead velocity planning across them using direct parameters for a kinematic system's per-axis maximum accelerations and velocities and a geometric parameter for deviation from path corners. I use MAXL and PIPES to virtually swap between this planning block and the model-based planner (block below) to compare the performance of this style velocity planner (which is the basis of most state-of-the-art motion controllers) with my model-based planner in Section 4.7.1.

2.5.4.5 Optimization-Based Planner Interface

The optimizers developed for velocity planning in this thesis are integrated within machine systems using MAXL via an interface block that I describe in Section 4.6.2.

2.5.4.6 CoreXY Kinematics

This block implements CoreXY [176] kinematics and is used in both 3D printers that I developed (Section 5.4.1 and 5.4.2).

2.5.4.7 Scara and Kaos Kinematics

Both of these were developed in a machine building workshop that I discuss in Section 2.7.5.1, in an earlier version of MAXL and PIPES. The Kaos kinematics (Figure 2.21) require only a small update to the CoreXY function (for the angle of the V-shaped structure). The Scara kinematics (Figure 2.22) are nonlinear,

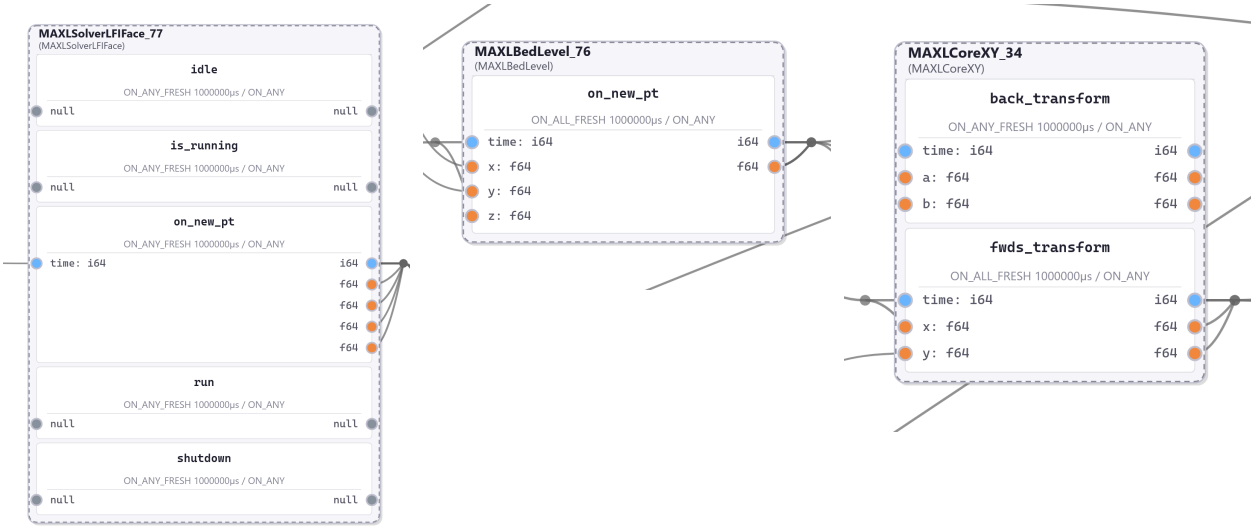


Figure 2.19: The model-based velocity planning solver interface, bed leveller, and CoreXY MAXL blocks.

2.5.4.8 Bed Level Corrector

In FFF 3D printing it is critical that print beds are well aligned with the machine's motion system so that the first layer of a print is of uniform thickness. However, beds often warp or are slightly misaligned with the cartesian side of the machine.

To correct for this, it is common to apply a transform to the system that adds offset in the z-axis which depends on the systems' position in xy. I developed the Bed Leveling MAXL block for this, which applies a planar function (whose parameters have been fit to measurements) to a stream of z-points, based on measurements in the stream of x and y points.

Listing 2.26: MAXL's implementation of CoreXY kinematics.

```
@pipes_class_implementer
class MAXLCoreXY:

    def fwds_transform(
            self, time: int,
            x: float, y: float) -> Tuple[int, float, float]:
        a = (x + y)
        b = (x - y)
        return time, a, b

    def back_transform(
            self, time: int,
            a: float, b: float) -> Tuple[int, float, float]:
        x = 0.5 * (a + b)
        y = 0.5 * (a - b)
        return time, x, y
```

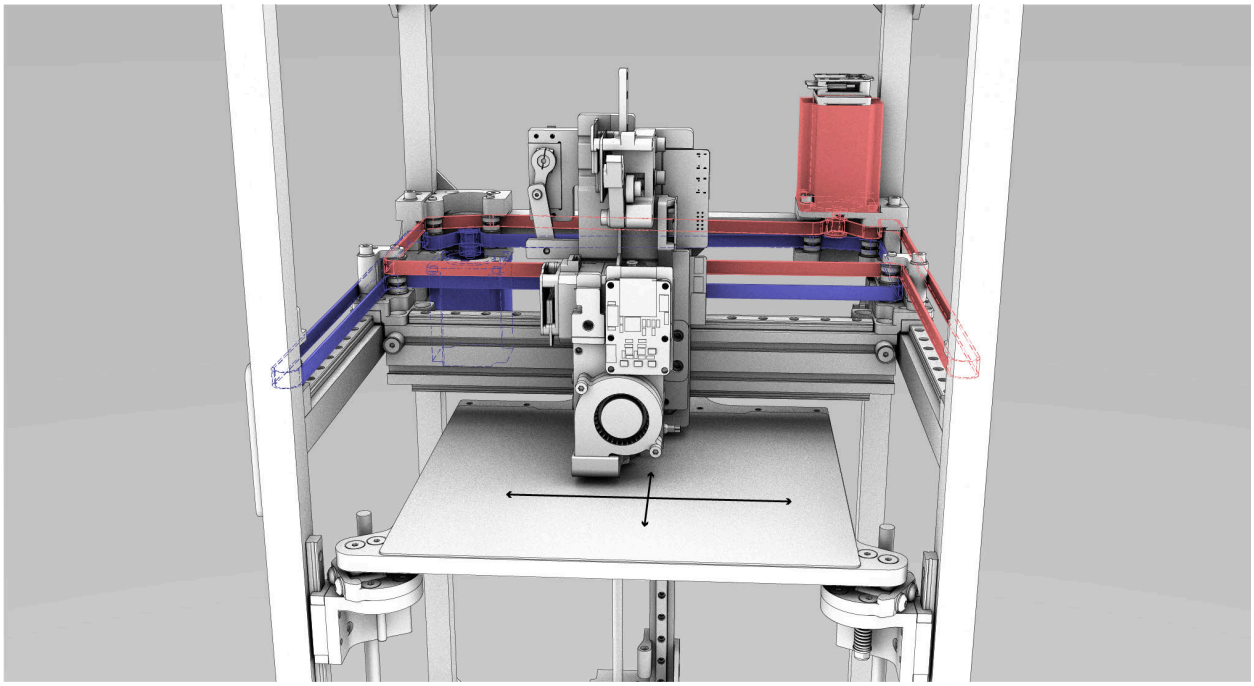


Figure 2.20: CoreXY motion systems use two motors (a and b) in parallel to control cartesian (x and y) motion systems. In this render of the RheoPrinter (Section 5.4.1), a and b belt paths are rendered in blue and red.

The bed level corrector is used in-line with z-axis motion, and also reads xy states, using those to interpolate through a bed correction map and add the offset correction to the stream of z points. See Section 2.7.3 for a detailed look at how the leveller is operated and integrated.

2.5.4.9 Spline Interpolator

Normally the terminal end of a MAXL system, the spline interpolator follows the logic outlined in Section 2.5.3 to generate interpolated $p(t), v(t), a(t)$ and $j(t)$ values. In this thesis, these are used by the motor controller in Section 4.4 as position and velocity control inputs.

Notably the interpolated spline outputs are not connected to lower layer hardware drivers hardware using Pipes, instead they are integrated directly within firmwares. This represents a performance limit in the system: interpolated outputs often need to be sampled at upwards of 15 kHz, which is too fast for OSAP's internal runtime to process even between two ports in the same runtime.

Listing 2.27: MAXL's implementation for Scara kinematics, which I developed with Miana Smith.

```
@pipes_class_implementer
class ScaraArm:
    def __init__(self, len_a_shoulder, len_b_distal):
        self.len_a_shoulder = len_a_shoulder
        self.len_b_distal = len_b_distal
        self.p_base = [0,0]

    # returns actuator angles *in degrees* and intermediate values,
    def cart_to_actu(self, time: int,
            x: float, y: float
            ) -> Tuple[int, float, float, float, float]:
        # the elbow is located based on circle-to-circle intersection,
        inter_a, inter_b = intersect_circles(
            self.p_base[0], self.p_base[1], self.len_a_shoulder,
            x, y, self.len_b_distal
        )
        # ad hoc guard for out-of-bounds values
        if inter_a is None or inter_b is None:
            raise ValueError(F"Singularity @ ({x:.2f}, {y:.2f})")
        # for elbow position, pick the biggest y always,
        p_elbow = inter_a if inter_a[1] > inter_b[1] else inter_b
        # some trigonometry,
        cos_shoulder = (p_elbow[0] - self.p_base[0]) / self.len_a_shoulder
        ang_shoulder = np.rad2deg(np.arccos(cos_shoulder))
        ang_elbow = 360 - np.rad2deg(
            np.arccos((x - p_elbow[0]) / self.len_b_distal)
        )

        return time, ang_shoulder, ang_elbow, p_elbow, cos_shoulder
```

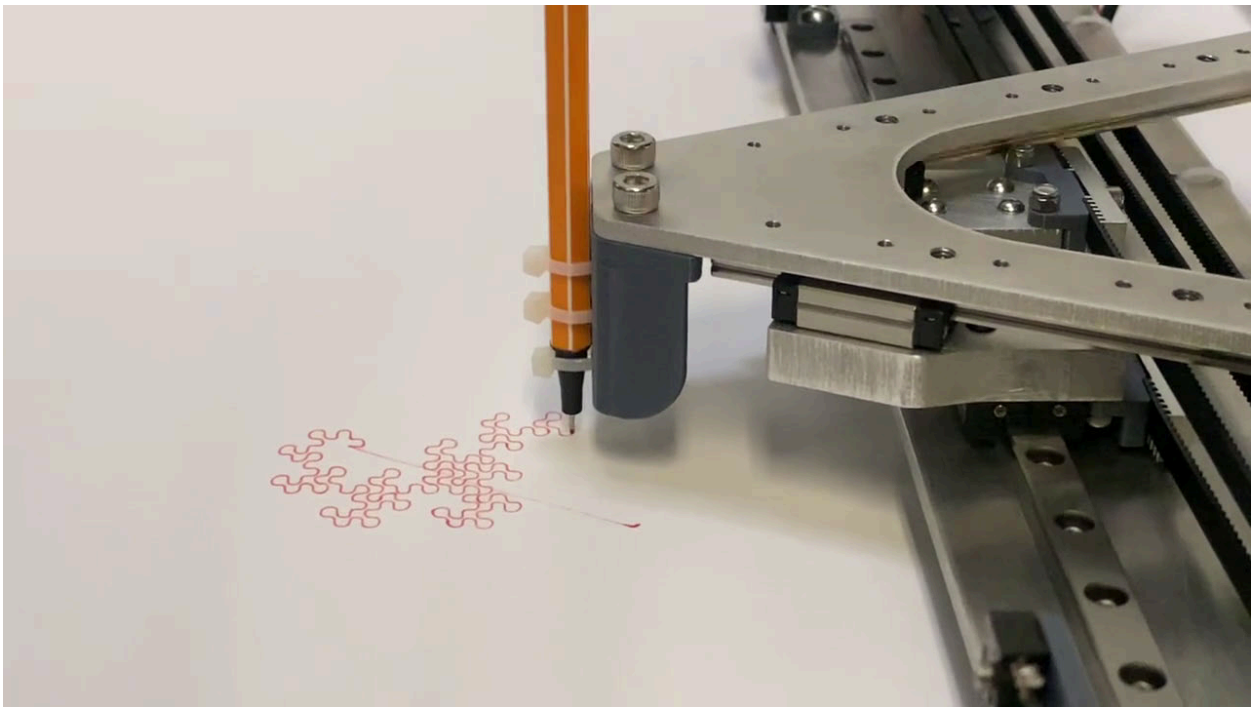


Figure 2.21: Kaos kinematics are similar to those in CoreXY.

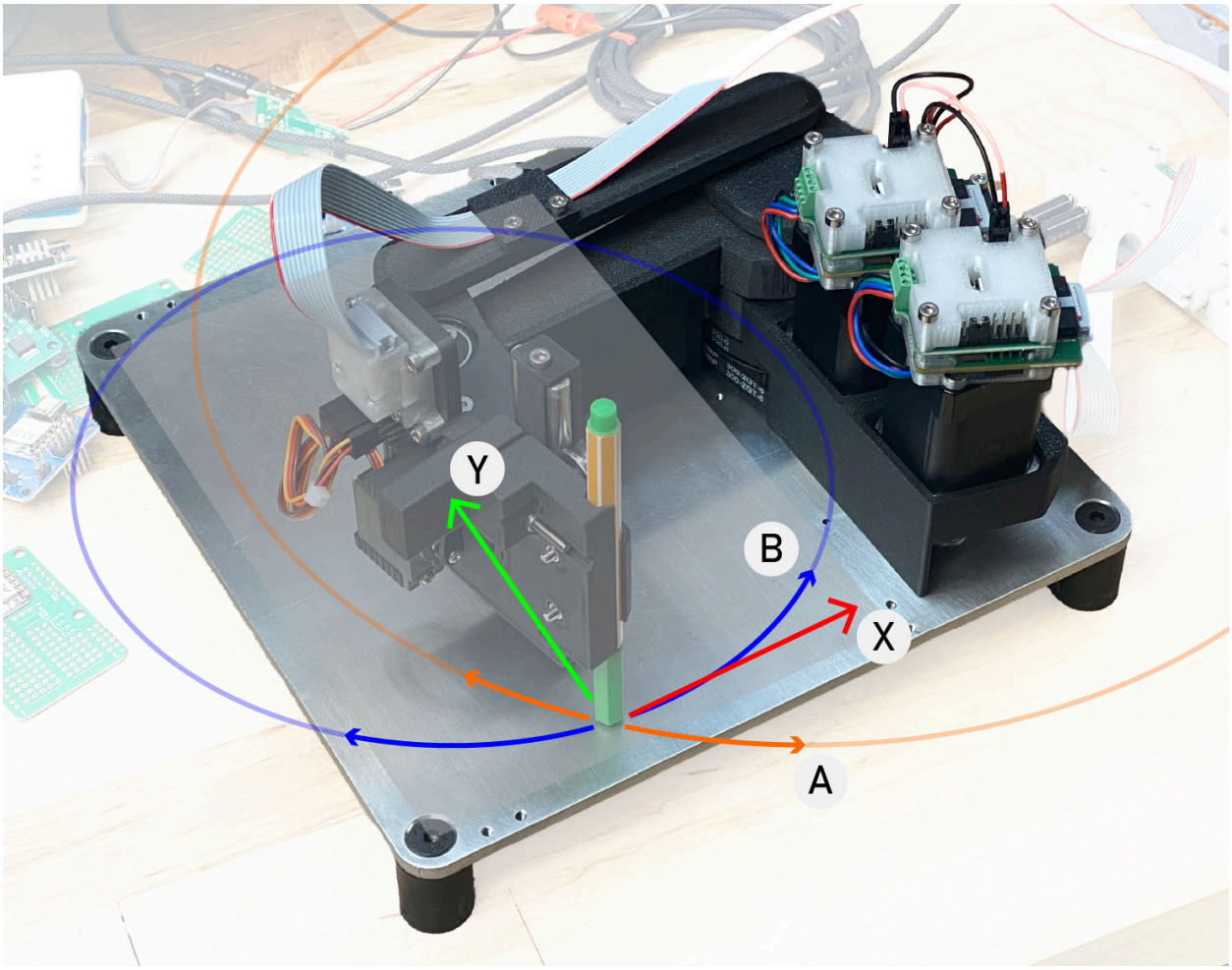


Figure 2.22: The `little guy` machine, and its nonlinear kinematics where motion of the tool-tip in cartesian space is composed of arcs traversed by the machine's two arms, which are connected to one another with rotary joints.

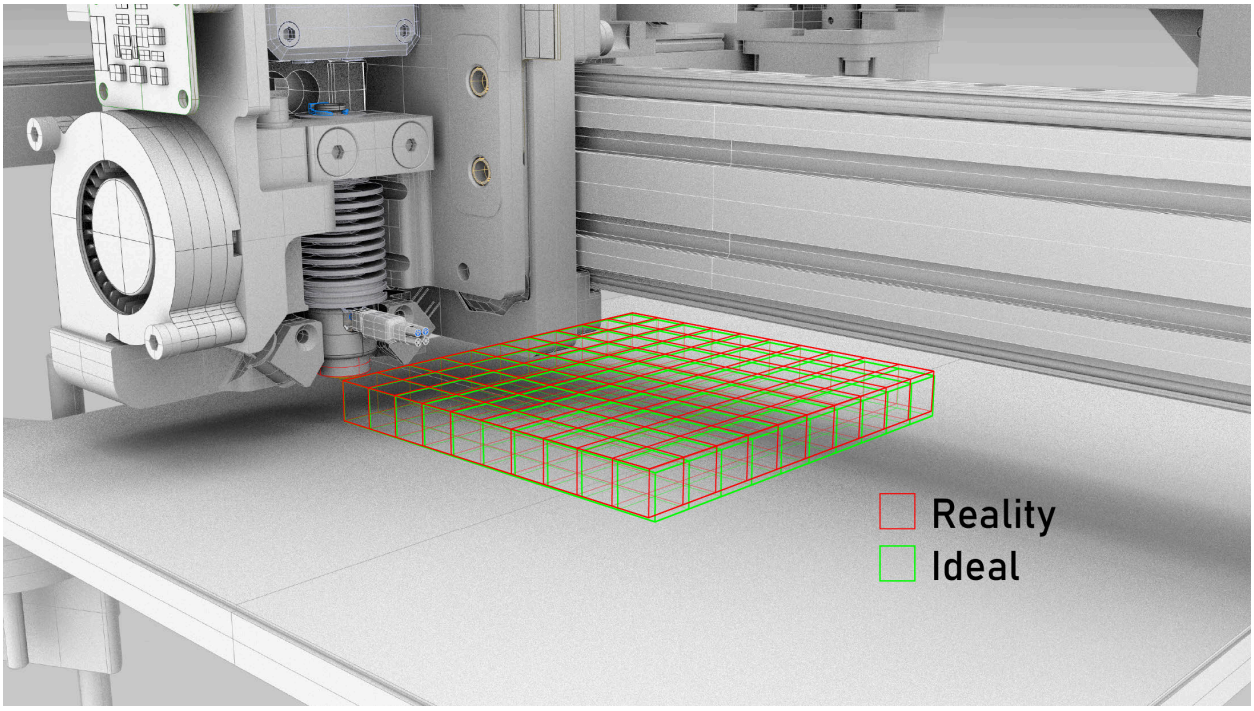


Figure 2.23: While 3D printer performance and reliability depends on their beds being well aligned to their motion systems, this is often not the case. We can apply mathematical corrections to account for these offsets.

2.6 Comparisons to Other Systems Integration Architectures

Now that we have covered background in networks, machine architectures, and also explained how OSAP, PIPES and MAXL all work together, I would like to draw a few clear differences between these architectures and those that are available in the state-of-the-art. First this will focus on the set of distributed architectures themselves, and then on their relation to machine interfaces and applications more directly.

2.6.1 Other Distributed Systems and PIPES / OSAP

PIPES and OSAP combine idea from many of these pieces of work but eschew a lot of the internal steps. These are design choices and are opinionated towards the particular set of constraints that I found while carrying out the other work in this thesis. While I make a clear architectural comparison between my systems and those surveyed above here, evaluating them rigorously against other distributed systems architectures more broadly is not the point of this thesis. Instead I show in Section 2.7 how they are uniquely suited to the challenge of "replacing" GCode with a software-defined alternative, and then extending into applications for model-based control. In 2.8, I relay what this particular set of trade-offs can teach us about systems integration tooling in a broader context. In particular, 2.8.5 discusses how the unique architectural arrangements of OSAP, PIPES and MAXL stack up such that distributed systems can be designed against their network constraints.

2.6.1.1 Function-Defined Interfaces, Typing, and Discovery

> PIPES is *code-first;* device APIs define data schemas and these are sourced from devices into proxies so that systems can be designed using those APIS. Most middlewares are *schema-first;* systems-level data schemas are designed in Interface Description Languages (IDLs) and then device programs are built to interact with those.

OMG-DDS, ZeroMW and most of the serialization tools from Section 2.2.3.3 (ProtoBuf, Capn' Proto) rely on centralized definitions of structs that are compiled at a systems level into a common set of serialized representations "on-the-wire." This is known as *schema-first* or middle-out systems design: interfaces are designed ahead of time at the middleware layer and then software modules are written to adhere to those schema. The approach works well when systems authors have complete control over the contents of their systems or have complete documentation of the schema used by imported devices, but it can lead to misconfigurations if documentation are not available, are not followed properly or if parts of the system are updated without reflecting those changes in other components. These schemes also have limited discovery.

PIPES is of the *code-first* opinion. It relies on a unified set of core data types, but uses the host languages' own typing system to define the schema that is required for each function (see

Section 2.4.2 and 2.4.3). So, rather than specifying exactly how each device should interact with the system (either designing or trying to anticipate system requirements first and then configuring device software to suit) module authors can write generic functional APIs in the same style as they would were they to expose their module as a part of a software library. Those are then automatically wrapped into network-capable interfaces via the tools that I describe in Section 2.4.2. This is more akin to how modules in open source software commons like NPM and PIP[63] are exposed to other programmers: designing good APIs is a well understood practice [177], [178] whereas designing network interfaces is less common. This is a relevant note with regards to later discussion on open source ecosystems (in Section 8.3, and alluded to in 1.4.1).

These types of tools are not completely novel; they are known as *metaprogramming* utilities and Capn' Proto [179] includes one, others include `Cista++` [180] and `Serde` in the Rust language [181]. The normal issue with these approaches is that the schemas are still required for interoperation: once we have our modules connected to the system, other participants still need to know how they should interact with them. PIPES uses proxies to solve this problem (in 2.4.4.2). Instead of generating schemas from remote *data structures* and then programming against those schemas, we write an intermediate schema (the function signature) and write interface functions — proxies — that serve as stand-ins for their remote counterparts. Those are typed against the user language's native typing system. They can be called directly (generating remote procedure calls) when we are scripting, or used as handles to instruct PIPES' manage to make connections between two functions (2.4.4.3).

The real underlying difference here is that PIPES devices and runtimes do not start up with any assumption about how they will interact with the larger system, their network configuration is initially stateless. Rather than configuring their internals against a data-based representation of the global system (an IDL), they each contain a small IDL that describes their local capabilities. Those are collected and then wired together at runtime to produce systems configurations, which I will elaborate on in Section 2.6.1.4.

2.6.1.2 Not Middleware, Underware

> OSAP and PIPES align operating systems, networks, and program configurations and integrate more directly with embedded devices than state-of-the-art distributed systems. They do so by combining runtime-level ideas from very low-level systems like SpaceWire with system-scale ideas from very large systems (datacenters).

In each of the systems I surveyed in the background section on distributed programming models 2.2.3, software components are exposed to distributed systems via middlewares. Those run *within* software components and rely on existing operating systems to interface with other parts of the distributed program via OS-level networking drivers. This means that each is limited to network technologies that are suited to operating systems and they struggle to "cross the gap" between OS-

[63]NPM, the Node Package Manage and PIP, the Package Installer for Python, are registries of software modules that can be used by programmers to rapidly build new applications.

level and embedded computing. For example MQTT relies on TCP/IP exclusively and ZeroMQ relies on Unix Sockets. DDS is more flexible, it technically runs on the "Real-Time Publish-Subscribe" protocol (RTPS, also from OMG) but RTPS is itself based on UDP/IP (and ROS runs on DDS…).

Those are relatively heavy protocols compared to the ones that are more commonly available in embedded devices that actually connect directly to hardware; larger microcontrollers can include IP-capable networking hardware but those are less common and more expensive than simpler link layers like UART, SPI or CAN bus. Middlewares also have lots of internal complexity for brokering and managing messages, buffers, topics, and Quality of Service (QoS); they themselves require tens of megabytes of RAM and relatively high speed processors to operate. This introduces a tension between realtime determinism and high performance computing; fast middlewares need both. For example ROS nodes that control servos directly rely on realtime-patched operating systems to maintain determinacy even though simpler microcontrollers are capable of the same core computing task were it not for the middleware's own overhead [129]. These are often deployed on smaller Single Board Computers (SBCs) that fall in the category of "threshold computers," they are not as powerful as typical OS-level devices but do run an OS, and they are not as deterministic as purely embedded codes but are much *more* deterministic than off-the-shelf operating systems.

To cross the gap into hardware, data from embedded devices are delivered over custom bridges like XRCE-DDS (eXtremely Resource Constrained Environments [123]), this bridge for OPC-UA on Arduino [126], or these RTOS (Realtime Operating Systems) for ROS [124], [125]: these do not run middlewares themselves, they instead provide tools to help build embedded *bridges* into middlewares. This is similar to the device virtualization strategies used by LabView from Section 2.2.3.4 and to FabNet, Klipper and MsgFlo from Section 2.2.4, whereas StepDance (2.2.4.6) is more genuinely distributed across embedded devices. Figure 2.24 below is adapted from [126] Figure 1, [123] Figure 1, and [125] Figure 4 - I add a colour code that approximately correlates to layers in the OSI model.

This is also related to how networks are integrated with circuits at a lower level; fieldbusses and IP-based network stacks are often connected to embedded devices using separate Network Integrated Controllers (NICS), which are subsequently connected to the device over a transparent data link through one of the microcontroller's low-level communication peripherals. This can add some additional delay because data must make its way across this hidden link on trips into and out of the embedded device's memory. In the systems that I developed, I chose primarily to use software defined network controllers - this is slightly lower level than the current scope, but it is discussed alongside the circuits that I developed in Section 3.4.

In OSAP and PIPES, software and networking links at both OS- and embedded-levels run *within* OSAP's runtime (2.3.1, a version of which was written for `cpp` and Python); at the core of each runtime is a list of ports (which connect "up" into software) and a list of link gateways (which connect "out" into network segments). Each of those has an equivalent software interface that has tasks to run: OSAP schedules operation of links directly and can add and remove them and packets delivered to Pipes Functions can drive their operation via network interaction, Pipes Functions can

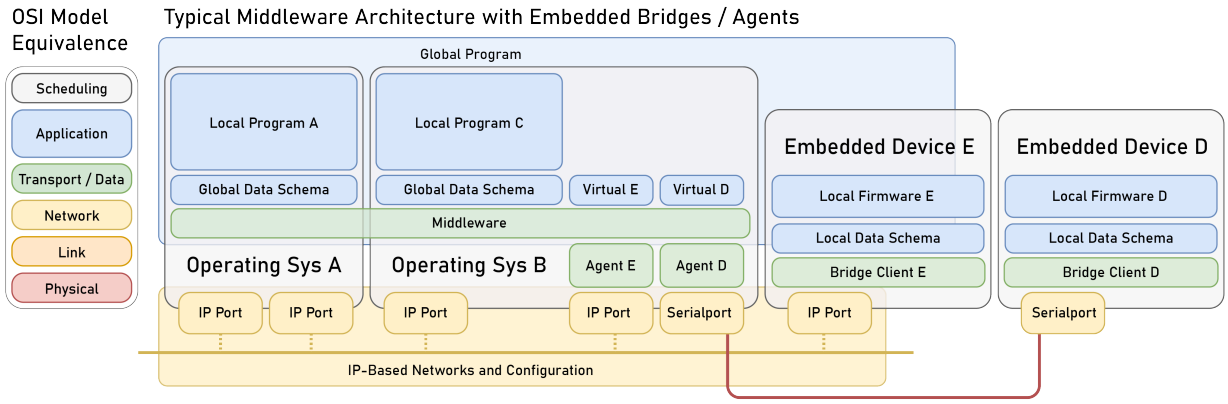


Figure 2.24: To integrate embedded devices into middleware-based distributed systems, a *bridge* architecture is used. Here, embedded devices are represented to the middleware using virtualized interfaces that run a separate networking "agent" which communicates to the device over an out-of-band network link connected to a "client" in the device.

also be remotely scheduled to run at fixed intervals *or* run by device programs directly. I compare this architecture to common middleware patterns in Figure 2.25.

That functionally replaces the OS with our own network-based runtime, so the analogy is that the systems in this chapter constitute *underware* rather than middleware. This is a strange arrangement because in embedded systems OSAP does constitute something like an RTOS[64] but for Python codes OSAP runs *on top of* the pre-existing OS and then within the Python asyncio loop. However, it serves the purpose of causing software execution in both contexts to behave under the same scheduling rules and allows us to add more flexible, software-defined link layers to the network. It is difficult to write custom network drivers for an operating system but relatively easy to write simpler *link layer* drivers in embedded systems. OSAP's runtime can use existing software APIs for network drivers to expose those links by wrapping them in Link Gateways, this includes IP-based network segments. However, this step requires software configuration of those links, see Section 2.3.1.2 for more discussion on how they are managed. OSAP's internal network is also simpler than IP-based networks; it uses source routing (discussed in Section 2.2.2.2) to minimize computing overhead but still provides core services like naming and discovery (2.3.2.1), flow control (2.3.1) and time synchronization (2.3.2.2). I discuss its relation to other source-routed networks in Section 2.6.1.3.

The distributed systems architectural model in the figure (2.25) above is adapted from [117] Figure 1.1. This figure helps to show how middleware-based systems act more like a shared memory layer than as a *network* configuration layer. The real value of different systems integration strategies can often come down to matters of opinion. The shared data model in many ways makes it easier to design systems overall because design of the interface (with an IDL) can be done ahead of time and then individual components can be designed around that layer. In many cases large computer programs are designed around their internal data representations, especially when many developers

[64]Is OSAP an RTOS? See 2.8.3 for discussion on whether it really qualifies for this label.

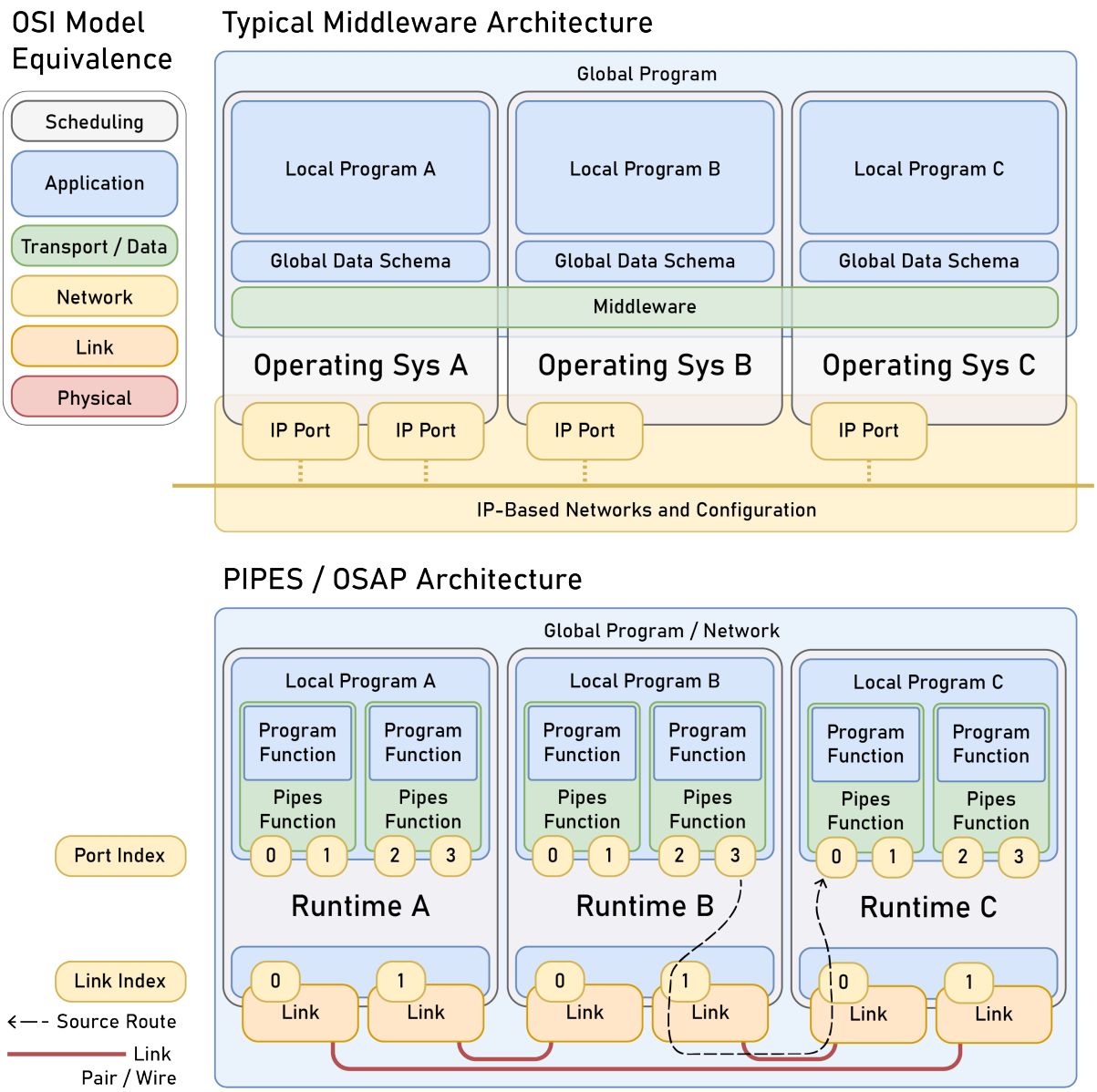


Figure 2.25: A comparison of middleware-based architectures and OSAP / PIPES, with a colour coding for each component's approximate relation to the OSI model; I collapse transport and data / presentation layers here. In middlewares, networking is hidden by the shared data layer whereas OSAP and PIPES expose networking within the program configuration itself using source routes.

must build varying tools that each operate on e.g. the same database. So, IDL-based system representations also *act more like normal computing systems* where operation is fundamentally based on reading and writing data structures in memory. In that sense middlewares follow the software designer's normal ethos, which is to hide lower-level operation in abstracted layers [182]. But this approach also hides network operation: new code is added to the IDL to define quality of service, actual inter-device connections, etc. IP-based networks further obfuscate important networking details because the route that any given packet actually takes through the system is not explicitly

configures. Key aspects of systems performance rely on understanding those configurations at a global level.

I will expand on how this compares to the PIPES / OSAP data model and configuration in Section 2.6.1.4; network paths for program data are configured explicitly. But first a look at source routing and SpaceWire, which is tightly related.

2.6.1.3 Source Routing and SpaceWire

> OSAP is similar to SpaceWire at a network level, with a few key differences around link heterogeneity and how routers are operated on a packet level. SpaceWire also lacks global discovery and configuration.

The standout difference between middleware-based architectures and PIPES / OSAP is the use of source routing. I covered this type of network already in Section 2.2.2.2, the recap is that where packets in IP-based networks are routed based on their addresses (leaving it up to the routers to decide how the network should operate), packets in source-routed networks contain routing instructions: each has a map of instructions and so routers need only to follow these simple instructions in order to deliver the packets. Section 2.3.1.3 describes how this is implemented in OSAP.

In their core networking schemes OSAP is most similar to SpaceWire [100], [171], which uses source routing for very simple and robust networking on orbit and even further away, i.e. in the James Webb Space Telescope [183]. SpaceWire was derived from the IEEE 1355 specification for point-to-point serial interconnects for scalable parallel-processor interconnects and embedded networks [184], that was itself based on the *Transputer,* a microprocessor designed from scratch for parallel computing: each chip had a CPU, memory, and four serial connection links such that they could be wired together to form scalable compute meshes [185]. Today SpaceWire is being extended over fibre links to increase throughput for modern systems with SpaceFibre [186], which also adds more complete QoS configurations and improved time synchronization. Time sync in SpaceWire / SpaceFibre is based on broadcast synchronization pulses, a lower-level mechanism than OSAP's distributed diffusion (from Section 2.3.2.2).

Both use point-to-point links and simple routing instructions, in SpaceWire instructions are a single-byte in length (one byte per port forwarding step) whereas OSAP uses two bytes, which extends the number of port indices available at each step and also uses a few bits to delineate different types of forwarding instructions. The actual routing operating in SpaceWire is also much lower level: they use a technique called *wormhole routing* where programmable gates in the router are physically switched based on network instructions and packets are passed directly through those wires [187]. This is faster and lower latency than OSAP, where links must capture entire packets and then ensure their data integrity before handing them to the runtime where they can be forwarded (this is known as *store-and-forward* routing). Notably, modern Ethernet switches implement *cut-through switching* which also allows the switch to begin forwarding a packet before the last byte of that packet has arrived [188].

SpaceWire / SpaceFibre do accommodate some mixed links, i.e. serialport-type links of varying speeds, Low Voltage Differential Signalling (LVDS) links, and optical fibres, but it is not as flexible as OSAP's software defined link integration; from a use-case perspective it is clear why there are differences here.

Interestingly, SpaceWire also includes provision for destination-based addressing and routers also include tables for this, but they are quite small. Those tables can be configured with a Remote Memory Access Protocol (RMAP, [189], [190]) that is also proposed to be useful for other remote configurations i.e. of instrumentation on the network.

Mostly, configuration of SpaceWire networks is done with manually authored source routes that are configured ahead of time and compiled into device firmwares. This is easy to do when you are building a satellite very carefully over many years in well coordinated engineering teams but not very useful for ad hoc systems development; SpaceWire does not include any remote discovery services — those are too complex to load into the resource-constrained and radiation hardened devices where it is most often deployed (i.e. on FPGAs). OSAP and PIPES include these provisions for global systems configuration, as we will discuss next.

2.6.1.4 Global Configuration for Distributed Dataflow

> PIPES trades operational complexity (which is less than that of other systems) for configuration complexity (which is greater). However, this enables global systems capture and configuration by overlapping programming models with network models.

In a few places above I mentioned the trade-off that source routing networks make: while they make *operation* of the network much simpler, *configuration* of the network becomes more complex. For example to define a data path between two components in PIPES / OSAP, we need to know the whole system graph: how devices are connected over the network and which ports each of our functions lie on. If we were to author these systems without a network and program discovery utilities, we would have to carefully trace both configurations from their source codes and physical network topologies and write those down in device firmwares. This is why I built OSAP's discovery service (Section 2.3.2.1) and PIPES' Systems Object Model (Section 2.4.4.1) and discovery routine for the same. They allow us to write connections in PIPES that map across data schemas and network addresses to write *Pipes,* the virtual wires that articulate how data flows through the system: these connect through program models and network models. Figure 2.4 from Section 2.3.1.3 shows the data structures of two of these Pipes, and explains how they are routed through the network. Section 2.4.4.3 describes how they are configured.

The trade is to add relatively compute-heavy high-level configuration patterns like the Model-View Controller (MVC, [147]; discussed in Section 2.2.3.5) and function discovery services (2.4.2) to very simple low-level operation patterns like source-routing and earliest-deadline scheduling (from [87], discussed in Section 2.2.2.1). These parts of PIPES and OSAP are actually much more complex and consume more program memory, RAM, and compute cycles (when they are running) than

actual program operation. But configuration steps don't happen very often, normally only once when the machine is first configured, or when it is edited.

It also changes the programming model; middleware-based distributed systems route program data according to topic labels, as I mentioned this is akin to building a shared data layer across components. PIPES systems do not explicitly have a shared data model; each device and function defines their own (except for a shared global set of serializations for atomic data types, see Section 2.4.3). To collect and use those schemas, I use functional proxies (Section 2.4.4.2) that are interacted with in high level configuration scripts: remote functions can be called directly using RPC semantics or configured as dataflow components and the entire system, including data routes and network topologies is represented to the programmer in the Systems Object Model. So, where middlewares abstract the actual data paths from applications, in PIPES they are explicitly configured.

This also relates to configurations for Quality of Service. In middlewares, QoS is defined per topic in the shared data model and managed by the middleware. In OSAP, the network layer is used only for transmitting messages under the earliest deadline scheduling technique and QoS is defined via PIPES at the application layer on a per-pipe basis, and can be configured remotely. Individual connections between functions are configured with time-to-live based priorities and transport types to specify whether connections should enforce delivery guarantees or not. This is a much simpler set of QoS configurations than exist in e.g. DDS but could be expanded in the future.

2.6.1.5 Comparisons to Other Hardware Dataflow Systems

> While other hardware dataflow systems surveyed present visual programming interfaces and PIPES is purely code-based, they each require that flows between embedded devices pass through a central broker whereas PIPES is genuinely distributed.

I should relate this quickly to actual dataflow programming environments that I surveyed in Section 2.2.3.4, i.e. LabView, Node-RED and MsgFlo.

LabView and Node-RED run in central interpreters: dataflows are coordinated between software blocks in those interpreters and specialized blocks are described to gather or transmit messages to hardware over bridges in the same manner as the middlewares connect to hardware. PIPES uses *distributed dataflow* - rather than passing messages along a bridge into a virtual device, they can broadcast messages directly into the underyling network and direct them to any data sink in the system. This means that it is possible to configure a pipe between two embedded devices without any oversight from a central runtime, which is a potentially powerful tool for low-level systems configuration.

Another point is that those systems' dataflow configurations cannot be read or modified remotely, meaning that composing a dataflow graph that spans multiple runtimes is not possible or must be done by hand. Nor do their runtimes themselves configure network links to other runtimes, they are standalone. The systems in this chapter work together to produce a global systems model

(Section 2.4.4.1) that combines network configurations from OSAP with program configurations from PIPES, both of which are discovered automatically.

MsgFlo is more similar to OSAP and PIPES: it is more genuinely distributed and devices and their functions can be discovered, but there are still a few important differences. The first is that networking in MsgFlo is fundamentally based on MQTT and IoT and overall network performance is bottlenecked by MQTT brokers which can be quite slow; multiple milliseconds of latency or more depending on traffic and link conditions making it unsuitable for realtime tasks [127]. Even flows that are sent from within one device back to itself must travel *up* and then back *down* from MQTT brokers, whereas each PIPES device runs an OSAP runtime where data passed between two functions within the device is transmitted via pointer ownership (i.e faster even than memory transfer, although data still needs to be serialized and deserialized see 2.3.1). Another performance limit for MsgFlo is in serialization, which is based on JSON. Finally, device capabilities are manually configured with an endpoint descriptor class and messages are not strictly typed, so misconfigurations are possible, whereas PIPES wraps functions automatically and types their inputs and outputs.

Of course the real difference is that these systems actually render dataflow graphs such that they can be interacted with visually, which PIPES does not do. Instead configurations are written as Python scripts. I discuss building a UI for PIPES in the future in 2.9.3.2.

2.6.2 Systems Integration from Related Model-Based Control Research

Moving towards the actual task at hand, it is relevant to relate how the state-of-the-art in distributed systems architecture overlays on the state-of-the-art in model-based machine control. There are two excellent sources in this regard, they are discussed also in the motion control chapter's background section 4.2.8. In that section I also point out that the researchers in both of these cases chose not to modify their machine's existing controllers; the software that they built works *on top* of those controllers via their "feed override" settings, which are high-level handles that set velocity targets rather than updating motor control loops directly.

This represents a limit of closed or difficult to modify systems. In the first case, a commercial machine controller is extended in a multistep bridge that crosses a fieldbus, a digital-to-analog converter (DAC) and coaxial cable, back through an analog-to-digital converter (ADC) that is then bridged into LabVIEW where the author's controller runs. They must also develop a method to *predict* the existing machine controller's own internal dynamics because they are hidden otherwise, again recall the note on the hidden optimization (1.3.3). In the second example, the authors are even using their own custom control architecture but still rely on an embedded bridge and still split state and control representations across that bridge. In Section 4.2.9 I explain how the architectures in this chapter allow me to integrate model-based control more intimately with low level machine controllers. I show the benefits that this entails in Section 4.7 and also in 5.10 where those controllers are extended for 3D printing.

First is Ward's work on CNC machining with model-based control from 2021 [191]. They use a commercial machining center and NC interpolator (the machine's own controller) that is connected to a LabVIEW instance that is running their model-based controller. The interpolator is configured to expose some internal states (commanded and actual motor drive positions and velocities, spindle RPM and spindle current) and accept feed override commands via its internal ProfiBUS network. These are connected to four fast analog input / output modules that convert the ProfiBUS values to voltages (and vise-versa), which are subsequently connected via coaxial cables to a multi-channel digital acquisition device from National Instruments that is subsequently connected to the author's controller in LabVIEW, which was discussed above. Ward reports that readings are taken at 250 Hz (the same rate as the interpolator's own position control loop) and feed overrides are updated at 100 Hz.

Then we have Xin Tong's work on a similar system from 2025 [192], this runs in an architecture that is developed by the authors in an earlier paper (2020, [193]). The layout is similar but avoids the analog input/output step; the machine's interpolator is connected to its drive motors via EtherCAT and this is subsequently connected to Beckhoff's ADS (Automation Device Specification) system, a PC-based automation platform that bridges to EtherCAT over IP [194]. Feed rate override commands are sent back over the same bridge into the interpolator, and the system input-output loop runs at 100 Hz overall. I compare these systems performance to those that I develop in Section 2.7.9.

2.6.3 Comparisons to Other Machine Control Architectures

> MAXL / PIPES is one step in a lineage of modular motion control frameworks, but is the first to use software-reconfigurable dataflow to describe motion controllers, the first to automatically reconcile firmware and network configurations against software specifications, and is unique in the way that it combines dataflow and scripting. Overall the framework is uniquely suited to enable the rapid development and deployment of model-based controllers across machines and processes.

OK, that was an extensive look through architectural differences in networks, distributed systems and programming models. Here I want to compare the actual *motion* control architecture that I develop in MAXL / PIPES to other machine-specific control architectures. The next chapter will begin with a summary of overall architectural deltas and an explanation of how they relate to the results from this chapter (Section 2.7.1).

In terms of control partitioning, this work is most similar to OOH and Klipper: most machine control and configuration is located in a Python script and remote devices follow simpler instructions. In MAXL / PIPES, motion control graphs are mostly configured in Python and then broadcast basis splines to motors while consuming data from remote sensors. The big difference is that machine configurations are made using graphs of dataflow blocks, time synchronization is done explicitly in the network layer, and I use basis splines rather than lower-level representations

for motion. StepDance is also configured using dataflow blocks, but does so in firmware and doesn't provide tools for remote capture of distributed systems.

Configuration of each of these systems is also nicely bundled in one easy-to-edit python program, but another key difference is that PIPES' configurations are automatically compared to underlying hardware data schema to ascertain whether or not these configurations are valid or possible given the currently connected hardware. Additionally, those remote devices' schemas are generated directly from source code and can be faithfully mirrored at the scripting level using automatically authored proxies, whereas none of the other machine architectures that I surveyed here include this functionality. That said it cannot detect actual hardware misconfigurations (like a mislabelled axis), but it still makes substantial improvements over i.e. ROS; as I discussed in Section 2.2.3.2 nearly one third of ROS bugs are data layer misconfigurations [131], [132].

The broader systems architecture in OSAP / PIPES simplifies some lower level configuration steps that are required in other machine systems; OOH and Klipper both require manual mapping between network addresses and controller configurations whereas OSAP's network discovery step is combined with semantic naming schemes in PIPES to automatically find and match network configurations to software configurations: you can re-wire the machine's network without re-defining the control program configuration.

MAXL / PIPES is also the first to combine dataflow with scripting for machine control, whereas other systems use one or the other. In my master's thesis [167] I developed a distributed dataflow machine controller that was entirely based on dataflow, it was difficult to operate hardware using this system because every small operation had to be authored using dataflow blocks even though high level machine operation is often sequential (do this, then do that…). Being able to operate dataflow blocks — and their virtual representations as proxies — in scripting routines is extremely useful, as I will describe in Section 2.7. At the same time, being able to describe static dataflow configurations for realtime control is valuable because it allows us to change remote components of the system without orchestrating that operation from within an operating system. The clearest example of this is in data collection, where we configure remote functions to continuously transmit data at fixed time intervals (2.7.7).

This data collection capability is also missing from most other machine architectures, or where it is possible it is integrated directly within GCode interpreters themselves rather than being exposed as a more flexible configuration step. In Section 2.6.2 I covered two examples of state-of-the-art methods where machine controllers were extended to include realtime data collection, those both develop ad hoc systems that interoperate but do not directly integrate with machine controllers themselves whereas PIPES allows these configurations to appear in the same programming model where the controller itself is built. In Chapter 4 I show how MAXL / PIPES is extended to integrate these data more intimately with machine control, and then Chapter 5 and Chapter 6 extend those methods to cover process control for 3D printing and process inspection in CNC machining.

Finally, adding dataflow without hiding simpler RPC calls turns out to be invaluable for tasks like model building and low-level controller tuning (I provide examples of each in Section 2.7.4), and for building simpler machine controllers (examples in 2.7.10).

2.7 Deploying OSAP, PIPES and MAXL

2.7.1 Architectural Deltas and Results

I will start this section by summarizing the comparison in 2.6 between the OSAP / PIPES / MAXL stack of architectural contributions and similar state-of-the-art technologies and articulating how those deltas lead to the results in this section. As I mentioned near the end of this chapter's introduction (2.1.3.1), the primary challenge with these systems lies in developing something that is descriptive enough to capture and flexibly configure complex distributed systems but simple enough to be deployed in our smallest components without substantially hindering their performance.

In Section 2.6.1.1 I explained that this work uses a code-first paradigm for interface development and how that differs from most state-of-the-art systems' schema-first style. This difference helps us develop modules that can be deployed in multiple systems because it decouples device development from systems development. Most other distributed systems are defined first at the systems level with an Interface Design Language and devices are subsequently attached to that; if the IDL is modified the devices may need to be updated. Section 2.7.2 shows how I assemble systems from the device layer up and Sections 2.7.5 2.7.10 show how I use this property to re-use devices in multiple machines without modifying device-specific firmwares.

Rather than integrating on top of existing operating systems, OSAP builds a common runtime for high- and low-level devices. Instead of relying on OS-level network drivers it uses APIs to integrate software-defined links and then includes a simple networking protocol (in source routing) to build networks from links. This brings operation of the network back into the purview of the distributed system itself rather than riding on top of IP-based backbones as I described in Section 2.6.1.2. To keep operation of these networks lightweight, I borrow ideas from extremely simple and redundant systems e.g. those used in spacecraft as I related in Section 2.6.1.3.

This enables tighter integration of embedded devices within our systems; Figure 2.24 in 2.6.1.2 shows how bridges are used in the state-of-the-art to integrate embedded devices and Figure 2.25 in the same section shows how embedded runtimes are integrated directly into distributed PIPES / OSAP systems. In Section 2.6.2 I gave some clear examples of how this type of integration is managed in the state-of-the-art; in each of those cases, integration between low-level devices and "smarter" controllers was indirect. In Section 2.7.8 I explain how similar systems in the MAXL / PIPES / OSAP stack are assembled and represented in a single coherent architecture, and Section 2.7.9 analyzes timing performance throughout.

Much of this work is enabled also by time synchronization across levels and a careful separating of time scales between the operating system and embedded devices. Besides enabling us to deploy more powerful compute devices for core planning tasks, moving trajectory planner into the operating system helps to make them more transparent. Doing so was posed as a problem in

earlier discussion of the *hidden optimization* 1.3.3; in Sections 2.7.6 and 2.7.7 I describe results that emerge from that capability.

The low-level networking scheme and code-first data schema would make for difficult system set up *if* we did not subsequently include systems for remote discovery and configuration: here we can borrow ideas from much larger distributed systems like datacenters e.g., the model-view controller, distributed clock synchronization and software-defined networking, those were covered in Section 2.2.3.5. These tools are used throughout systems in the thesis. With the additional step taken by MAXL to develop motion control using a composable set of dataflow blocks, we can re-use devices across machine projects (2.7.5), re-use software and control modules and rapidly reconfigure software defined controllers (2.7.4), and re-use core motion control modules across variable kinematics (in Section 2.7.5.1).

Overall this makes for tighter integration across the layers of machine control that GCode is used to separate; it lets us transition from GCodes to just… codes, on networks.

2.7.2 Bootstrapped Systems Assembly

So, a lot of this architectural work was organized around the goal of making the development of new machines more straightforward and flexible. In model-based machine building and in systems assembly more generally we often have the problem that assembly can't happen all at once; we need to build piece-by-piece: components are modelled, configured, and then integrated into subassemblies, those integrate into *other* subassemblies and so on. Being able to do this "smoothly" is important; with e.g. GCode interpreters we cannot do systems integration tests using the GCode interface itself, we have to manage multiple code bases and systems "views" etc. Those interfaces also make it difficult to subsequently reconfigure hardware for spot tests, for example in the 3D printing work here the printer runs intermittently as a printer and as a materials testing machine. Both the printer and the milling machine are used to generate data for their own kinematic model fits, etc. In these cases and in other debugging situations it is valuable to be able to quickly develop new subroutines and systems that run only parts of the hardware.

Here I want to work through the example of my retrofit of the FrankenPrusa (Section 5.4.2), showing how I bootstrap controllers as systems of systems. This is relevant both for the modelling methods that will come in Chapter 4 and Chapter 5, but also to see the results of our new systems integration tools.

2.7.2.1 Capturing the Physical Layer

Firstly we need to outfit our hardware with the embedded devices that we will use to control them. I explain this process in more detail in Section 3.6.2, but I include Figure 2.26 here for a quick example. When I did the Prusa → FrankenPrusa retrofit I was surprised to see how small the set of "terminal" electrical connections to the machine was (top left of Fig. 2.26). This small set of wires is the thin interface that the system senses and controls the machine through.

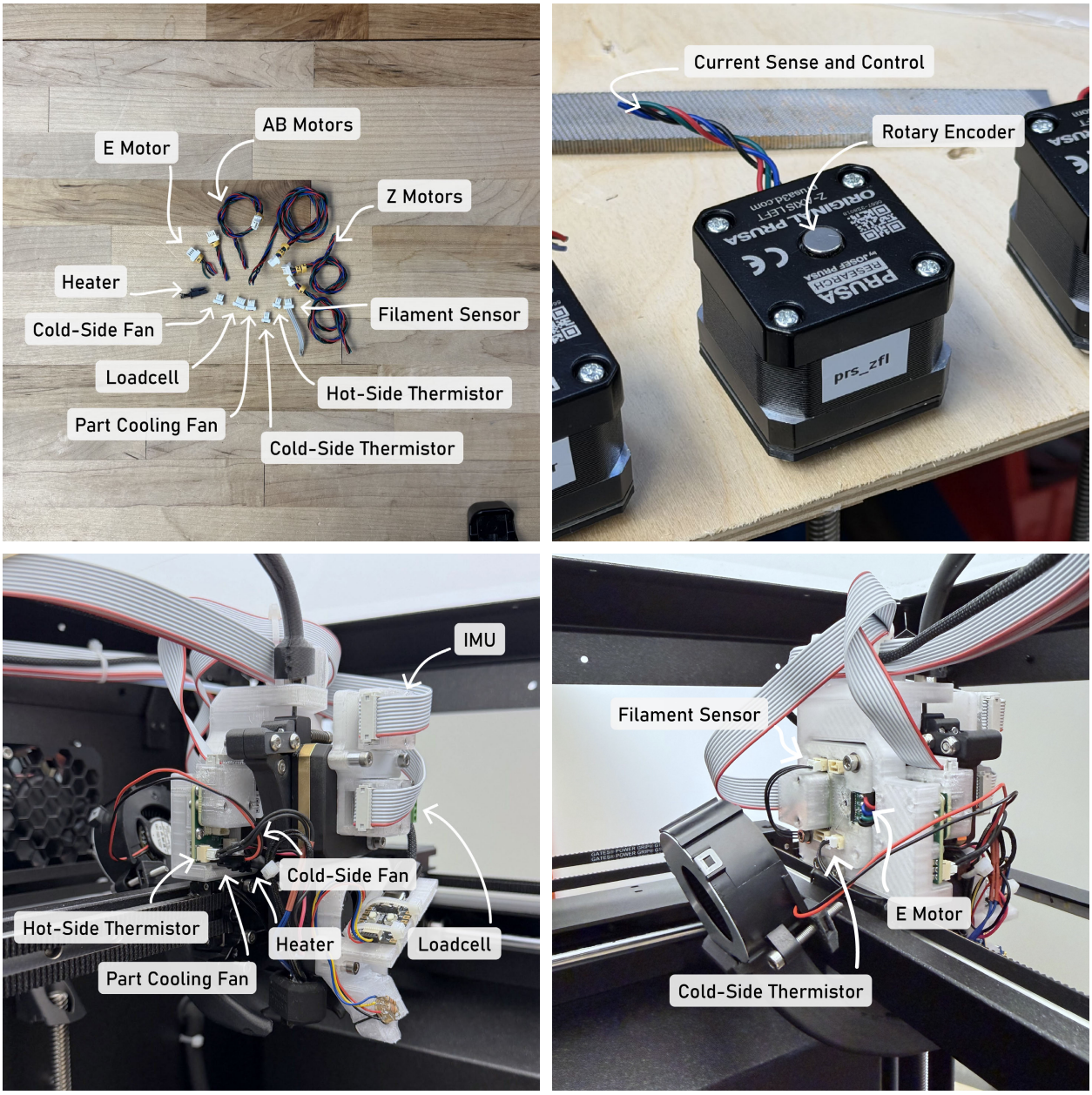


Figure 2.26: Top row: terminal interfaces to the Prusa: all the electrical lines (left) and one of the motors, onto which we attach a diametrically magnetized disk magnet which serves as an interface to our FOC Stepper's encoder (Section 4.4). Second row: the Prusa's hotend with new controllers mounted and hardware interfaces labelled, front and back.

Here I also mount circuits adjacent to the hardware that they control. For the FrankenPrusa this required that I develop some new bracketry. Industrially it is more common to mount control boards in an electrical cabinet (on DIN rails) and build a wiring harness between the cabinet and hardware. The harness from the Prusa appears in Figure 3.24 later on, alongside more discussion on the topic; the short version is that as circuit modules become smaller they approximate the size of the connectors we use to interface to hardware anyway — especially for sensors. In any case

this layer is not required by the computing architecture, it is just the way that I have elected to manage this step.

To each motor I also add a magnet to the back side of the shaft, this is how the motor controller reads rotor position (which is required for closed-loop control). Some more advanced motor controllers can sense rotor angle from electrical signals alone, but I won't get into that here.

2.7.2.2 From Hardware to Software with Firmware

Once devices are mounted, we need to write the firmware that will run within each. At this layer we are effectively writing software interfaces that describe both what our hardware can *do* (by exposing APIs) and what they *are,* i.e. naming them.

I tried overall to maintain just one firmware for each circuit module and then use higher level configuration to assemble them into systems. For devices that have one clear purpose (the motor controller, load cell sensor, and accelerometer) I was successful in this regard. But there are two devices that I developed to be repurposed for multiple tasks where this was more challenging: the H-Bridge module and Deadbugger (see Section 3.2) are both "generic" hardware that I use for multiple tasks. For example the H-Bridge module can be used to drive DC motors, LEDs, or heaters (generic DC loads), but each of those hardware configurations requires slightly different firmware. In the FrankenPrusa I use it to control the nozzle heater, hotend fan, and part cooling fan and so here I load the version of its firmware that is set up for those tasks.

The second step here is in configuring each device's OSAP runtime to attach the link gateways that will be used in the particular system, Listing 2.9 in Section 2.3.1.2 shows how this is done: different gateways can be instantiated according to a compile-time configuration. I will note that we do not need to give the device a network address — those are discovered — but we do need to set up the runtime such that it will load the relevant driver. This is required with the devices

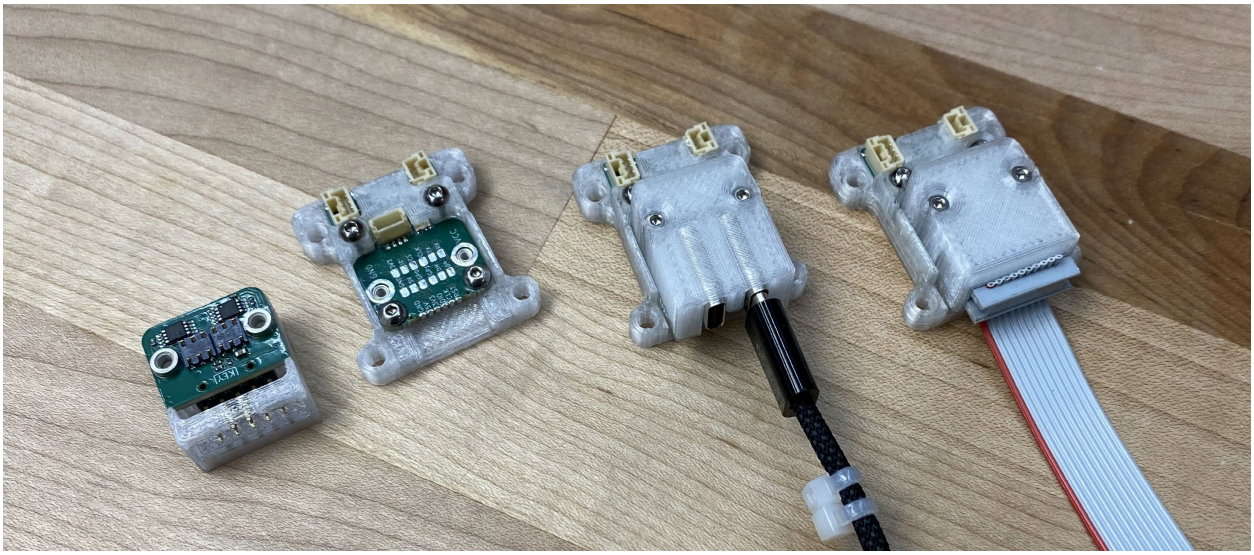


Figure 2.27: This image includes (from left to right) a Backpack module, the HBridge Knuckle with exposed backpack interface, the HBridge with the USB Dually Backpack mounted and finally the HBridge with the UART/RS485 Backpack

that I use because I also built the modules to be physically reconfigurable for different network types with "backpacks" (see Section 3.3 and Figure 2.27). Circuits that are statically configured with multiple network interfaces could simply add multiple gateways to the runtime.

Finally, each device should be named in a manner that is semantically meaningful at a systems level, for example the A and B motors in the FrankenPrusa are named `prs_a` and `prs_b` respectively. These will become software handles in the next step. In Section 2.3.1.1 I described how that is managed in the set up of the OSAP runtime: firmwares can define a static name via a compile-time configuration, but names can also be written remotely and are stored in non-volatile memory. So, it is possible to eschew this firmware configuration and do instead an initial set up step where networks are discovered, default device names are found, and then they are updated using a remote software interface.

2.7.2.3 Making Network Connections

So, by now we have hardware mounted and firmwares configured. Of course, we also connect networks; besides defining which link gateways will be instantiated in each device runtime there is no additional configuration other than physically connecting all the network links and mounting whichever intermediate routing devices or hubs (see Section 3.5) will be used. To be able to isolate certain subsystems later on, I normally organize network topologies according to sensible decompositions in this regard: for example the hub in Figure 3.13 connects to the top half of the RheoPrinter: the nozzle heater, load cell, A, B and extruder motors. When I run material flow tests, I only need to have this device connected to the Python workstation and script. In that figure I have labelled the physical ports according to which device is connected, but this is not a required step: again, network addresses are discovered and reconciled with the semantically meaningful names that I just described at runtime. Certainly it is still useful to label ports for other unforeseen debugging steps, e.g. in the case of physical port failure it is then easier to swap devices between ports to identify which end of the wire is broken.

To connect out of the operating system, we need also to tell the Python script where it should look for devices. This is a similar step to the firmware link gateway configuration. In the OS we can add some additional discovery tooling. For example Listing 2.6 shows a utility that I use to search for connected USB CDC devices that the OS has found, and automatically built new link gateways for each that matches a list of device IDs (a USB-layer identifier that identifies e.g. which type of microcontroller is connected per-port) that match to a list of known device IDs. This tooling (which is software-defined) prevents us from having to reconfigure the runtime every time (for example) the operating system re-assigns logical ports to hardware.

2.7.2.4 Collecting Device Schemas

OK, we have our devices connected to a Python runtime, but we need to know how to program against them. This is where we use the systems discovery routines and the PIPES SOM (Section 2.4.4.1) to automatically build that programming schema. To build those, I developed a generic Python script that wakes up, connects to any devices that are attached to the operating system, scans the network for remote devices, and then collects the SOM and writes *proxies* (Section 2.4.4.2) against it.

This results in one software interface for each type of remote runtime in the system. So for example all motors share the same proxy because their APIs are identical, and then systems can be collected by defining which device name should match to each software object. An example proxy (for the load cell) is in Listing 2.24, and Listing 2.29 shows how I typically assemble those into higher level systems that are described as software objects.

2.7.2.5 Per-Device Calibrations and Models

But developing complete hardware objects for systems is not a required step, it is just a useful abstraction that can be added to manage more complex projects. For example before we can author complete systems, we often need to make smaller subsystems and models. The motor provides two good examples; each requires that we run an encoder calibration routine (see Section 4.4.0.1) and then a motor modelling routine (see Section 4.5.1). For each of these I have standalone Python scripts that connect to one motor only and perform the relevant data collection / fitting routines.

Another good example is load cell calibration, which is a similar standalone script; an abbreviated definition of that is in Listing 2.28. This one is quite simple, I read data from the sensor, filter it slightly, and print it out to the terminal. I then attach weights to the load cell and make notes (with a pen!) for raw readings at each weight. Those go into another script that fits a linear and offset term for the load cell, which I write down in the printer's configuration file. Not everything has to be complex.

I have a litany of other such scripts, for example for tuning heater PIDs, testing networks and synchronization, debugging various devices, etc. In most machines, developing these routines is difficult because GCode does not contain standard codes to operate low-level hardware on a piecewise basis. Some other modular machine architectures do develop these interfaces, like Object Oriented Hardware and even the Duet Firmware (which provides an out-of-band interface, eschewing GCode itself for this step). But those require that software interfaces at a high level are developed to match embedded device APIs, which can lead to misconfigurations.

I invested a deal of time developing the underlying tools that enable the automatic collection and authorship of device proxies. That involved template programming in firmware to read function signatures against source code, automatically building network interfaces against those signatures, serialization routines for "almost any" data type, the network and PIPES discovery routines and graph traversal codes, etc. Those tools turned out to be invaluable, especially for firmware development because they automate the assembly of systems-wide software interfaces.

Although it is common to try to develop subcomponents in a principled way so that systems can be assembled bottom up, in reality we often need to edit low-level interfaces in response to emergent systems integration challenges. This was the case throughout my experience with the machines that I will show in this thesis; I developed firmwares alongside systems, for example realizing that I need to expose a lower level interface for a particular task, or a thinner one to reduce network loads. In the subsection on the closed-loop motor controller (4.4), I show two software interfaces: one is a rich set of debugging values that I used in motor modelling and MAXL debugging steps, and the other is a lighter set that I transmit from actuators while machines run for *machine*

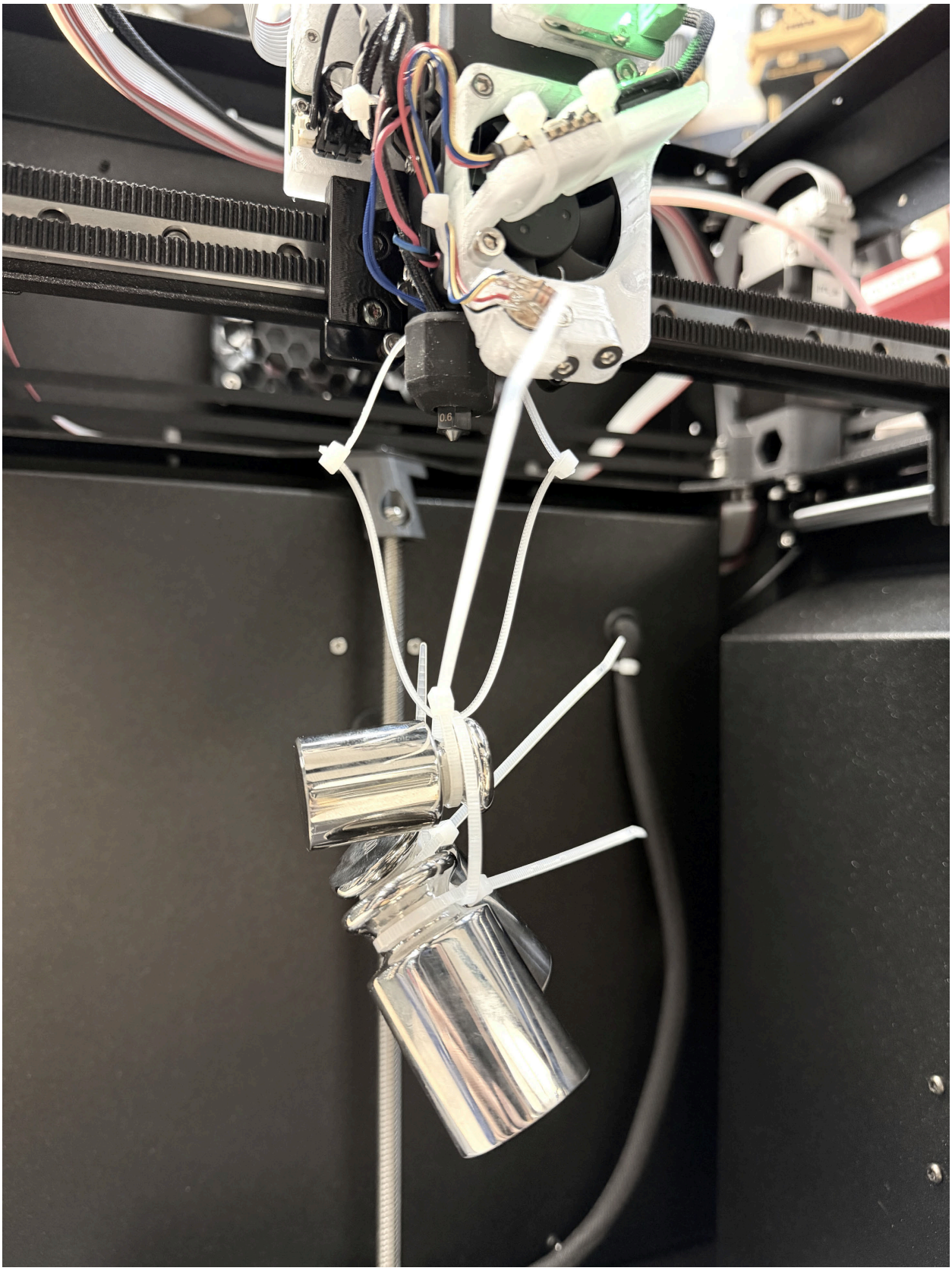


Figure 2.28: Calibrating the load cell in the FrankenPrusa.

Listing 2.28: A calibration utility script for the Load Cell Knuckle from Section 3.2.

```
# in 'run_calibrator_loadcell.py`
# ...
loadcell_name = "prs_ldcl"
# ...
async def main():
    # ...
    osap, manager = await network_tools_setup()
    # ...
    # discover and instantiate the loadcell alone,
    await manager.refresh_system()
    loadcell = LoadcellCs5530D21(manager, loadcell_name)

    # set prior calibration if testing, or ignore if not,
    loadcell_slope = -0.000206289
    loadcell_offset = -12.6785
    await loadcell.set_calibration_params(loadcell_slope, loadcell_offset)

    # just poll and print values,
    while True:
        await asyncio.sleep(0.25)
        stamp, reading = await loadcell.get_raw_loadcell_reading()
        stamp, calibd = await loadcell.get_calibrated_reading()
        r0 = r0 * a0 + reading * (1.0 - a0)
        r1 = r1 * a1 + reading * (1.0 - a1)
        print(F"({calibd:6.3f}): \t{reading}, {r0:.0f} @ 0.95, {r1:.0f} @ 0.99")
```

scale modelling. Instead of carefully writing network interfaces for each of these, I simply wrote new functions in firmware that returned different sets of values that were already present in the controller itself, and then updated proxies to match automatically.

2.7.2.6 Systems and Kinematic Configurations

Once piecewise devices are configured and calibrated (if required), we can start to build higher level systems descriptions. Listing 2.29 shows the `Printer` class that I use for the FrankenPrusa, it bundles devices in that machine into one logical object that can be used across flow tests, motion modelling tests, and to print parts themselves. Listing 2.30 shows how MAXL / PIPES connections can be established (and Pipes Classes / MAXL Blocks instantiated) within that class using the PIPES Manager's functionality (as in Section 2.4.4.3) to define the machine's kinematics.

2.7.2.7 Making and Maintaining Devices, Systems, Processes, and Workflows

Assembling systems-level software objects is roughly the terminal step of the bootstrapping process. At that point we have all the interfaces and calibrations required for subsequent work; in the next sections I will show in some more detail how each of these components are used to make machine models, operate machines, reconfigure them, etc.

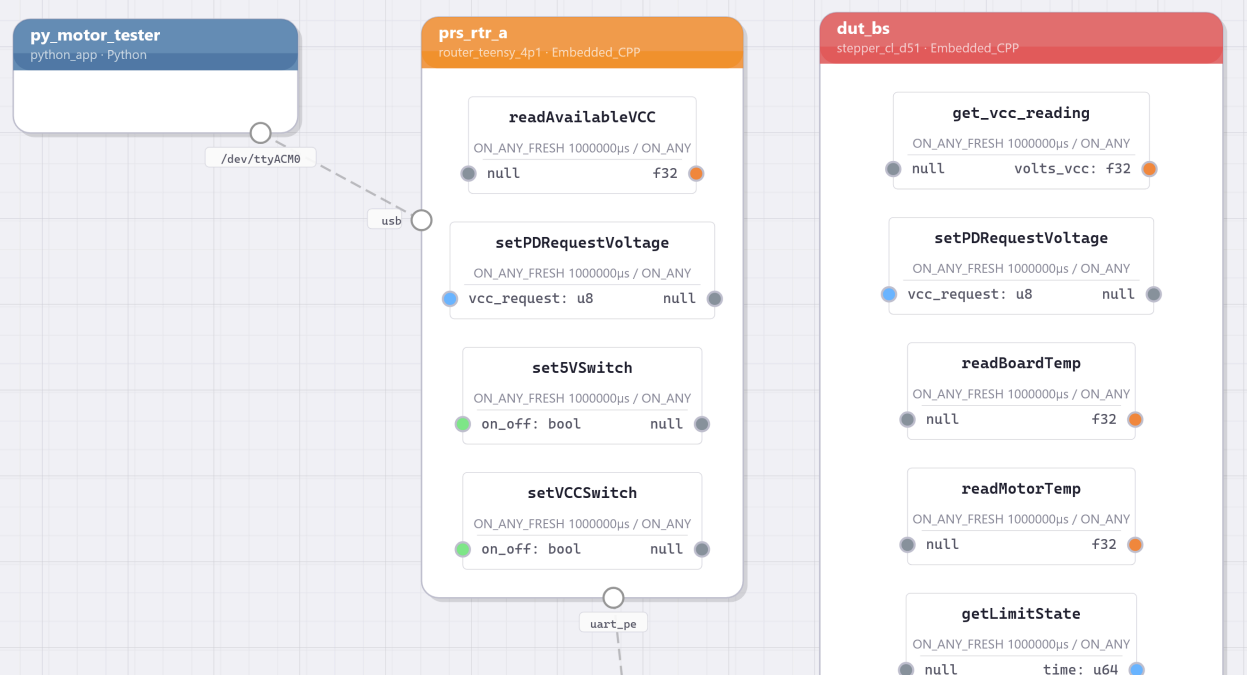


Figure 2.29: For many simple modelling and development tasks, using RPCs alone in simple scripts that contain only one or two devices is often sufficient. The PIPES SOM is still useful in these cases as it contains function signatures that can be used to write proxies for RPCs. Other existing devices can also be useful in these cases; for example the Knuckles hub (Section 3.5, centered in this SOM) contains power management circuitry that can remotely reset or disable devices that are under test.

Overall I hope that it is clear from this smaller set of examples how systems development cycles are improved with MAXL / PIPES and OSAP. I presented this in a bottom-up progression, but in reality development is nonlinear. The tools here are particularly useful in those cases; for example when device firmwares change, new proxies can be quickly written against them. When those are imported into scripts or systems that depend on them, changes to function signatures are reflected in the development environments that we use to author those systems. At the same time if firmwares are updated but proxies have not been changed to reflect that, misconfigurations are discovered during set up steps to alert the programmer that the underlying schema has changed.

In a broader industrial context, it is also unlikely that all of these components (firmwares, networks, systems) will be developed by one team or even one organization. In those cases I think that the clear separation of interfaces also shows clear value. Standalone proxies are probably most valuable to device developers and firmware authors to perform small unit tests or e.g. factory calibrations. It is a bonus that those same interfaces could be used by their customers during systems integration, where the same functions that are perhaps called via RPC during simple tests are then connected to realtime systems via PIPES dataflows. Because each is typed and described in the firmware itself, device manufacturers (or e.g. open source developers) may not even need to provide substantial documentation for their parts; if we can maintain standard protocols for self-describing hardware the world of mechatronic systems could become “plug-and-play,” as I will elaborate on in Section 8.3

Listing 2.29: Once proxies are collected, we can write software defined machine objects that bundle devices to describe systems.

```
# in 'machines/printer.py'
# ...
# this class is written automatically against the PIPES SOM
from proxies.printer.stepper_cl_d51 import StepperClD51
# ...
# this class is authored for the particular system,
class Printer:
    def __init__(self, osap: OSAP,
                 manager: MetaManager,
                 config: PrinterSetupConfig):
        # ...
        # in some cases the class will run partial hardware,
        self.mot_a: StepperClD51 | None = None
        self.mot_b: StepperClD51 | None = None
        # ...
        # instantiate local Pipes Classes / MAXL Blocks,
        self.dof_x = MAXLOneDOF(xy_dofs_max_vel, xy_dofs_max_acc, 1.0)
        self.dof_y = MAXLOneDOF(xy_dofs_max_vel, xy_dofs_max_acc, 1.0)
        self.corexy = MAXLCoreXY(xy_dofs_out_scalar)
        # ...
    # ...
    async def setup_devices(self):
        # ...
        # ensure network map is up-to-date,
        await self.manager.refresh_system()
        # ...
        # now we can get hardware,
        if self.config.use_xy:
            self.mot_a = StepperClD51(self.manager, self.config.motor_a_name)
            self.mot_b = StepperClD51(self.manager, self.config.motor_b_name)
        # ...
    # to check at runtime if devices are present,
    def require_ab_motors(self):
        assert self.mot_a is not None
        assert self.mot_b is not None
        return self.mot_a, self.mot_b
    # ... continued ...
```

2.7.3 From GCodes to Code

Having shown how machine systems are assembled with these new tools, I want to carry on by explaining how these patterns (some of which are new, some of which are essentially Peek and Moyer's Object Oriented Hardware patterns [20], [71]) are used to operate and program machines.

Listing 2.30: Configuration includes instantiating systems-specific Pipes Classes / MAXL Blocks (in Listing 2.29) and connecting them: here internal connections are made between two OneDOF blocks and the CoreXY transform, and then that is connected to hardware devices in the cases where those devices are present. Not all the configuration steps are shown, Figure 2.12 renders more of the complete system that includes e.g. the MAXL Timer block, the printer's model-based velocity planner, data collection blocks, and other hardware.

```
# in 'machines/printer.py'
class Printer:
    # ...
    async def setup_kinematics(self):
        # ...
        # set up corexy block to wait for all new inputs from both gates
        self.corexy.fwds_transform.set_input_mode('all_fresh')
        # xy dofs -> corexy,
        await self.manager.connect(
            self.dof_x.on_new_pt,       # output
            self.corexy.fwds_transform, # input
            "0, 1", "0, 1")             # argument mapping
        await self.manager.connect(
            self.dof_y.on_new_pt,
            self.corexy.fwds_transform,
            "1", "2")
        # ...
        # now do hardware downlink,
        if self.config.use_xy:
            mot_a, mot_b = self.require_ab_motors()
            await self.manager.connect(
                self.corexy.fwds_transform,
                mot_b._maxl_add_control_point_proxy,
                "0, 1", "0, 1")
            await self.manager.connect(
                self.corexy.fwds_transform,
                mot_a._maxl_add_control_point_proxy,
                "0, 2", "0, 1")
    # ...
```

Here I will still be looking "inside" the system, i.e. at a task that is classically run within a GCode interpreter.

This is "Bed Levelling;" as I explained in Section 2.5.4.8, it is important that 3D printers' motion systems are well aligned in-plane with their print beds, but beds are often slightly skewed or warped. To correct these errors we can apply a position-varying offset to the Z axis of the machine according to measurements made of the bed's surface.

This is an excellent example of a capability that is normally difficult to add to machines because interpreters are difficult to extend, as I showed in Section 2.2.1. It involves precise timing between sensors and motion systems (to measure the bed surface), updates to some of the machine's

kinematics, and that we perform some more advanced mathematics (to fit a plane or other offset correcting function to data). It also requires that we develop a new function to perform the routine, so in a GCode interpreter we would have to author all these steps in firmware alongside the rest of the machine controller's functions, mode-switch the machine to run the subroutine, store and fit data, etc — and somehow relay to machine users which `Gxx` string to add to their programs in order to trigger the new code.

Here we do this in a few contained steps. I show in Listing 2.31 how the leveler's correction logic works: the block has `a, b` parameters for the correcting plane, and is connected into the MAXL pipeline through the `on_new_pt` function, which intercepts the machine's `x, y` and `z` positions. It uses the `x, y` position to evaluate the plane function at that point and adds the offset to the existing `z` offset. Because these are encoded in time, any upstream or downstream acceleration control that is applied is outside our concern.

So, part of the block is in the MAXL stream, but it also has "normal" function interfaces that can be used to perform the fitting routine and apply updated parameters. To collect the points for the fit, we need to coordinate machine motion with sensing. Here, time-aligned data and MAXL's blocks are useful again. In Figure 2.11 and Listing 2.20 I show the OneDOF block. The OneDOF block contains logic for homing an axis, as I described in Section 2.5.4.2. That essentially involves running the axis in a velocity control mode until a switch signal appears (via PIPES) to indicate that it has reached a known position (its "endstop"). Bottling this capability into the OneDOF was invaluable for machine development here because we often want to apply homing routines on *either* actuator axes *or* motion axes. For example the CoreXY system in the printer homes in X and then Y, but the actuators are in A, B. In this case the OneDOF block is located before the CoreXY block.

There are four of these in total to control the Z axis of the printer: one of them is located ahead of the bed levelling tool (after the motion planner), and the other three are in-between the levelling tool's output and the individual motors because each of them occasionally needs to be controlled independently. When the bed initially homes to its starting position we need to ensure that they are synchronized in position with one another; all three are homed to their respective endstop simultaneously.

To level the bed, we can repurpose this homing routine to probe Z heights, and we can repurpose the load cell as an endstop switch. I show the block of code for the setup steps here in Figure 2.30, using a figure instead of a listing because I want to highlight a valuable property of the proxy system, which is that code is self documenting. Because each proxy contains type and output argument names, when they are used in modern IDEs (integrated development environments), their APIs can be auto-completed and inspected with e.g. the hover-over pop-up that is shown in that figure.

Also shown in Figure 2.30 is a call to `self.require_loadcell()` which ensures that the device is present before the next steps, and how RPC and graph operation is combined; first we collect

Listing 2.31: Implementation of the Bed Leveler MAXL Block.

```
@pipes_class_implementer
class MAXLBedLevel:
    def __init__(self, a: float, b: float):
        # params
        self.a = a
        self.b = b

    # xyz -> z + correction
    def on_new_pt(
            self, time: int,
            x: float, y: float, z: float) -> Tuple[int, float]:
        # apply planar correction, return (time, z)
        return time, x * self.a + y * self.b + z

    # update the fit,
    def apply_params(self, a: float, b: float):
        self.a = a
        self.b = b

    # calculate a, b params
    def _do_fit(self, pts):
        # reshape points,
        n = len(pts)
        a_mat = np.zeros((n, 3))
        b_mat = np.zeros((n, ))

        a_mat[:, 0] = pts[:, 0]
        a_mat[:, 1] = pts[:, 1]
        a_mat[:, 2] = 1

        b_mat[:] = pts[:, 2]

        # fit using numpy,
        a, b, c = np.linalg.pinv(a_mat) @ b_mat

        # return a, b and the offset,
        soln_pts = pts[:, 0] * a + pts[:, 1] * b + c
        soln_pts = np.column_stack((pts[:, 0], pts[:, 1], soln_pts))
        return a, b, c
```

a number of readings to tare the load cell, then we set up its internal comparator[65] (both using

[65]The load cell's comparator function is a good example of something that could be added on-the-fly to an embedded device if lower-level program reconfiguration were possible, which I explore in Section 2.9.2.1. It is a small software-defined module (so it does not need to be device-specific) that reduces bandwidth requirements for some applications: rather than sending data from a sensor all the time, send it only when a signal reaches a certain (configurable) threshold. It is also something that not every device developer may think needs to be included in

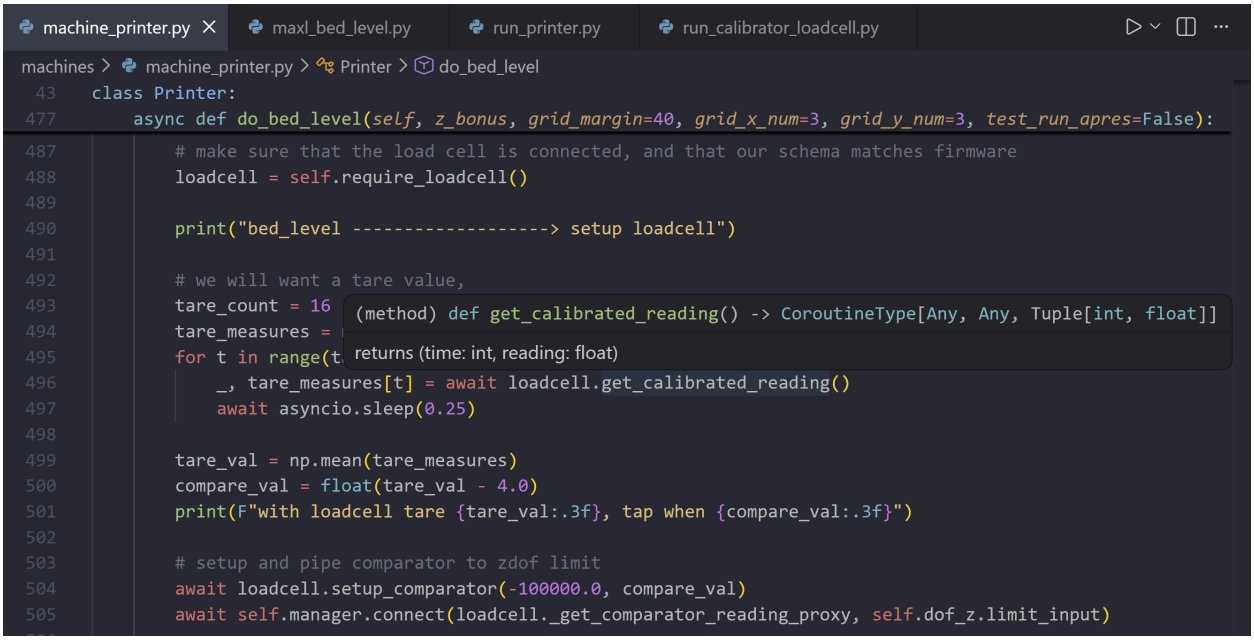


Figure 2.30: A screenshot from my Python IDE where the bed levelling routine is defined. This shows steps to acquire the load cell hardware / software object, tare it, configure its internal comparator, and connect it as a graph component as a virtual endstop.

RPCs), but connect it to the Z axis OneDOF block using `manager.connect()`. This effectively decouples parallel / asynchronous control (the switch state should *trigger* the OneDOF's homing program to stop and turn around) from sequential control (iterating over an array of data points to sample and collecting each, as in Listing 2.32). Once the points are collected, the call is made to the leveller's `._do_fit()` method, and fit parameters are applied. The additional offset (the `c` parameter) is used to update the global height offset in the current machine state.

But I mentioned that this also involves the time synchronization layer. The "MAXL Gap" (which can be tens of milliseconds, see Section 2.5.2) is between the OneDOF block that is generating points and the load cell's readings; so comparator events appear in the past, but we need to know exactly where the Z axis was when the event occurred (not when the signal arrived). This constraint is why bed levelling routines must normally run within a GCode interpreter itself; they rely on centralized timing for this step.

Here, the OneDOF block receives a time-stamped switch state and uses that to query its local history of control points to ascertain where it was when the event occurred, i.e. interpolating along the spline defined by those points. The timing gap is still present and so in the interval between the switch trigger and the homing cycle's update, some milliseconds pass with the machine still moving towards the endstop. In this case that means that the structure is being loaded against itself, which is permissible when the homing velocity is low enough — the motors can also be configured to use less torque in this step. The future work section of this chapter discusses how

their hardware; I only developed it once I need to perform this bed levelling procedure, so I took a loop through the firmware update and proxy collection cycle that I explained in Section 2.7.2.

Listing 2.32: How the bed levelling probe routine is defined in software.

```
# in 'machines/printer.py'
class Printer:
    # ...
    async def do_bed_level(self, ...):
        # ...
        # now we can tap with the zdof home routine,
        async def z_tap(rate, backoff):
            # set stream up,
            await loadcell._get_comparator_reading_proxy.set_input_mode('timer', 250)
            tap_at = await self.dof_z.home(rate, backoff)
            # stop stream,
            await loadcell._get_calibrated_reading_proxy.set_input_mode('all_fresh')
            # return probe measurement,
            return tap_at
        # ...
        # probe at each,
        for p, pt in enumerate(z_pts):
            # go to position,
            self.dof_x.goto_pos(pt[0])
            self.dof_y.goto_pos(pt[1])
            await asyncio.gather(self.dof_x._wait(), self.dof_y._wait())
            # get val here,
            z_at = await z_tap(-1.0, 2.0)
            z_pts[p][2] = z_at
        # ...
```

more of this logic could be re-located to improve this delay. But even in faster networked systems there will always be *some* timing delay between modules, so the approach here remains relevant.

Finally, we have the higher level interface. With GCode we would be writing down a new `Gxx` (although bed levelling itself is now semi-standardized at `G32` [77]) and updating other scripts to reflect that. Here, we can simply add a new function to our `machine.home()` routine, or expose it as its own `machine.do_bed_level()`. Since those are defined in software, inspection and modification of function internals are much more straightforward.

2.7.4 Reconfiguration of Systems for Model-Building and Deployment

Same hardware, new software.

I mentioned in Section 2.7.2 how I use these tools to build systems-scale controllers from smaller components, e.g. with the load cell calibration and with motor modelling and encoder calibration routines. Since model building is so central to the thesis, I want to expand briefly and show

especially the cases where not just individual devices or complete systems are used, but smaller sub-systems that lie somewhere in between those scales.

The most relevant sequence is in the motor— and then motion-modelling workflow. To calibrate motor encoders I use a small MAXL graph combining just a Timer block with a OneDOF block to drive the very lowest level of the motor's controller (its voltage drivers) while sampling encoder readings and motor current measurements (see Section 4.4.0.1).

To model the motor I use a similar script, but now drive the motor directly using RPC calls that set drive current targets directly. I collect the same data from the motor, but now it is operating according to its own dynamics rather than following a MAXL spline. Using that time-series I can recover important motor parameters that relate its current readings to real torque values (see Section 4.5.1).

With a model for each motor, I can assemble machines to measure axes friction. In these tests most of the machine controller is assembled, except for the core velocity planner. Rather than driving it with a planner, I use MAXL's chirp block to oscillate axes. Because kinematics are described separately from the planner itself, this can happen piecewise; for example in the 3D printers from Chapter 5, I can "chirp" the machine by driving its X and Y axes even though the motors are in A, B. I can also define separate tests for the X, Y half of the machine and the Z axes, which are kinematically separated. I describe that fitting process in Section 4.5.2.

This method extends through to process modelling. In all, I use each printer in four total configurations across modelling and operation steps. Each of these run on the same hardware but using different configuration and tasking scripts:

1. To run chirp tests on the XY/AB kinematic system.

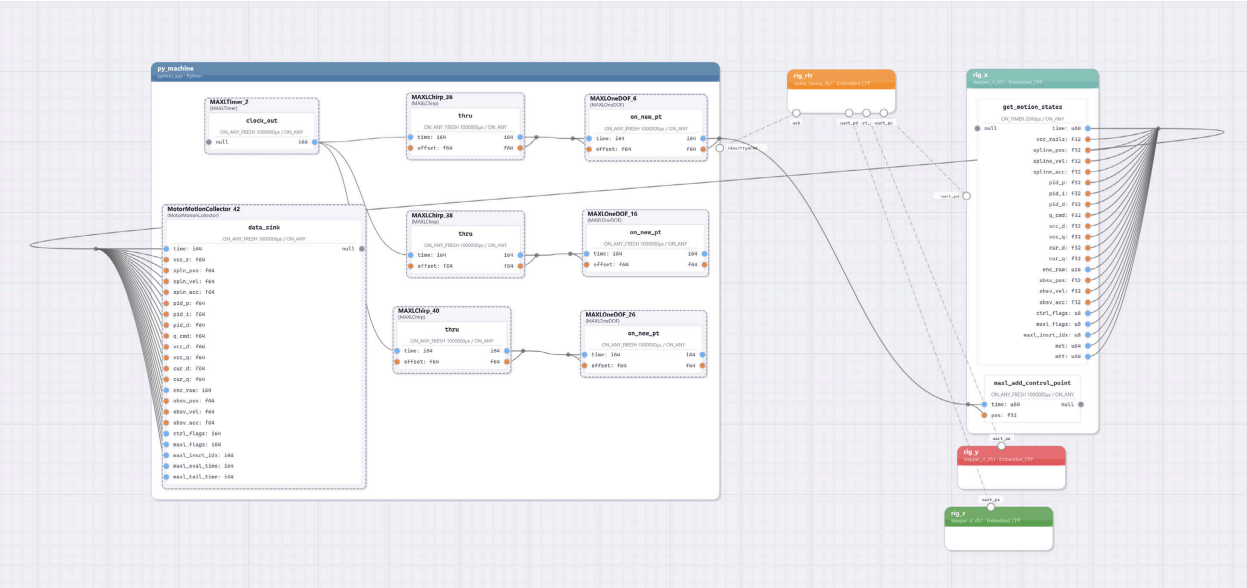


Figure 2.31: The PIPES / MAXL layout of a chirp test. The system is configured to instantiate *chirp* generators in front of each axis. These are used to drive the axes-under-test, and time-series data is collected from the motor (right) into a dataset collector (left).

2. To run chirp tests through the Z motors.
3. To fit flow models, where just the extruder system is activated. In this case, the motion control graph consists only of a Timer, a Chirp, and a OneDOF block (and the extruder motor). All other operation of the system (setting and waiting for nozzle temperatures) is done with RPCs, and data is collected with a series of Pipes: from the load cell, the extruder motor, and the nozzle heater.
4. To operate the machine: using the full gamut of devices and adding the printer's velocity planner interface.

Configurations of the CNC mill from Chapter 6 are similar, and I was able to build the machine and assemble these scripts in a few days because they didn't require that I write any new MAXL blocks or firmwares, just modified scripts:

1. To fit motion models, three configurations — one to test each axis.
2. To run the machine, two configurations: one of which uses a trapezoidal planner, the other uses the optimization based planner from Section 4.6.

PIPES/MAXL also simplifies integration of the optimization based planners that I develop in Section 4.6 and 5.9 — I explain how they are connected to hardware via the frameworks in this chapter in 4.6.2, and Figure 2.32 renders one such PIPES integration (for the 3D printer, another system for the CNC machine is rendered in 4.15).

Because it is supported here by other MAXL blocks, the planner itself doesn't need to manage auxiliary logic for things like axes homing, simpler kinematic transforms like bed levelling (whose effects have negligible ramifications for system dynamics), or any other control states and configurations: which motors are connected and how, their names, etc. Adding these features to the planner itself would be more cumbersome primarily because each implementation of the planner

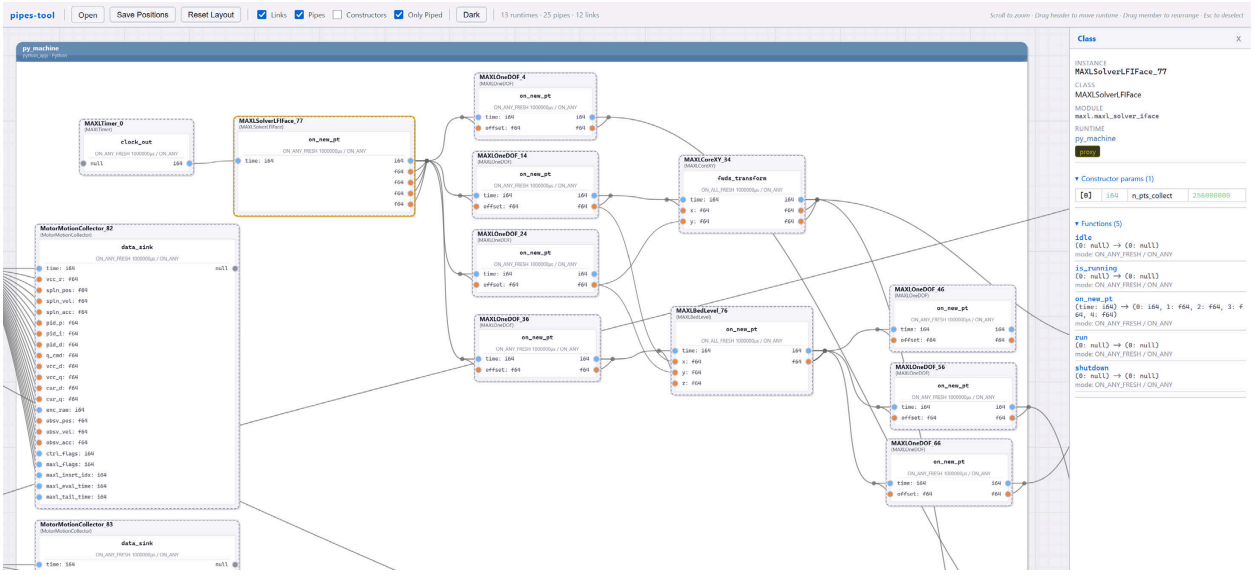


Figure 2.32: The FrankenPrusa's (5.4.2) planner (highlighted in orange) makes up one component in that machine's PIPES graph 2.4. This view is centered on the Python components in the machine, hardware is represented out of frame.

is (partially) unique to the machine it controls. But also because the planner runs in JIT-compiled code on the GPU (see Section 4.6) that is more difficult to inspect than vanilla Python blocks.

It also means that we can configure machines without the planner at all for each of the simpler routines that I just described. Because it is a module, we can swap it out entirely to test different types of velocity planner formulations on the same underlying hardware; I show results from an experiment that does so in Section 4.7.1. In that case the motion models that describe the system are still unifying; both planners make reference (one directly and one indirectly) to the same underlying physics.

Conversely, we can also disconnect the planner from hardware entirely and run it virtually, in which cases integration of "the rest" of the controller (homing, etc.) is not required at all; here we can simply drop those parts. If our models are well enough aligned this is nearly equivalent to running a simulation of the machine.

While the planner is less easily edited and inspected than most Python blocks due to its GPU deployment, it is more easily edited and inspected than the state-of-the-art model-based controllers for robotics that I survey in Section 4.2.7. Those are developed in a number of discrete steps and eventually compiled to run in hardware.

2.7.5 Reconfiguration of Modules for Machine-Building

Same software, new hardware.

So, the section above ran through a gamut of examples where we reconfigured software on top of the same underlying hardware, but I also have some examples of the opposite situation: variable hardware with re-use of software components. This in itself is not novel because most GCode interpreters can be reconfigured for different machines, but the range of machines is worth showing and the reconfiguration strategy is quite different.

In terms of core components, the OneDOF module is repeated by far the most often in different systems and configurations, which makes sense given that most machines have at least one degree of freedom. Its utility in quickly adding simple control interfaces for "the rest" of machine control (e.g. jogging, homing, etc) is invaluable.

This has a limit though, for example the velocity planners for the 3D printer and CNC mill are each unique: they are "reconfigured" for each machine by re-writing software. In kinematics this is true as well, we cannot compose arbitrary mathematics with the existing set of MAXL blocks, and need a new module each time we encounter some new arrangement of axes.

2.7.5.1 Reconfiguration Across Variable Kinematics

When we do encounter new kinematics, authoring new MAXL blocks to suit is easier because they are written in Python and declaring that a generic Python class should be interpreted as a Pipes Class is straightforward: we just add the `@pipes_class_implementer` class decorator, as I showed in Listing 2.20. That again reduces the problem to just the one new class definition. The same is true of many other machine control systems, i.e. Klipper [58], Duet (on the RepRapFirmware

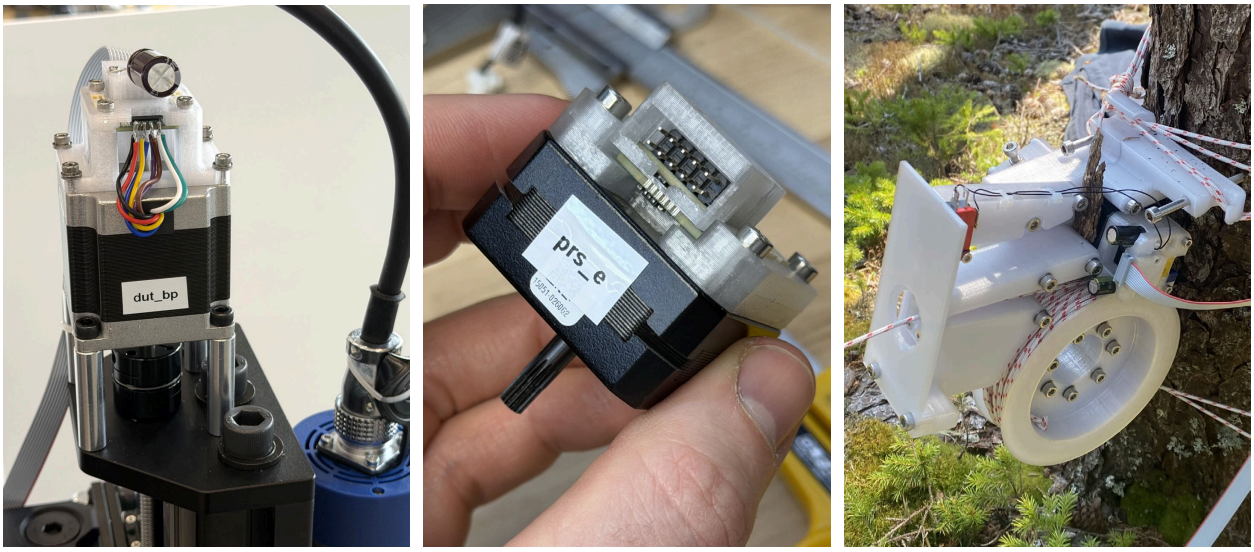


Figure 2.33: Cap modular reconfiguration allows us to use the same hardware (in this figure, a motor driver) to drive many systems: a CNC milling machine (left) 3D printer (middle, the printer's extruder motor), or (at right) in less serious explorations like the Blair Winch project (in Section 2.7.10).

[76]) and Smoothieware [195] can add kinematics in a similar manner, but in each of those cases they are defined in firmware that is then recompiled. StepDance [151] makes great strides in terms of modular kinematic descriptions and actually uses a similar strategy to this one, but is also firmware-defined and provides limited interfaces out of embedded devices (see Section 2.2.4.6).

During MAXL's development I helped to host a workshop at the CBA where participants developed drawing machines. I participated as a machine control enabler and used MAXL and OSAP to quickly develop controllers and kinematics models for each of these machines. That involved working with machine designers to describe their kinematics as functions that map tool-tip positions to motor positions (inverse kinematics or IK functions), and then authoring those together as Python functions. Those were then written as MAXL blocks, and connected to hardware with

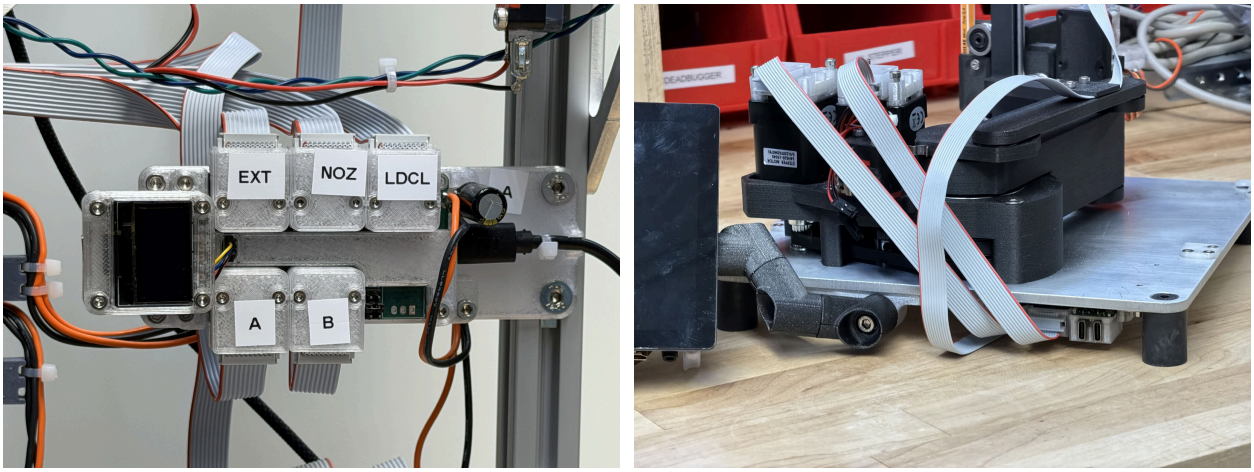


Figure 2.34: Core networking components we re-used as well, this is a hub device deployed in the RheoPrinter (at left) and on the Little Guy (right, integrated on the underside of the machine).

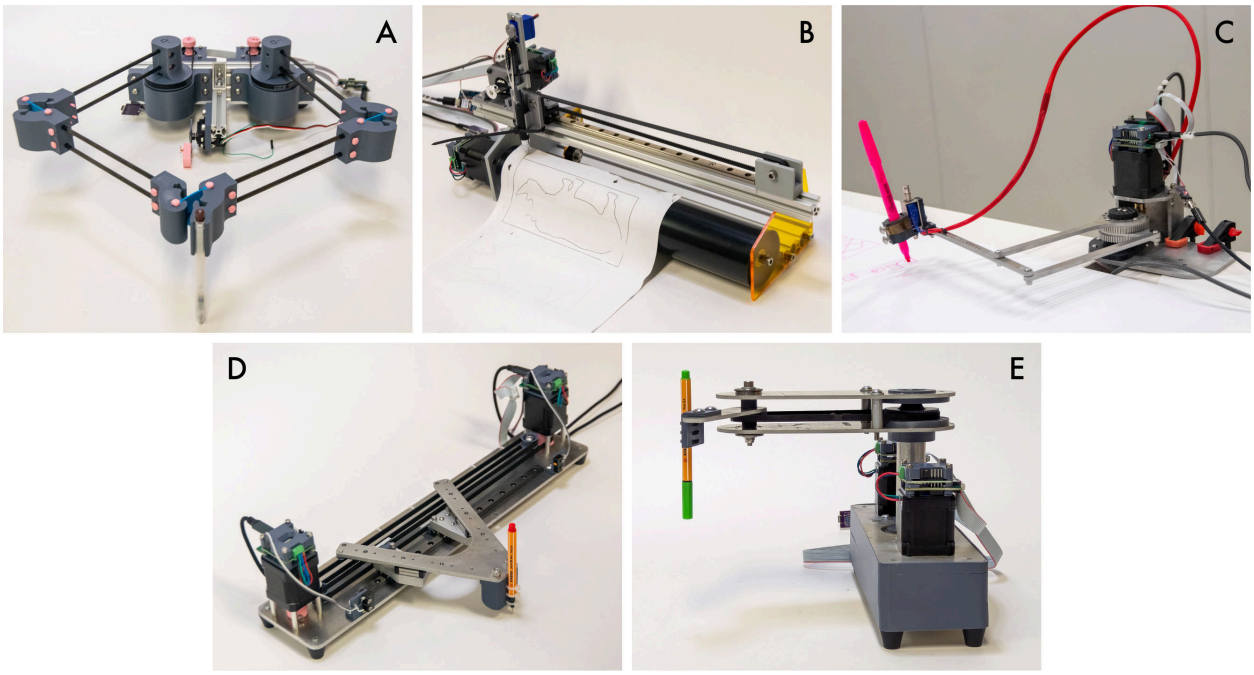


Figure 2.35: A series of plotters developed by participants in a workshop that I co-hosted at the CBA with Ilan Moyer and Leo McElroy. Participants were each experienced machine builders, but not experienced controls engineers. We were able to use MAXL to develop working kinematic models for each of these machines.

an early version of PIPES. We successfully drew `.svg` files with each machine. The machines from the workshop are in Figure 2.35. Two MAXL blocks were developed then that were also re-used or based on re-use; I re-used the Scara kinematics block in the "Little Guy" machine which I built after the workshop (it is pictured in Figure 2.34) and the Kaos kinematics block required only a slight modification to the CoreXY block.

2.7.6 Visibility of Internal Control States

Most velocity planners are in firmware, beneath GCode, so we cannot see the results of their internal operation. MAXL and PIPES go to some length to *remove* velocity controllers from firmware, relocating them as software objects. One of the primary benefits of this is the simple utility of being able to inspect these controllers' outputs and easily modify them. Their configurations and those configurations' relationship to our machine hardware and our target path geometries can combine to produce interesting results.

For the first example of this, see Figure 1.8 from the introduction, where I show that in some parts of path- and machine-parameter space, the feed rates that GCodes specify may never be reached when they are actually run on hardware.

For another example here, we can consider CoreXY ([176] Section 2.5.4.6): machines with this layout have highly anisotropic dynamics. Both motors work together to move the machine in X and in Y, but the moving mass in X (just the end effector) is significantly lower than that in Y

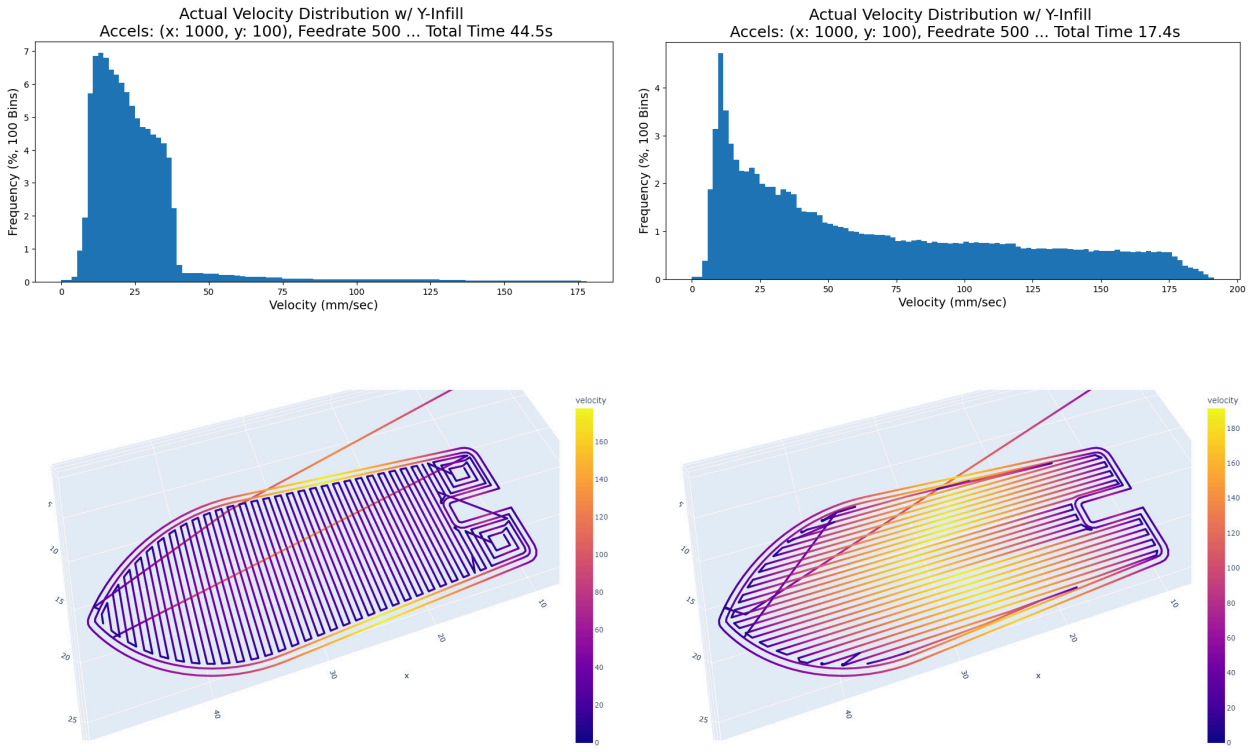


Figure 2.36: On the left, I show a speed histogram and heatmap for a print layer with infill in the Y axis (where it is misaligned with the machine's "preferred" axis for high acceleration), and at the right, the same system with infill aligned on X (where it is aligned).

(which includes the end effector along with the y-beam). In the figure below, I show outputs using a trapezoidal planner which deploys 10x more acceleration in the *X* axis than in the *Y;* we can see that changes to path geometry (like aligning repeated, long line segments to the motion systems' anisotropy) can significantly change outcomes (like total processing time, and actual processing feed rates). Exposing these results to machine users and builders could help them better understand why their machines behave in unexpected ways.

In Section 6.3.1, I show how this result from MAXL's structure enables the development of a simple tool for visualizing chip load deviation in CNC machining toolpaths. In that context, this knowledge can be invaluable before a path plan is run on a machine because significant chip load deviation can lead to chip welding (rather than cutting). The next section describes how this method is extended to more easily map sensor readings back onto trajectories, which is valuable when we want to make process measurements as our machine is running.

2.7.7 Time Synchronized Sensing and Data Collection

Using a time basis for trajectory representation leads to many other outputs in this thesis, because it allows us to combine sensor data with motion data. I show how this capability is used for model building in:

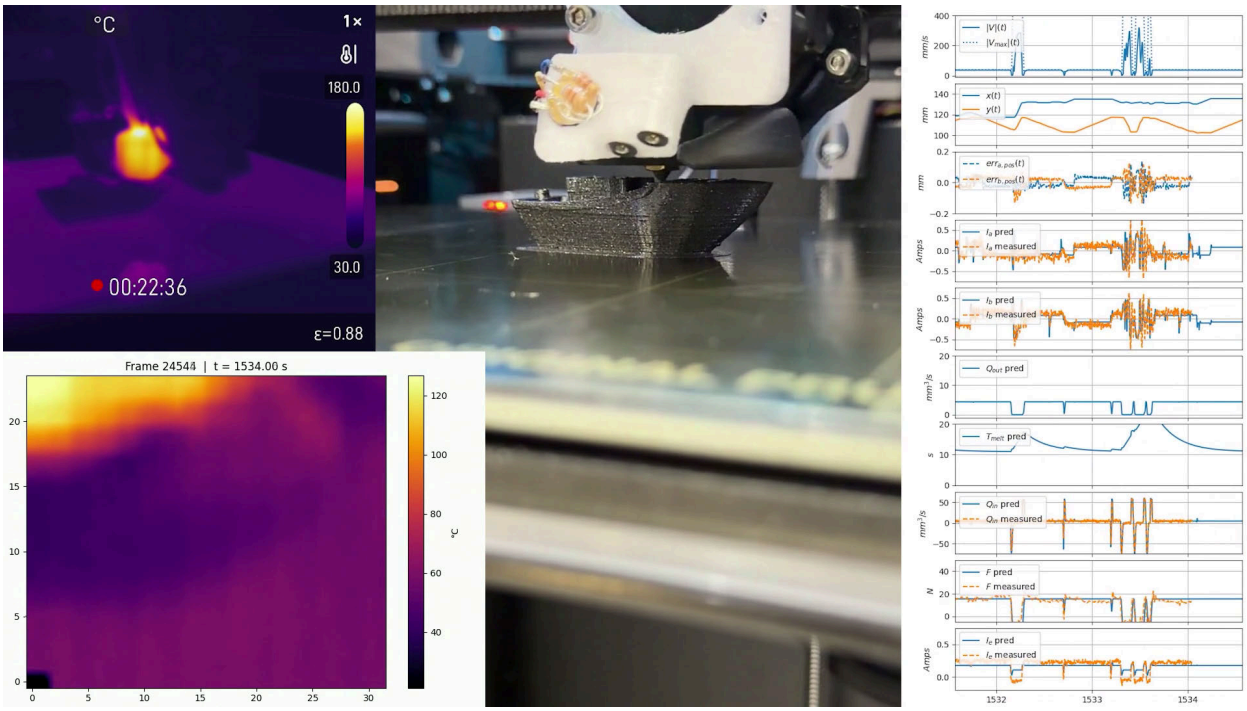


Figure 2.37: Collecting time-synchronized data streams from many sources was a key capability for other work in this thesis. This frame shows (at right) data from the FrankenPrusa's velocity planner (Section 5.9) that is sent to hardware (and stored) with timestamps that are synchronized with the clocks in each of the machine's modules (from Chapter 3). Planner data and predictions are in blue, and sensor readings are in orange. These are additionally synchronized with frames from the printer's microbolometer (a small thermal camera), one frame of this data stream is in the lower-left corner of this figure.

1. Fitting kinematic models, Section 4.5.2
2. Improving kinematic models using data generated during machine operation, Section 4.7.3
3. Fitting dynamic extruder parameters, Section 5.5.3
4. Fitting coupling terms between extruder motors and melt flow models, Section 5.6

In each of those cases, I use motion controller inputs to drive systems while collecting time-series data of motion states alongside sensor data, which are then used to fit (or update) models. I also show how this capability is used to evaluate motion systems:

1. To evaluate the quality of kinematic models, Section 4.7.2
2. To generate new insights from 3D printer data that combines sensor, motion, and planner states, Section 5.11.1
3. To make cutting force estimates, combining kinematic models, motor models, motor data, and trajectory data, Section 6.4.4

Piped connections are useful here because we can configure remote functions to send data back to our configuration and control scripts in a timely manner. That involves making connections up through the network using `manager.connect()` and then configuring each function's input mode to run at a defined interval. Developing data collection in this way means that the Python script itself doesn't have to repeatedly poll devices, which would introduce extra jitter in data samples. The

transmission intervals can also be tuned according to network performance and data resolution priority, and of course sensor readings are all time stamped against their local device's estimate of system time (from Section 2.3.2.2).

2.7.8 Enabling the Application of Large Compute in Machine Control

Connecting high-powered computing systems to machine hardware is often difficult because operating systems can be nondeterministic and networks that span between them introduce lag. I discussed this in some detail in the background section to this chapter, and made direct comparison between this systems architecture and model-based controllers that have been developed by other researchers in Section 2.6.2.

I described in Section 2.5.2 how we can introduce a fixed-size, fixed-interval *timing gap* between these layers to approximate deterministic operation across this layer. This is what allowed me to deploy the GPU-based velocity planner from Chapter 4 and 5, which requires substantial computing power.

However, the method is also somewhat limited. As I will discuss in Section 4.6.2, I had to develop an additional separation between the OSAP runtime that controls the machine in Python and the planner code itself. Although also written in Python, the planner blocks Python's `asyncio` loop while it generates new solutions, which can take up to 100 milliseconds. PIPES is designed with parallelism in mind, but OSAP relies on cooperative scheduling and while the planner is working the rest of the system (that is collecting data and transmitting control points from prior solution components through a PIPES graph), needs to keep running. I discuss how OSAP could be improved to support this more natively in Section 2.9.

Introducing that timing gap does not come for free; the gap is configured against the worst case network and computing delay between the operating system and remote devices. To understand how we can configure the gap intelligently, and to see how much of a gap is really required, we have the next section on timing.

2.7.9 Timing Performance Analysis

I've mentioned in a number of places how network and compute performance can constrain machine performance. The OSAP runtime, PIPES architecture and MAXL framework are each designed with this in mind, favoring simple low-level operation that that can be computed quickly but adding a slower global configuration step to combine that with flexible development. I also discussed the trade-offs between flexibility and overhead; adding layers and abstraction tends to require additional overhead.

In this section I first make a broad timing comparison between the model-based machines that I developed in OSAP / PIPES and MAXL and other researcher's efforts, and then work through packet size overhead, underlying timing performance using measurements from the clock synchro-

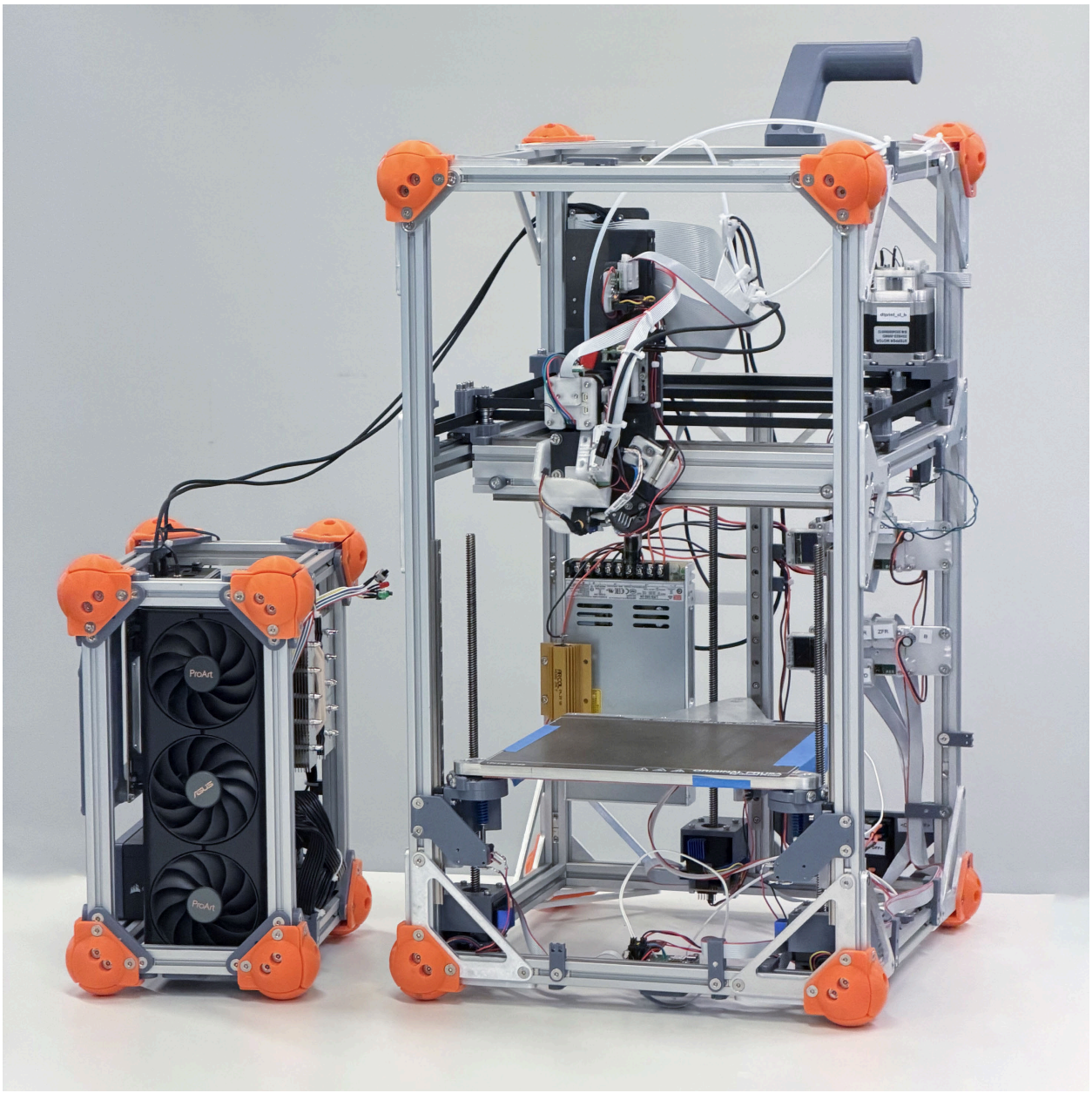


Figure 2.38: How to assemble a controller that spans GPU and CPU, big OS, and embedded devices?

nization tests, and finally relay how those relate to overall bandwidths / performance of these systems.

2.7.9.1 Timing Comparison to Related Model-Based Control Research

In Section 2.6.2 I looked at two other researchers' efforts to deploy model-based control for machine systems (both CNC milling centers). I already discussed how those differ at an integration level, but a timing comparison is worthwhile as well; one of them (Ward [191]) collects readings at 250 Hz and issues new control outputs at 100 Hz and the other (Tong [193]) does both at 100 Hz. Both integrate over heterogeneous networks (including IP-based links, fieldbusses and even analog links)

that are manually configured to bridge between realtime control components and higher power workstation computing where models are deployed.

The controllers that I build with the systems in this chapter collect data from actuators at 500 Hz and transmit new control points at 250 Hz. The actuator data includes more motion derivatives and internal current controller states than Ward and Tong collect, a full list of these is listed in Section 4.4.1. I collect data from the 3D printer load cells at 150 Hz, and temperature sensors at only 30 Hz, see Section 5.4.2. The basis spline control points that I send through MAXL are delivered directly to interpolating motors and can be evaluated for position, velocity, and acceleration values. Ward and Tong's systems both connect model-based controllers to their machine's internal *feed rate override* handle, which adds more and somewhat unpredictable delay between updated commands and real response from hardware; their machine's hidden velocity planner uses the feed rate override as a reference signal that it subsequently tracks. In fact this signal is indirect in another way because it modifies the target feed rate that is written into the *GCode* that their machines and model-based controllers are simultaneously consuming.

I was surprised to see these relatively low bandwidths. It is hard to tell if these authors are limited by their model's complexity and computing (both build substantial control algorithms) or by their networks, network bridges, or by the machine interpolators. It seems as though even if the controller's predictive bandwidth were limiting they would have preferred higher bandwidth input data if it were available to improve measurement resolution. That said, they may have identified that their system's dynamics were not themselves much faster than 100 Hz — these are bigger machines with larger time constants than the ones that I have instrumented.

2.7.9.2 Packet Size Overhead

I've collected a few key messages and calculated packet overhead for each across PIPES and OSAP in Table 2.6. OSAP header bytes (second column), include a baseline of six bytes for each packet's instruction pointer, maximum transmission unit, time to live, and (after the route) the transport protocol type. Headers are variable length because routes can include multiple instructions; one byte for each link gateway traversed, two for each fieldbus gateway traversed, and three bytes to encode the source and destination ports; see Table 2.4 for more details on the OSAP routing system.

The third column counts bytes in the *raw* data that is transmitted in each message, i.e. the size of the message data in CPU memory. PIPES bytes (fourth column) accounts for bytes used to encode the argument mapping (which values will be delivered to which input arguments of the receiving function) and each value carries its own type byte so that the receiving function can check against its signature and convert (or reject) data where necessary; see Table 2.5. For each PIPES message, there is a one-byte key, one-byte count of arguments, and each argument requires two additional bytes (one to map, one to type); so we have $2 + 2n$ bytes of overhead for each where n is the number of arguments. PIPES overhead (fifth row) is the share of application layer bytes that are used for this mapping / typing encoding. There is substantial overhead here, but the encoding removes state from PIPES connections. We may be able to have both, as I briefly discuss in Section 2.9.2.2.

Total overhead is calculated as the percentage of each packet that is used for OSAP and PIPES operation. Small packets along long routes (e.g. in the third row) are the most punishing in this regard, and overhead is smaller for larger packets. For a brief comparison to IP-based networks, the IPv4 packet header is 16 bytes long and IPv6 header is 40 bytes. These are normally wrapped in Ethernet frames for another 38 bytes of overhead.

So, in an IPV4 network over Ethernet the packet in the third row (at 62.5% total overhead with OSAP) would include 83.3% networking overhead. Over IPv6, 87.5% (this is still including the PIPES overhead). At the same time, Ethernet's bitrate is much faster than the simpler embedded networks I use in these examples to compute wire times (which are explained shortly) so, while larger, IP/Ethernet frames would move faster. That said, Ethernet baud rates require Ethernet-capable devices to read them and there is a natural correlation between a microcontroller's processing power and its ability to manage extremely high data rate networking. In Section 3.4 I explain in some more detail why I prefer simpler (slower) link layers for these systems.

Message	OSAP Header	Raw Data	PIPES Map and Types	PIPES Overhead	Total Overhead	Wire Time[e] A	Wire Time[f] B
	Bytes	Bytes	Bytes	%	%	μs	μs
MAXL Control Point[a]	10	12	6	33.3	57.1	224	107
MAXL Control Point[b]	11	12	6	33.3	58.6	232	110
MAXL Control Point[c]	14	12	6	33.3	62.5	256	120
Motor Motion Data[a]	11	42	24	36.4	45.5	616	270
Motor Modelling Data[a]	11	93	46	33.1	38.0	1200	513
Load Cell Data[a]	11	12	6	33.3	58.6	232	110
Heater Data[a]	11	16	8	33.3	54.3	280	130
Time Config Query Response[a,d]	11	65	n/a	n/a	14.5	608	267

Table 2.6: Overhead for packets transmitted through PIPES / OSAP.

[a] Routed across one link gateway.
[b] Routed across two link gateways.
[c] Routed across three link gateways and one fieldbus gateway.
[d] The time configuration query response is made through OSAP's native model-view controller; data is packed according to a pre-defined schema and so there is no PIPES overhead in this case.
[e] At 1000 Kbps.
[f] At 3000 Kbps through a UART link including COBS packet encapsulation and CRC16 error checking.

For example MAXL control points contain two values: a timestamp (a 64 bit / 8 byte wide integer) and a floating point value (32 bits). The motor motion data and motor modelling data sets are in Tables 4.3 and 4.4. The timing configuration packet (last row) is unique to this set in that it is

not part of the PIPES system, it is one of the OSAP model-view controller messages. This is not the message that is used to synchronize clocks, it is the message used to inspect clocks.

I also include two "wire time" values, these show how long it takes to transmit each message on network links under two conditions: the first is a generic 1000 Kbps link and the second is a 3000 Kbps UART link (where each byte adds a start and stop bit) that includes COBS packet delineation and CRC16 error detection; this is the type of link that I used most often to connect devices to one another across OSAP.

2.7.9.3 Timing Analysis with Clock Synchronization Data

So, that covers packet overhead and wire time gives us an estimate of the network layer's underlying performance, but time is also consumed processing messages within the OSAP runtime. Because OSAP uses software defined links, some processing time is also required to e.g. delineate packets and process CRC in each (depending on the link gateway).

To make an estimate of how much time is used in these processing steps, we can look at data from the time synchronization tests (Section 2.3.2.2). Figure 2.39 shows round trip times measured

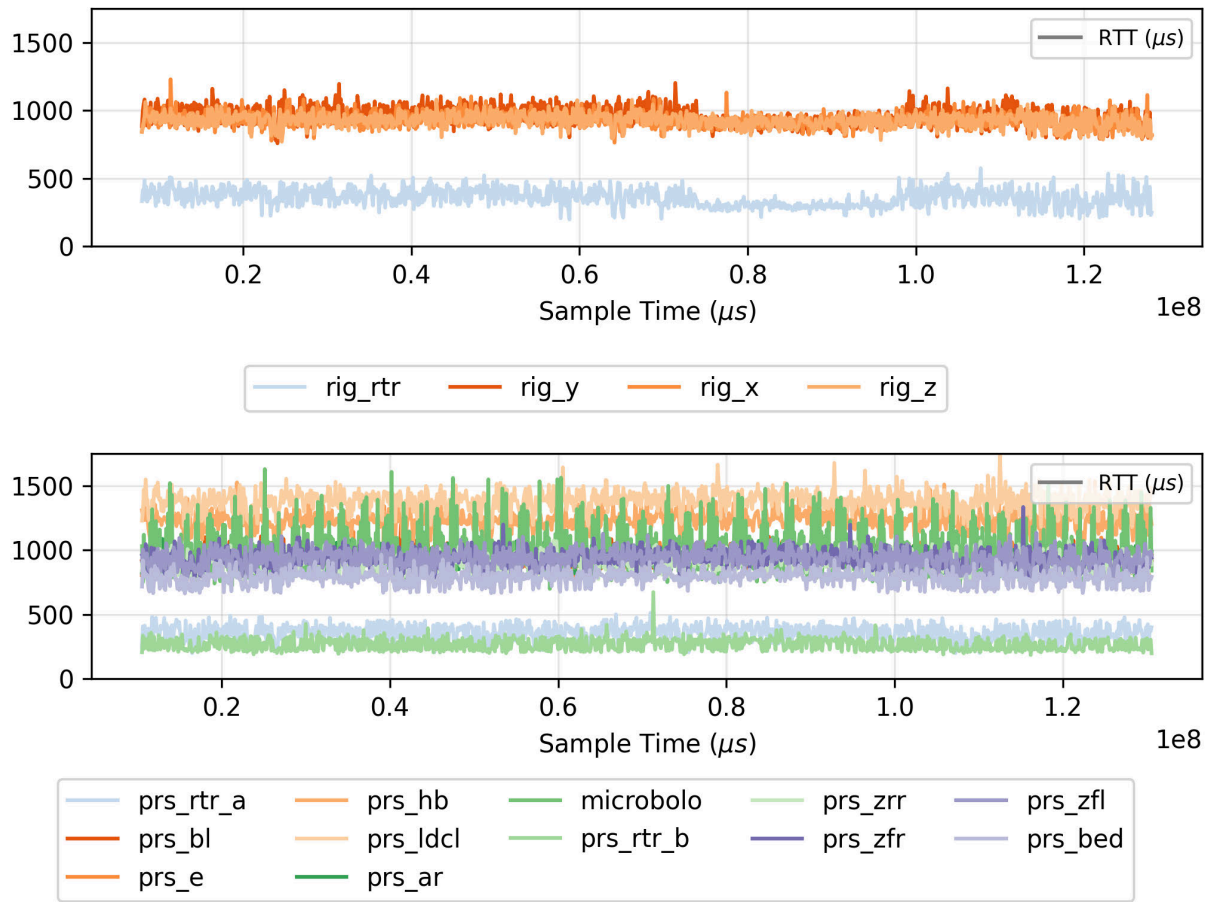


Figure 2.39: Round-trip times for clock configuration request messages between a Python script analyzing their clock synchronization performance and devices in the CNC milling machine (above) and the FrankenPrusa (below).

between a Python script and each device in the milling machine from Chapter 6 and the Franken-Prusa from Chapter 5. In these tests a Python script queries all the remote OSAP runtimes that it discovers using the MVC "Get Time Controller Configuration" message, which is one of the messages in Table 2.6 above. This returns data that is useful to analyze how each device's clock synchronization routine is working, but here we are just using the round trip times (RTTs) for each message, which is simply measured in the Python script. Otherwise, the message load itself does well to approximate all the steps that a runtime manages when it receives any other message: add the packet to the runtime's stack, handle it in the main loop, pull some data out of memory and perform some calculations, and then reverse the route and issue the reply. This message does not go through the PIPES system, but it does use the same data serialization routines that PIPES uses. Each query response is 76 bytes long (including the packet header and the message key) with two `uint64` values, one `uint32` value and eleven floating point values.

To work out which part of each message's total RTT is taken up by the runtime's processing steps (which represents computing overhead) rather than the underlying network link delay, we can remove each packet's wire time (each RTT includes a trip up and back down the wire). For the devices that are connected immediately to the Python runtime (first three rows of Table 2.7, where results here are displayed), the measured processing time includes both the device processing time *and* the Python processing time. Based on the fact that one of these devices (the microbolometer) shows a processing time much greater than the other two indicates that the Python processing time must be below $\sim 250\mu$s. For other devices, we remove from their RTT the mean RTT of the intermediate device. This is presuming that the intermediate RTT accounts for both the upstream wire time and also the Python runtime *and* the intermediate device's processing time to passing the message on. This is somewhat lossy because passing the message is probably faster than handling the message, but it should get us in the right ballpark. In Section 2.9.1.3, I explore how OSAP could change in order to make this type of timing analysis more straightforward, as it turns out to be an incredibly valuable systems assembly tool.

Device	Devices and Links in Route	Closest Link Wire Time	RTT Mean	RTT σ	Max RTT 99.99%	Processing Time Est.
		μs	μs	μs	μs	μs
Message Hub A[a]	USB 2.0 HS (CDC)	3.2	350.0	64.0	541.9	346.8[f]
Message Hub B[a]	USB 2.0 HS (CDC)	3.2	264.7	48.4	580.6	261.5[f]
Microbolometer[b]	USB 2.0 FS (CDC)	32	979.3	185.8	1843.6	947.3[f]
Bed Heater[c]	Hub B + UART	534	822.9	58.3	963.3	24.2[g]
Extruder Motor[d]	Hub A + UART	534	920.0	59.1	1170.6	36.0[g]
B Motor[d]	Hub A + UART	534	951.6	68.1	1246.5	67.6[g]
A Motor[d]	Hub A + UART	534	947.8	66.0	1155.7	63.8[g]
Z Motor[d] (Rear)	Hub B + UART	534	890.9	68.2	1126.5	92.2[g]
Z Motor[d] (Right)	Hub B + UART	534	917.9	71.4	1166.7	119.2[g]
Z Motor[d] (Left)	Hub B + UART	534	920.2	71.1	1140.7	121.5[g]
Extruder Heater[e]	Hub A + UART	534	1217.3	68.4	1567.1	333.3[g]
Load Cell[e]	Hub A + UART	534	1358.9	83.3	2033.4	474.9[g]

Table 2.7: Devices, routes, and round trip time statistics for devices in the FrankenPrusa.

[a] NXP i.MX RT1062 at 600 MHz.
[b] Raspberry Pi RP2350 at 266 MHz.
[c] Raspberry Pi RP2040 at 200 MHz.
[d] Atmel ATSAMD51 at 180 MHz.
[e] Atmel ATSAMD21 at 48 MHz.
[f] This includes processing time in the endpoint device (one step) *and* in Python (two steps: generating the query and receiving the response).
[g] Calculated by removing the mean RTT of the intermediate device (Hub A or B) from the mean RTT of this device, and also removing the closest link wire time.

2.7.9.4 Timing the MAXL Gap

So, with a better understanding of the packet and runtime overheads, we can reflect those onto relevant timing and bandwidth characteristics for MAXL. Earlier I compared data rates between the systems in this thesis and two other researchers efforts to also build model-based machine controllers, showing that data collection in this case is faster than their implementations. Again it is hard to know if their timings are based on systems dynamics or network / systems performance limits.

That said, MAXL also inserts a gap (see Section 2.5.2) between the trajectory source (in Python on an operating system) and sinks (embedded devices). This can be as wide as $128ms$ (7.8 Hz) or as short as $2.048ms$ (488.3 Hz) depending on the underlying networks and the tolerance for occasional errors that arise from the operating system's indeterminacy. Really there is no lower limit in the *framework;* the value I presented just now is based on an optimistic estimate on best-

case computing delays if e.g. we ran all of a system's actuators on high speed links and devices a-la the first two rows of Table 2.7 above.

To quantify this gap, I have Table 2.8. It shows a range of intervals and network / operating systems gaps; these are configurable according to overall system performance. I have discussed how network feedback and inspection tools could be used to configure each gap in order to minimize overall delay, and how PIPES / OSAP enables reflection between program configuration and networking; source routes are configured alongside program wiring and data in the network is typed and inspectable. See Section 2.8.5 for more on that topic.

The second column of the table shows bandwidth required to transmit MAXL control points to four actuators across a range of interval sizes (in Kilobits per second Kbps). That is calculated for packets sent across two link gateways through OSAP and PIPES, including argument type maps etc.; see Table 2.4 for the OSAP packet header (11 bytes in this case) and Table 2.5 for the PIPES data encapsulation and argument mapping (18 bytes for each MAXL control point via PIPES).

The third column of the table is the gap required for safe interpolation of control points according to the interval; each spline interpolator needs at least three points in the future to be loaded in memory in order to compute the basis spline's values.

The last two columns are resulting bandwidths in Hz. The first is for feed forward responsiveness between trajectory generation and arrival at interpolators and the second is for round-trip cycles in feedback systems. This includes total delay between sensor sampling and return of that data through the network under worst case operation, computation of new trajectories and finally their interpolation in hardware.

Interval	Data Rate	Spline Interval Gap	**Net / OS Gap**	Feed Forward	Round Trip
ms	Kbps	ms	ms	Hz	Hz
16.384	57	49.15	98.3[e]	6.8	4.1
4.096	227	12.29	49.2[a]	16.3	9.0
4.096	227	12.29	4.1[b]	61.0	48.8
4.096	227	12.29	32.8[c]	22.2	12.8
4.096	227	12.29	4.1[d]	61.0	48.8
1.024	906	3.072	1.0[f]	244.1	195.3
0.512	1813	1.536	1.0[f]	390.6	279.0

Table 2.8: Intervals, data rates, gaps, and total bandwidth for PIPES / MAXL systems with varying underlying performance. The second row is representative of the 3D printing systems in this thesis and the third is representative of the CNC milling machine.

[a,c] These are the configurations I used for the FrankenPrusa and milling machine.

[b,d] These are the fastest possible configurations for either, supposing we limit the network / OS gap to approximately 2x the worst case RTT as measured in the time synchronization tests.

[e] This is representative of very slow, non-deterministic networks, e.g. using simple wireless links. While this is obviously not responsive, the loose spline interval reduces the data rate significantly and clock synchronization would still enable co-ordinated motion.

[f] These are representative of two configurations for a system that runs e.g. entirely on the high-power microcontrollers that I used for the Hub / Message passing devices; Table 2.7 shows that their maximum RTT is only around 600μs on the safe side. Getting the message *down* to the device takes only half of the RTT.

In practice, I ran the machine systems in this thesis with a "MAXL gap" of around 64 ms. This is only 15.625 Hz and above the threshold for human-perceptible delay for interactive computer systems, e.g. drawing on a tablet with a stylus [153], but under the threshold for closed-loop teleoperation of robotic systems through the human visual-motor loop which is around $250ms$ [154].

This gap worked well because in both the FrankenPrusa and milling machine the trajectory generation steps are feed-forward at runtime, for example the velocity planner that I develop only runs at around 10 Hz. It uses feedback from motors and systems to build the models that it uses to predict optimal control outputs, but doesn't integrate new data at run time; see Section 7.5 for more notes on feed-forward vs. feed-back control and Section 4.6.1 for the velocity planner's overall formulation. Lower level loops in e.g. the motor controllers themselves do use feedback control.

While the timing analysis suggests that I could have configured the gap to be much smaller, the extra gap prevented errors in the long tail from causing problems; while running one of the 3D printers from this thesis for one hour about seventeen million packets are processed through the PIPES / OSAP stack described here. Handling all of those flawlessly would represent an error rate under 0.0000001%. This gets to a core tension in the thesis, which is that I am proposing to move substantial parts of our controllers in operating systems (which are not deterministic) and on simple networks where message passing and compute are merged. Both of these present real

issues when we look towards safe and reliable systems design. At the same time, the benefits are immense. There are a number of ways that the current state of this architecture could be improved towards that goal. Ultimately the task is to move safety authority into simpler layers of these networks where determinism is possible, as I discuss in Section 2.9.4.3. I also think that being able to capture global systems configuration helps in this regard, because you can't debug what you can't see.

I should also note that delays in the table above's *network gap* column represent configurations for worst-case delay, not e.g. average delay. Also, although the actual delay is variable the operation is still synchronous due to the clock alignment step. Feed-back delay is also variable, but in systems where data are expected from multiple sensors, a second "return gap" could be configured in a similar scheme to manage loops that are rolled over multiple devices and runtimes. Pipes Functions' input gating (see Section 2.4.2.4) could be used in these cases to synchronize computation to the slowest-arriving readings.

Finally, the gaps I show here are relevant for systems that span from the operating system into embedded devices, but the PIPES / MAXL architecture does not *require* that we run controllers in this way. The future work section of this chapter (2.9) is largely dedicated to the steps required to embed more computation into hardware to more easily reconfigure lower level devices. That would enable more rapid assembly of distributed but embedded control loops that eschew a lot of the indeterminacy / timing problems that arise in the operating system itself.

2.7.10 Enabling ad hoc Systems Development

2.7.10.1 The CNC Xylophone

As a playful machine demonstration for Fab Class [30], I worked with Quentin Bolsée and Jens Dyvik to build a computer controlled xylophone. It is controlled with two OneDOF blocks connected to two open-loop stepper controllers through MAXL and OSAP. The mallets are solenoids that are activated using RPC calls. To this Quentin added a computer vision system and MiDi interface, but rather than integrating his code directly with the whole MAXL / PIPES / OSAP stack, he simply wrote an *additional* network interface on top of the xylophone's controller and interfaced with that to his own script.

2.7.10.2 The Blair Winch Project

Last summer, I made a small piece of machine art that I called the *blair winch project,* [196] which suspends an orb of light amongst some trees in the woods in Maine (at Haystack [197]).

For this project, I used direct control of the motors' current via RPC to move the orb; rather than using MAXL blocks, I simply wrote a feedback controller in Python that measured cable lengths (more RPC calls to get motor positions), computed torques that would apply the tensions that should move the orb into the desired position, and then sent those torque requests back to motors.

This is a good example of a system whose own *time constant* is much larger than the underlying network, which is not normally the case in machine systems. Here, simply writing the whole



Figure 2.40: Quentin plays a CNC Xylophone interactively, using computer vision to track the position of his fingers above a keyboard so that the Xylophone can slew towards the anticipated note *before* he hits the key, which then sends a MiDi packet through his interstitial controller that is subsequently connected to the machine systems developed in this chapter.

controller in Python is permissible because the total bandwidth between Python and the low level controllers is faster than the system's own dynamics. Exposing low level interfaces rather than *always* relying on bigger abstractions is valuable in these cases and in others that I already described, it allows us to eschew complexity where it is not warranted. In the discussion chapter Chapter 7, I overall try to relate how dynamical constraints, network constraints, and computing constraints are all really combined in the machine control design problem.

Listing 2.33: In this snippet, Quentin opens a connection to a separate Python process (not figured) that calculates his fingers' positions from a video frame. That process sends commands to this snippet, which interfaces directly with MAXL to control the machine hardware. Building machine controllers in software packages where we can more easily "wrap" them in *additional* software-defined interfaces can help to develop ad hoc interactive systems.

```
async def handle_echo(reader, writer):
    while True:
        data = await reader.read(100) # get data from Quentin's process
        if not data:
            break

        msg = pickle.loads(data) # deserialize the data (a 'pickle')
        if msg.get("running", False):
            stop_requested = True
            break

        reply = {"ACK": True}
        writer.write(pickle.dumps(reply))
        await writer.drain()

        if "hit" in msg and msg["hit"]:
            using_a = True
            if "note" in msg:
                p = note_to_pos(msg["note"])
                pa = dof_a.get_position() # control the machine using
                pb = dof_b.get_position() # MAXL OneDOF Blocks
                # no explicit synchronization, just wait for the motion
                # to complete:
                if abs(pa-p) < abs(pb-p):
                    await dof_a._goto_pos_and_await(p)
                    using_a = True
                else:
                    await dof_b._goto_pos_and_await(p)
                    using_a = False
            if using_a:
                await fet_a.pulse_gate(0.85, 6) # drive the mallets via RPC
            else:
                await fet_b.pulse_gate(0.85, 6)
        else:
            if "note" in msg:
                p = note_to_pos(msg["note"])
                pa = dof_a.get_position()
                pb = dof_b.get_position()
                if abs(pa-p) < abs(pb-p):
                    await dof_a.goto_pos(p)
                else:
                    await dof_b.goto_pos(p)
```



Figure 2.41: A light painting, drawn with a cable-driven "Blair Winch" machine using controllers from this thesis.

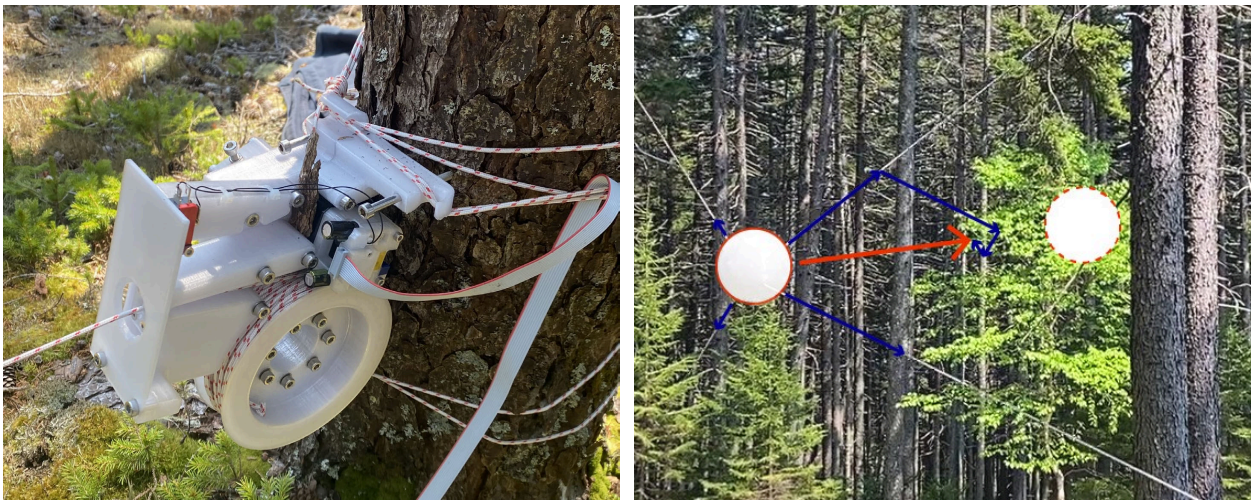


Figure 2.42: At left, the computer controlled winch from the Blair Winch Project, and at right, a diagram of the control scheme for controlling the orb's position using cable tension.

2.8 Discussion

Here I will keep the discussion of these results only to those that are most immediately relevant at the systems layer alone, Chapter 7 explores a broader set of topics that connect through systems design and machine control together.

2.8.1 Key Architectural Limits

There area a few key limits to these systems that I expect are relatively clear by now, but are worth collecting here.

The first set relates to performance and scheduling. Programs in OSAP need to be carefully designed so as not to break the cooperative scheduling routine. So although the systems are designed with the goal of being able to rapidly add new and untested code, doing so can break existing systems. While the MAXL gap enables synchronous operation across high- and low-level systems, the basis spline imposes some additional timing constraints and because the gap is designed against worst-case behaviour it is often larger than it could be. This has to do with scheduling as well because worst case behaviour occurs when the Python runtime is momentarily blocked by either the operating system itself or Python's own garbage collector.

There are also some key configuration limits; link gateways need to be configured manually within each runtime, and firmwares cannot themselves be modified on the fly, they can only be connected to systems in new ways. The configuration system also requires extensive internal tooling, and even then the types that can be used within PIPES remains somewhat limited. This for example prevented me from including also the model fitting routines in PIPES programs; only the data collection routines are there (I mostly use Python notebooks to fit and analyze data). In the introduction to the thesis I mentioned the goal of connecting entire machine workflows in unified software representations, so these systems come up short in that regard.

One more very important distinction: middlewares enable *multicast* message passing: one device can write to a topic that may other devices read from. This is in fact a very useful property that PIPES does not currently implement; data transmission in PIPES is strictly point-to-point. To broadcast data to multiple devices we can configure multiple Pipes from the same function, each terminating at a different input. This is not so different in practice but in potentially large systems where very many devices may want to read data from the same source it could be a bottleneck. The network solution to this problem is to use busses as I discussed in Section 2.2.2.2: over busses, a packet that needs to arrive at multiple locations can simply be broadcast over the shared medium — everyone receives a copy. OSAP does not currently support message passing over busses even though it should (2.9.1.1).

2.8.2 Simpler Representations for Composable Systems

One way to articulate the frustrations that I have expressed about machine systems from the state-of-the-art is to say that they are not *composable;* they are difficult to take apart into pieces and put back together in different arrangements. Machine systems are fragile.

In contrast, music is highly composable: it can be remixed and edited in any number of ways without breaking. This may seem like an absurd comparison, but there are also excellent systems integration tools in music and performance fields that share these properties; MaxMSP, PureData, and OSC are all good examples. All are essentially dataflow systems for music composition and orchestration of not just audio but also lights, cameras, and other ad hoc systems used in performance. Audio also shares timing properties with machine control, in that multiple tracks must be synchronized and computing must be done in a timely manner against a real world constraint.

I thought about these systems often as I developed the architectures here, and I think the key take-away is that simpler representations make composability more likely to emerge.

2.8.2.1 Basis Splines

The cubic basis splines that MAXL uses are the best example of this. In earlier work I used linear segments with additional velocity information to encode motion. Those are perhaps more precise than basis splines because they are direct interpolation, but it is much harder to define how they should be combined. In the basis spline representation, especially where time intervals between each point are uniform, combinations, mixes, etc. are actually very similar to musical tracks. Again StepDance is most similar to MAXL in this regard, and they make explicit mention of tracks and channels of motion data. There are other reasons that basis splines are advantageous at a systems integration level, I discuss those towards the end of this document in Section 7.4.

2.8.2.2 It's All Tuples to Me

While there is a tendency to want to build abstractions to enable simpler control of hardware, I often found that building firmwares first such that they exposed very simple low-level interfaces provided most of the utility required. It is hard to know how unforeseen applications will want to interface with a module, and abstractions that were meant to simplify can end up adding more complexity than they remove.

Being able to drop these layers (or easily see through them) can be valuable. For example in the Blair Winch project I eschewed all of MAXL but was still able to write motion control by operating the motors' as simple torque-producing devices directly. I provided a few examples of these simpler systems earlier in the bootstrapping section.

In the design of PIPES, a breakthrough was to simply use tuples throughout. Serializing, deserializing and naming hierarchically structured data is much more difficult, and if two functions use even slightly different structures reconciling them is nontrivial. Given that embedded systems need to be explicitly typed, this was a tension in the systems design early on. Tuples are much simpler, and the argument mapping and weaving system that I developed in PIPES approximates management of structured data. For example a kinematic component that reads three inputs can

Listing 2.34: Attaching an accelerometer's main data producing function to PIPES in firmware source code. Rather than declaring these data as a hierarchical structure, a flat tuple of values is often enough.

```
// a global function, defined in the accelerometer
// knuckle's `main.cpp`

auto get_data(void){
  // read, flip, sendy
  auto tup = std::make_tuple(
    data_stamps[data_reading],
    data_accel[data_reading].xData,
    data_accel[data_reading].yData,
    data_accel[data_reading].zData,
    data_gyro[data_reading].xData,
    data_gyro[data_reading].yData,
    data_gyro[data_reading].zData
  );
  data_reading = data_reading ? 0 : 1;
  return tup;
}

// wrapped with PIPES_FUNC to compile a PipesFunctionIFace class.
PIPES_FUNC(get_data, "", "stamp, ax, ay, az, rx, ry, rz");
```

be provided with data that is sourced from three separate blocks. This can also be useful in data collection and network management: if we only need a subset of the values produced by a certain function, we can pick them out one-by-one.

2.8.3 Is OSAP an (RT)OS?

I have made some mentions of this question, which pertains mostly to the scheduling problem. Like an operating system, OSAP schedules a set of tasks to run on limited resources. It does so cooperatively, i.e. it cannot stop a task that is already running to run another that has e.g. higher priority. I have mentioned in a few places that this represents a real limit to the systems here overall. Cooperative scheduling is not uncommon, Python and JavaScript both implement "asynchronous" frameworks for concurrent programming; neither truly runs tasks concurrently but simulate doing so with cooperative scheduling. But in realtime systems it is almost a requirement because an uncooperative task can easily cause unsafe operation.

OSAP is also strange in that it primarily schedules packets to handle, not tasks. The proposition there is that in a networked system, packets are a good proxy for subroutines that need to act, and their deadlines are a good proxy for which one should go first. But in PIPES, where we can set functions to run on specific intervals, there is a clear need for preemptive (i.e. uncooperative) scheduling.

There are other benefits to developing OSAP as a "real" operating system, which I will discuss in the future work section (2.9.1).

2.8.4 OSAP and the OSI Model

In the introduction to this chapter I mentioned that the OSI model is only loosely followed, and that it is violated especially in embedded systems where the overhead that network interfaces produce becomes much less acceptable. OSAP makes an attempt to resolve that by clearly separating link, network and transport layers, with PIPES in the application layer. But clearly these systems are interconnected at a deep level: PIPES writes OSAP network routes and extends its discovery layer. I also wrote OSAP with PIPES in mind; to build a compute model that was systems-wide, I developed its runtime to be internally consistent with networks: even though two blocks of software in an OSAP runtime could easily be configured to pass data directly between one another, they do so by writing packets into the runtime's message stack. In fact most of the value that I propose comes from these three layers (including MAXL) is that they are well aligned with one another. Again the same kind of theme appears; abstractions between components are something of a myth, even or especially at a systems level.

There is an interesting RFC (3439) from The Internet Society [198] that is related. In Section 3 they note:

> [...] However, in the data networking context structured layering implies that the functions of each layer are carried out completely before the protocol data unit is passed to the next layer. This means that the optimization of each layer has to be done separately. Such ordering constraints are in conflict with efficient implementation of data manipulation functions. One could accuse the layered model (e.g., TCP/IP and ISO OSI) of causing this conflict. [...] For example, layer N may duplicate lower level functionality, e.g., error recovery hop-hop versus end-to-end error recovery. In addition, different layers may need the same information (e.g., timestamp): layer N may need layer N-2 information (e.g., lower layer packet sizes), and the like [...]

This is all tightly related to the partitioning problem; to split systems we find that we need to understand them more precisely than existing abstractions reveal.

2.8.5 GCode, Timing and the Partitioning Problem

So, I would like to try to relate all of this to earlier notes in the introduction and background sections. There is a thread in Section 2.1.1 and 2.2.2 on how network performance constrains controller design; programs are limited by compute power, controller distributions are limited by timing constraints, and networks are limited by bandwidth and delay. However, each of these layers is normally separated and so optimally configuring them together is a challenge because

representations of the problem change at each layer. This was noted also by [114] in their review of challenges for determinism in cyber-physical systems.

In Section 2.2.3.5 there is a note on how microservice software architectures allow for optimal configurations of distributed software modules in datacenters, and how source routing enables optimal configurations of the networks that connect those modules using software defined networks. Both perform a type of optimization based on *policies* that declaratively express how configurations should be updated according to dynamic traffic and use.

While they are much simpler and smaller, distributed mechatronic controllers have the same set of constraints. In fact, they have even tighter constraints: the networks we use are typically slower, the programs we write are more sensitive to indeterminism and the embedded devices we use have much less compute power. And they have the additional set of constraints that emerge from the actual hardware that is being controlled; i.e. we need higher bandwidth controllers for systems with smaller physical time constants.

In the state-of-the-art, ensuring that these requirements are met by the system constraints is done by hand. This is one reason why e.g. industrial controllers (2.2.4.2) don't allow for reconfiguration: their internal control loops and networks are carefully designed such that performance remains deterministic. If Heidenhain allowed machine users to add new software or hardware to existing machines, there would be no way to guarantee that the new composition would not fail. So, this is perhaps another reason why GCode remains where it is; the intermediate layer is useful in this regard *because* it prevents users from running arbitrary programs on hardware that is designed for performance and safety above all.

On the other hand, we want to bring higher performance, more compute-heavy control methods into these systems. We also want to rapidly assemble systems from modules. This means that we need to articulate the systems integration problem in the same way we would like to articulate machine operation, i.e. as an optimization against constraints that can be discovered rather than defined. The way that I try to move towards that here is to develop inspectable models of operation that span multiple layers of representation. Ideally, each of those could feed information about their constraints and requirements back to systems designers.

Here in the systems chapter I make some progress in that regard (building inspectable models like the SOM), but not much is done about modelling the constraints and requirements. In machine operation I make more progress; in Chapter 4 and Chapter 5 I use machine hardware to assemble component-wise physical models to describe constraints and then optimize over those directly.

So while PIPES, OSAP and MAXL are not ready to robustly solve this problem end-to-end, I think that their architectural deltas from state-of-the-art solutions are worth considering as we take steps towards doing so.

For example the systems object model (2.4.4.1) contains a network map that is overlaid with a software map where data pathways are specified literally in source routes. Function types and transmission intervals are specified directly by PIPES, so it is possible to measure the bandwidths required over each link in the system. MAXL's explicit timing and "gap" relates those bandwidths

to delay. These are requirements, constraints emerge from network performance. The timing analysis in Section 2.7.9 is an example of how networks might be measured, and in Section 2.7.9.4 I relate those measurements to the program's performance (MAXL). In Section 2.9.1.3, I discuss how network analysis could be improved in OSAP by integrating time synchronization more directly.

This presents an opportunity to build a *feedback-based* distributed systems design tool that could discover and reflect a distributed control system's current network loads and limits. That may give designers an interface with which to make intelligent changes either to their networks or software configurations, i.e. modifying their solutions to the partitioning problem against real measurements of constraints.

2.9 Future Work

There are a number of limits to these systems at the moment that I think will be important to manage in the future. The overall goal is ambitious; the hope is to build a set of systems integration tools that almost anyone could interface with to build new mechatronic applications (see Section 8.3). Section 2.8.5 posed the overarching problem: for integration across heterogeneous devices, we need to build tools that allow systems integrators to make sense of the system that they are managing across network, compute, and physical constraints.

I of course have a much longer list of questions and next steps in this regard, but I will try to be concise and high-level here. Organizationally, I will work from the bottom up through OSAP, PIPES and MAXL.

2.9.1 Improvements to OSAP's Runtime and Structure

I think that the most obvious improvement to make in OSAP overall is for it to graduate as a real RTOS. I mentioned this in Section 2.8.3 and the larger discussion here has circled *scheduling,* which is at the core of the whole system; we have limited resources but potentially many tasks to complete and messages to send. Scheduling algorithms define how that is managed, and cooperative scheduling cedes authority on the matter to application layer code. Where we endeavor to allow "almost anyone" to put new modules into the framework, this is obviously dangerous.

So, preemption seems like a requirement, see e.g. Section 2.3.2.2.1. Besides actually implementing this (there are well-defined patterns for that), a primary difficult with scheduling is developing a scheme that exposes handles on the problem to system designers. For example in UNIX processes can be given a `nice` parameter; higher niceness indicates that a process will yield to others more often.

At the moment OSAP is scheduled only based on packet deadlines, which I think is a good proxy for task priority when the system is based on networks. But Pipes Functions that are defined to run at specific intervals need to be included in the same scheme. It is likely that we can simply extend the message stack in OSAP to become a more general purpose task stack that includes

these and other tasks. The trouble there is that an interval task that has timed out should perhaps not simply be deleted, and they should neither be run as fast as possible. This means that encoding priorities for both tasks and message passing could become complex.

Treating OSAP more like a real operating system in the Python context could have real benefits as well. The Python interpreter is the main source of indeterminism in these systems, and really it is not a good site for networking. Instead, OSAP could be written in a compiled language for workstation computing and interface to Python blocks via the Port interface, which is already an excellent abstraction for multicore processing. UNIX allows us to dedicate a set of cores to a particular task, OSAP could run there, and Python codes could pass messages to- and from that interstitial layer. This wouldn't remove the underlying issue that e.g. a motion controller written in Python will sometimes be indeterministic, but it would prevent that code from blocking other network activity.

Addressing multicore operation in embedded devices is also relevant, as more modern chips include two or more CPUs. Because OSAP is based on parallelism and serialization already, this extension should not be exceptionally difficult and the performance benefits could be immense.

2.9.1.1 More (and different) Links

I did reserve some protocol for fieldbusses, but did not implement any in this version of OSAP. They remain highly relevant for distributed control and one ambition in OSAP is to also wrap e.g. CAN or EtherCAT network segments. Overlaying network routes through broadcast layers poses an interesting design challenge; for example consider a packet that is generated in the OS and then travels down a point-to-point link where it is *then* broadcast over a fieldbus. The route descriptor effectively needs to include a "split over ports" opcode.

There is also the question of remote reconfiguration of links. For example in a system where a control and configuration code first connects to another workstation-scale computer, we should be able to remotely generate a new link in that device to subsequently drill down into hardware attached to it. This may involve extending OSAP's MVC layers, but it seems more likely that maintaining the separation between custom code and network-specific code is important to avoid cluttering the runtime (which is meant to stay simple). Discovery *of available network segment drivers* would be one way to manage this.

As I've mentioned these links are heterogeneous, we would like to add Bluetooth, WiFi network segments, etc. OSAP's link gateways can do this, but there are always unforeseen requirements at these layers. There is also the question of more appropriately integrating link-specific capabilities more broadly: WiFi time synchronization (based on simultaneous arrival of wireless packets) can be exceptional, but is specific to the link.

2.9.1.2 More Efficient use of Links

Framing multiple messages into one link frame could be a useful performance boost: if sending and transmitting runtimes need to only encapsulate and decode one packet that contains ten messages the overall processing power and link framing overhead are both reduced. But this again would

require more intelligent scheduling; if one message is available to transmit on a link, do you wait for another in some interval before bundling them?

2.9.1.3 About Time

Good time synchronization is clearly valuable for coordinated control and sensing, but it also has value in scheduling and operation of the network itself. The diffusion-based approach that I developed is simple but not as performant as it needs to be to manage extremely high performance systems. It also requires that we tune gains, which should not be the case for heterogeneous systems. Better methods [199], [200] are based on updating *estimators* for underlying clock skew, rather than controlling skew directly — these approaches look promising. Clock discipline can also be computationally expensive if it is not done carefully, which means that every time a software module has to evaluate the system clock, additional processing overhead is added. On sync itself I will also note that the scheme should be based on "epoch *ns*," which is standard in UNIX systems (a 64 bit nanosecond measure of time that has elapsed since January 1st, 1970). Using this as a basis would both improve precision and also allow data captured from even different runtimes to be reconstituted. Aligning against January 1st would require a global reference in each system, that part is probably not required here but would be valuable where available.

Probably the more interesting direction is to make time management more of a core component in the runtime. For example, I described that packet time-to-live is not counted down *across* a link gateway. But synchronized clocks would enable this, as packet timings could be reconciled across devices. But sync is also stateful; should we want it to be the case that networks don't work until the synchronization is set up? It may be a worthwhile trade-off to make if this set of architectures is specifically organized for real-time systems.

This could also enable better network and performance feedback. If we can assume that all runtimes have synchronized clocks, writing trace packets that measure network performance becomes straightforward. The same would be true for software profiling, which is another aspect of the scheduling problem. If the real "answer" to the scheduling problem is not better algorithms but better feed-back about its constraints to systems designers, these are key.

2.9.2 Expanding PIPES' Reach

I mentioned in the introduction that a core tension in this systems design exercise was between simple representations for operation and the capturing of complex software components. But overall, the system ends up lying somewhere in between high- and low-level devices; it was slightly too complex to fit entirely within embedded devices and slightly too simple to completely capture high-level workflows. There are a few steps to take to expand that scope.

2.9.2.1 PIPES Manager in Firmware

Embedded PIPES devices are not reconfigurable in the same way that the Python instances are; we cannot add code modules there on the fly. But doing so would be invaluable. While I have assembled systems that *can* do these things before, we should bring the same back to PIPES. My master's thesis includes a version of this [167], which is very similar to the model-view-controller

from 2.4.4.1 but including a few more features, namely for instantiating and deleting class instances remotely:

1. To load and delete functional blocks, in firmwares, at runtime.
2. To register software modules that *could* be instantiated within a firmware.
3. To remotely operate other firmwares and devices, from within firmwares (i.e. outfit the embedded PIPES build with our scripting and systems assembly interface).

The first of these requires some amount of runtime resource management, which brings memory management requirements and can be dangerous in embedded systems without a good RTOS. The second requires (or is difficult to do without) an *async* API, which is difficult to build in C++, but straightforward in Rust.

2.9.2.2 Improving PIPES Types

Pipes Functions and Classes are also limited in complexity because they can consume and output only plain tuples. This was a key bridge between high and low-level systems (see Section 2.8.2.2), but e.g. most relevant Python codes are written using Numpy. My colleague Quentin Bolsée has noted that the design pattern used within Numpy should be extensible into a system like PIPES, and the typing scheme that we developed together (in Section 2.4.3) includes reservations for this.

Another reservation that Quentin and I share about the current typing system is that the overhead it presents (see Section 2.7.9.2) is too large. One proposal is to hash types rather than encode them directly. That way they could be checked (but not inspected) against a much smaller representation. The other challenge is in PIPES' argument mapping, which forms a not-insubstantial part of the overhead.

2.9.3 Improvements in Systems Integration Tooling

2.9.3.1 Remote Editing of Source Code

Whereas it is possible to author *many* of the modules we will want to have available in each of our devices, the current strategy (where they are loaded into devices a priori and then assembled) has two major drawbacks.

1. Remote functions become black boxes. Especially because machine design and control mixes nomenclature from multiple domains, it is difficult to ascertain what exactly a functional block *actually does* from its name and input/output types alone (although *types* are extremely valuable here). If we could pull source code from these functions as well (or at least, good documentation), much of this confusion could be avoided.
2. We can never completely anticipate what functional blocks we will want to build. A great programming point I once heard was something like *"my favourite part of dataflow programming systems is the little block I can add that I can write code inside of"* — this rings true.

In Python- and JavaScript-based PIPES / OSAP runtimes, this shouldn't be too much of a challenge, and micropython (a small Python interpreter) may be a good way to implement the same in embedded devices, although that has its own drawbacks. I expect that it is possible to spot-compile functions for a microcontroller and build a kind of *partial bootloader* to inject new

functions on the fly, but this would be a substantial project in itself: it may also be the case that a building a network bootloader handle in OSAP is enough, and where we want to add new blocks we simply recompile the device's entire firmware, reload it, and reset it. In any case, the value in being able to *completely* describe a distributed system in this manner is sure to be of major importance if we go towards trying to build robust, safe, and verifiable systems in an architecture like this.

2.9.3.2 Developing a Visual Graph Interface

It is clear that the systems deployed in this thesis are (1) always distributed and (2) sometimes messy. Structurally, they are all graphs, but I do not have a tool to visualize them as such. I would like to build a tool to do so.

A graph visualizer and editor would let systems developers quickly debug which hardware modules are connected, inspect their APIs, and build low-level data streams between devices. I have built a similar system in the past, but made the mistake of *over burdening* the graph representation: programs there had to be described entirely as graph entities. In an updated version, I would like to be able to interchangeably use scripting and graphs.

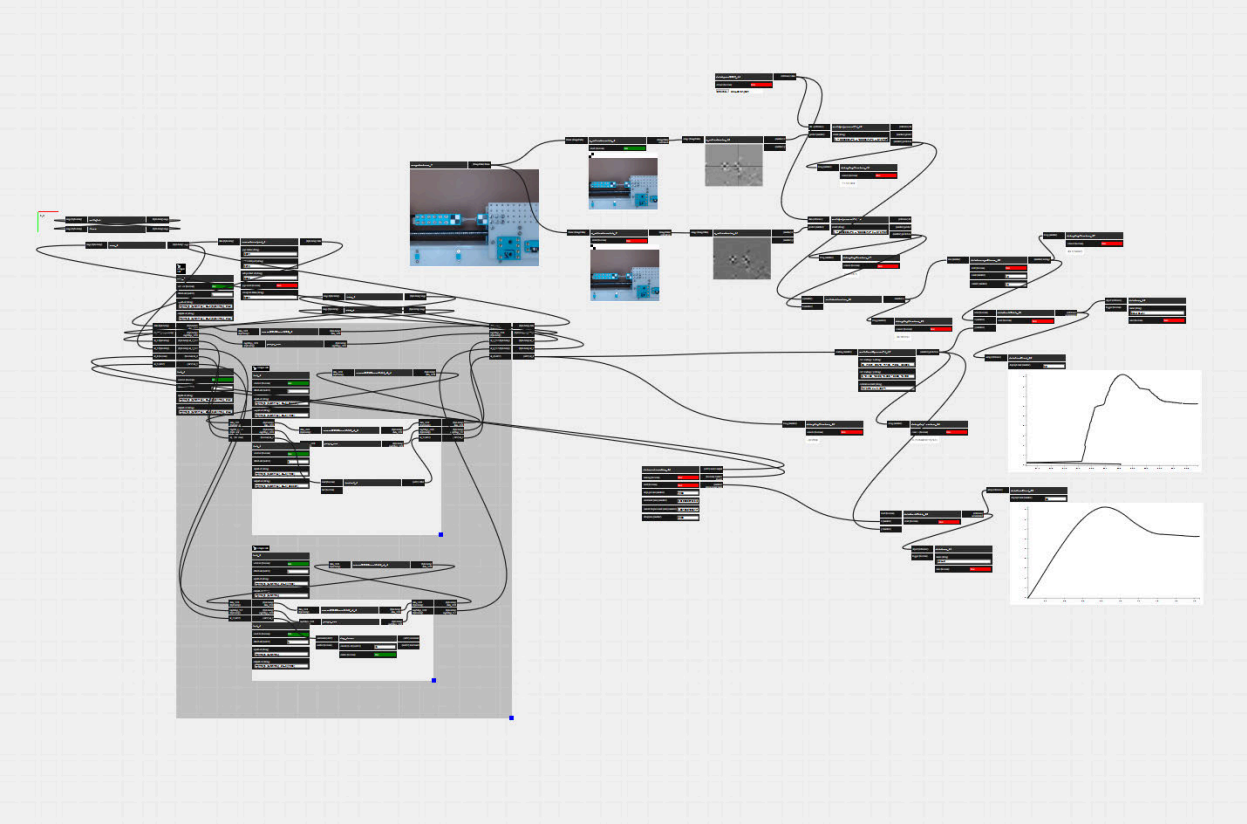


Figure 2.43: This is a screenshot from the development environment UI that I implemented during my masters thesis. The system was functionally similar to work presented in this thesis, albeit of lower quality systems design. The graph visualizer was nonetheless a promising tool, allowing users to coordinate data-flows across all layers in the system.

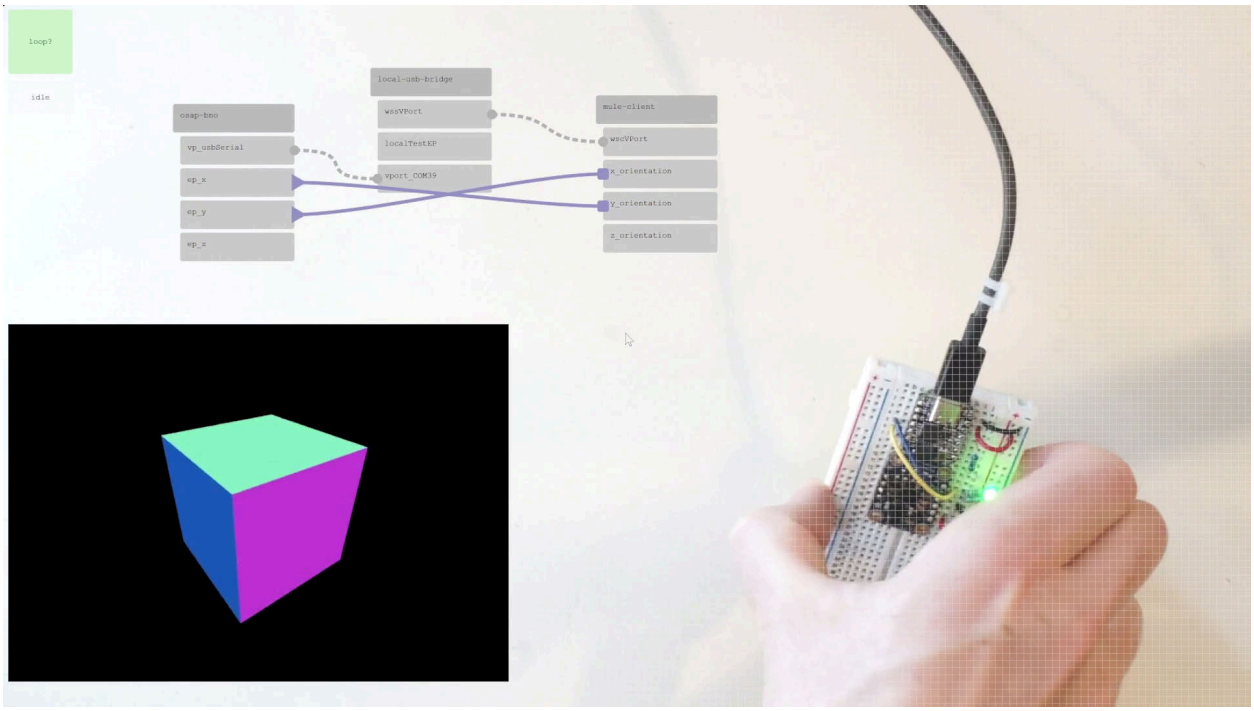


Figure 2.44: In this demo, I show a prototype graph interface to the PIPES system, which lets us discover and then connect an embedded device (an accelerometer) to a software block (an orientation viewing tool). Using a tool like this, programs can be configured and visualized *while* they are running, enabling a tight workflow between systems development and testing.

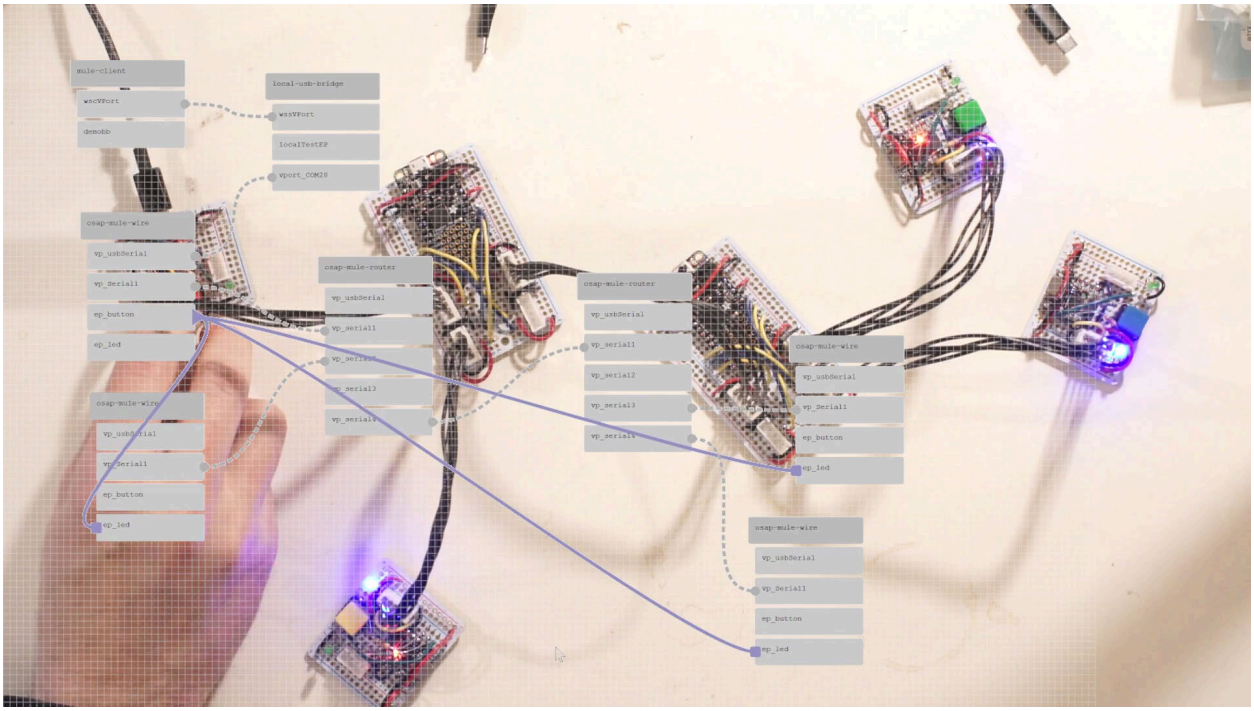


Figure 2.45: Here I show configuration of dataflows across more complex network graphs, including live discovery of devices and dataflows.

2.9.4 Redistributing MAXL

I designed MAXL with the intention of using it to more easily reconfigure distributed motion controllers. In the implementations in this thesis, most of the MAXL blocks run inside an OSAP Python runtime, but the programming model is extensible across runtimes; the CoreXY block (or any other) could be instantiated in firmware if it were authored there in C++.

However, doing this with flexible reconfiguration *within* embedded devices (not just *of* embedded devices) would require the work that I described in Section 2.9.2.1 to be complete. The advantage gained from doing so is maybe clear by now: network and determinism based constraints (the partitioning problem!) mean that it is valuable to be able to relocate some parts of a controller into "the edges" where latency is lower and determinism is higher.

I've already mentioned Ilan Moyer's StepDance [151]; its characteristics are representative of what I am imagining here: responsive, realtime control using composable mathematic blocks. StepDance uses firmware-level block configurations step-and-direction to encode motion (sometimes using a specialized encoding to transmit more than one channel of step-and-direction over the same wire), whereas MAXL uses basis splines and time synchronization. That introduces *the gap* (Section 2.5.2), which introduces at least *some* delay between trajectory generation and execution. The size of that gap is based on network latency, and can be set accordingly; Section 2.9.1.3 discusses how measurements made by OSAP can be used to estimate how minimal this gap can be — but latencies and determinism of software components also figures. This is why profiling tools for networks and for software blocks in OSAP and PIPES are also important next steps.

This is especially true for responsive control interfaces and realtime control loops. Despite all the feedback *via models* that is shown in this thesis, I do not mix very much realtime feedback into control flows, besides at very low levels. But we can imagine using MAXL and PIPES blocks as a basis for mixing and reconfiguring consistently timed control loops, e.g. driving load cell data into a PID pressure controller that drives a torque signal into the extruder motor (see Section 5.12.3.1). Upstream planners could transmit tracking signals to that controller.

2.9.4.1 Using Local Look-Ahead

For a final note, queuing motion as a function of time also opens up the possibility of using smaller look-ahead controller within each device. As we drive performance of our motor drivers in the future (and continue to develop better models of these systems, and more embedded compute performance), using small model-predictive controllers for look-ahead may become more prevalent and common (at the moment most practical controllers are simple PID loops). These require that each device has a future window of control commands to inspect, and time-encoded basis splines will be a useful representation in these cases.

In even simpler scenarios, some devices have known (and constant) lag times between actuation and output: for example an electromagnet or solenoid driven at the same voltage will always have the same delay (between when voltage is applied and when the target current is reached). Devices with fixed lag can simply inspect their target trajectory that many milliseconds in the future, and begin actuation *ahead of time* in order to delete lags.

2.9.4.2 Expansion of MAXL Flow Types

In the earlier MAXL paper [164], I developed a more complete set of "track types" that could encode motion *or* e.g. events, sensor trigger timings, etc. These are harder to synchronously model alongside basis splines (what does it mean to multiply two even channels with uneven timings together?) but maybe the solution to that composability problem is actually simple. There are many potential use-cases for those types of tracks, and I think that their re-integration is worth exploring.

2.9.4.3 In-Firmware Safety Backstops

Finally, but perhaps a critical piece of work, is the notion of pushing safety critical control into lower level devices. In a number of places I've mentioned that one trade-off with model / optimization-based controllers is that they are less deterministic and inspectable by nature. I worked to make the velocity planner that I develop in Chapter 4 more inspectable than other end-to-end approaches, but because it models sometimes messy physics it remains unclear to me if it can be made completely deterministic (see Section 4.8.3). Other advanced controllers (like reinforcement-learning methods) are surely not deterministic. Networks are also never 100% reliable, especially here where we are proposing to add "almost any" link layer technology.

But we still want to deploy these systems in practice because they provide invaluable performance and capability improvements. At the moment I do have simple backstops in MAXL's spline interpolator; if one point is missing, but the two adjacent points are present it is easy to add the missing point between the others. More robust versions of this type of approach could be developed, for example running a per-device energy minimization policy where packets are missed.

3 Modular Circuits for Machine Control

Knuckles

3.1 Introduction

Many machine controllers (and most that are open source) are controlled with *monolithic* control boards, where the control logic is centralized and boards include a limited set of IO devices (motor drives, inputs etc). It is common to repurpose monolithic boards to control machines that they were not originally intended for, but this practice can quickly hit limits (2.2). I won't belabor the point, modular systems have advantages that are probably familiar to readers of this thesis.

There is a growing trend in open hardware towards modular circuit ecosystems, but they hadn't emerged when I was starting this work. Also, most industrial automation systems are already modular, but are not *hackable*. Without a herculean reverse engineering effort, I would not have

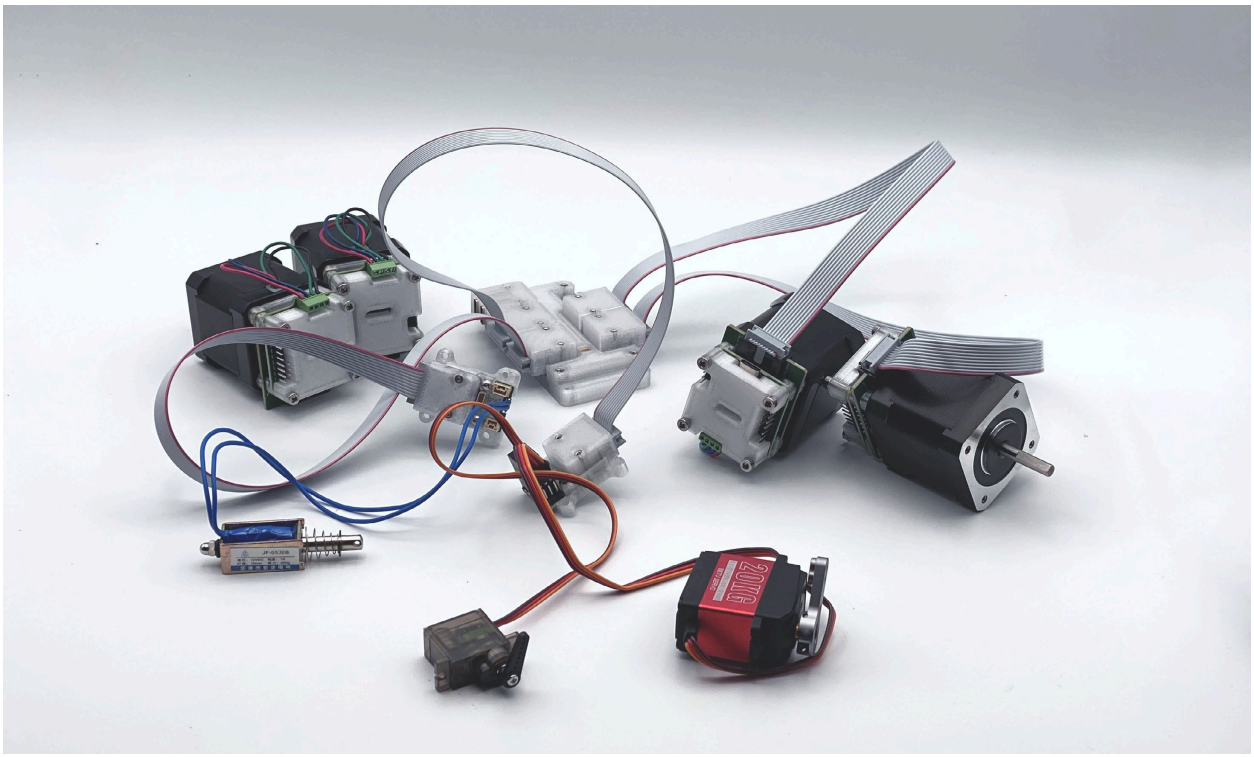


Figure 3.1: I have been developing this reusable kit of modular circuits and firmwares over the course of my time at MIT. With the most recent set of these (pictured here), I focused on re-usability of individual circuits.

been able to use them to develop the *other* methods in this thesis, many of which require that I make edits to fairly low levels in the controllers. So, following in a long Center for Bits and Atoms tradition, I developed my own set of modular circuit hardware.

This system of circuits was a key enabler of the methods and systems that I developed in this thesis, but there is nothing completely new or unique about the hardware itself; my contributions are in the networks (2.3), programming models (2) and controllers (4, 5 and 6). It is best to consider this chapter as a methods section, with some discussion on the design and use of modular circuits as a practice - and how that practice connects to the architectural topics raised in other chapters. In another lens this is a *motivating* chapter, showing how we might be able to flexibly re-use modular hardware if we are successful in our attempts to re-configure control architectures.

So, below I include a brief discussion of the challenges for modular circuit systems 3.1.1 and then the set of circuits that I used to develop the content of this thesis. I call these *Knuckles* because they aspire to help us spread computing and control into the joints and nodes of our networks. Section 3.2 enumerates the control modules, 3.3 explains a swappable network hardware layer that

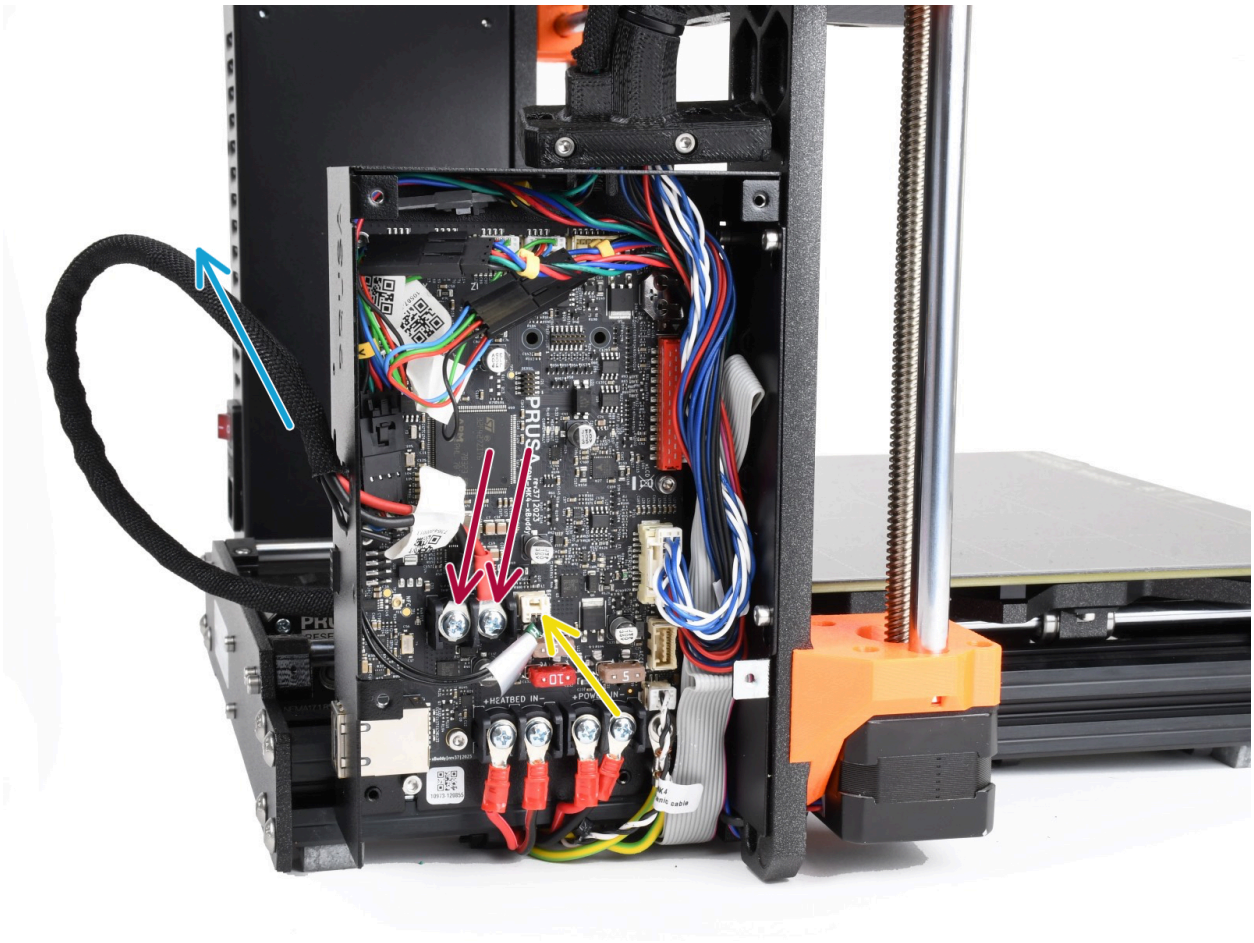


Figure 3.2: This is a "monolithic" motion control board in a Prusa 3D printer. Monolithic boards are designed to cover a range of machine designs, but can be difficult to adapt to more complex systems where i.e. the number of motors required by the machine exceeds the number of motor drivers installed on the board...

I develop (which allows us to re-use modules across different network link layers, basically a meta-connector) and in Section 3.5, the nodes that I developed as dedicated message passing devices. In the discussion section, I highlight some of the ways that I have used these boards, showing device re-use in Section 3.6.1, and machine retrofitting in Section 3.6.2. I close the chapter by discussing the importance of ecosystem hackability (3.7.3) and some thoughts (3.7.2) on the relationship between power networks (which should be Directed Acyclic Graphs (DAGs)) and data networks (which, ideally, can be Cyclic Graphs).

3.1.1 Challenges in Modular Hardware Design

One of the primary challenges that sets modular hardware apart from modular software is that hardware is more difficult to make *generalizeable;* not only do specific tasks require specific hardware *arrangements,* components also need to interface across variable networks and at variable power levels. For example even though motor drivers are all *essentially* the same, there is a broad range of particular motors to drive, each needing e.g. more or less current, voltage, and variable packaging. Besides trying to generalize circuit modules for different tasks, we may want to deploy them in different contexts: motor controllers can be found in battery powered robots or drones, or in mains-supplied machine systems (each running on different voltages). There is also a heterogeneity of networking solutions available in mechatronic systems (which I discussed in 2.1.1): users of hardware modules may want to integrate a new device on a CAN Bus, over Ethernet, or simply via USB or UART, etc. All of this means that even for the same *core function,* we need to develop many possible pieces of hardware to appropriately suit the task and context. In software (where modules are more common), computer languages, package managers, operating systems and compilers make it easier to deploy the same lines of code in different contexts; we hardly even think about it when we can write code on our PC's and then have it run identically in a datacenter, or on a different instruction set, etc.

Another challenge for modular hardware design is to develop a set of modules that can be broadly applied across different tasks. From an economic perspective, it is more productive to develop a smaller set of modules and apply them to many tasks to better amortize module engineering costs. From a systems engineering perspective, it is better to make general purpose modules that can be applied across many tasks because this means systems assemblers can pick from a smaller catalog of parts to accomplish their goals. Of course this is *most* of why we want to build modules in the first place: by breaking a system into parts, we can easily re-assign those parts in new systems. The challenge is in making performant and simple modules that can also be "assignable" to many different tasks. Much of that challenge applies more to the programming models within our devices, but the hardware matters as well.

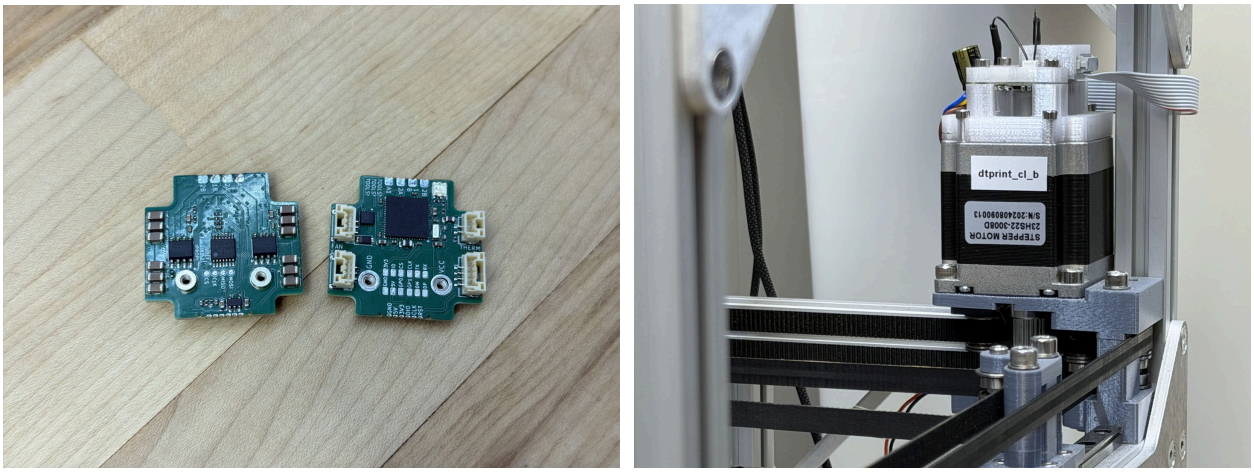


Figure 3.3: The Field-Oriented Control Stepper (4.4), which uses an ATSAMD51 mcu with two DRV8251A H-Bridges and an AS5047 magnetic encoder. The board (top and bottom) at left, and set up on the RheoPrinter (5.4.1) at right.

3.2 Knuckles Devices

The current kit of circuits includes five devices, which are all below. This set — and some ad hoc inclusions (3.7.3) cover the tasks in this thesis. I use a mixture of microcontrollers (mcus); SAMD21′s for simple boards, SAMD51′s for the stepper driver, and more recently the RP2350. The availability of cheap, high-powered microcontrollers is a big part of what makes this project possible.

The closed-loop stepper driver appears the most frequently, I write about it at length in Section 4.4.

The hbridge driver is useful in many contexts as a generic DC load driver. I discuss those in Section 3.6.1.

The load cell amplifier can sample at nearly 4kHz. I use it to measure pressure for modelling polymer flows in Chapter 5.

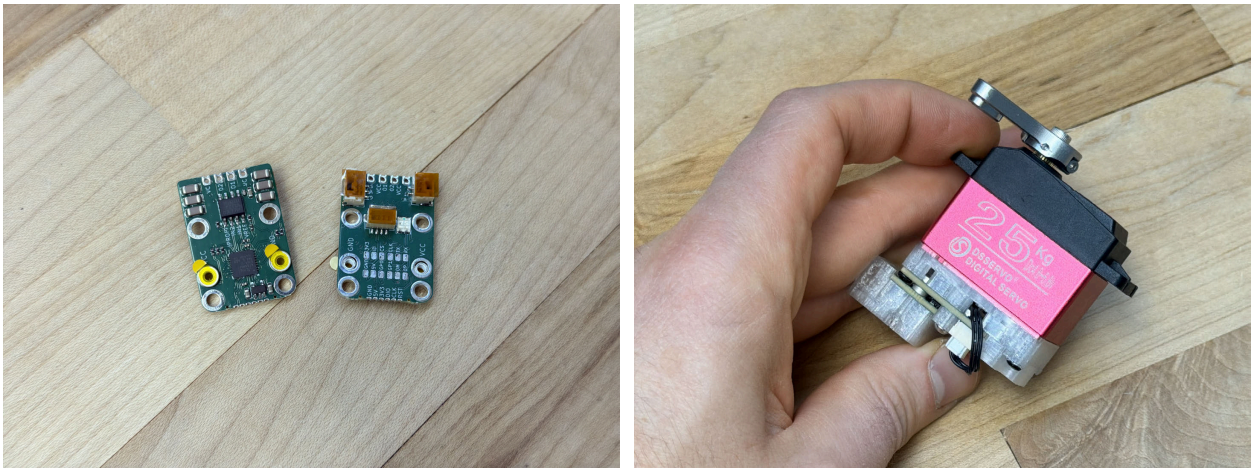


Figure 3.4: The HBridge Knuckle, which uses an ATSAMD21 mcu with the DRV8251A hbridge. The board (top and bottom) at left, and set up to control a hobby servo (right).

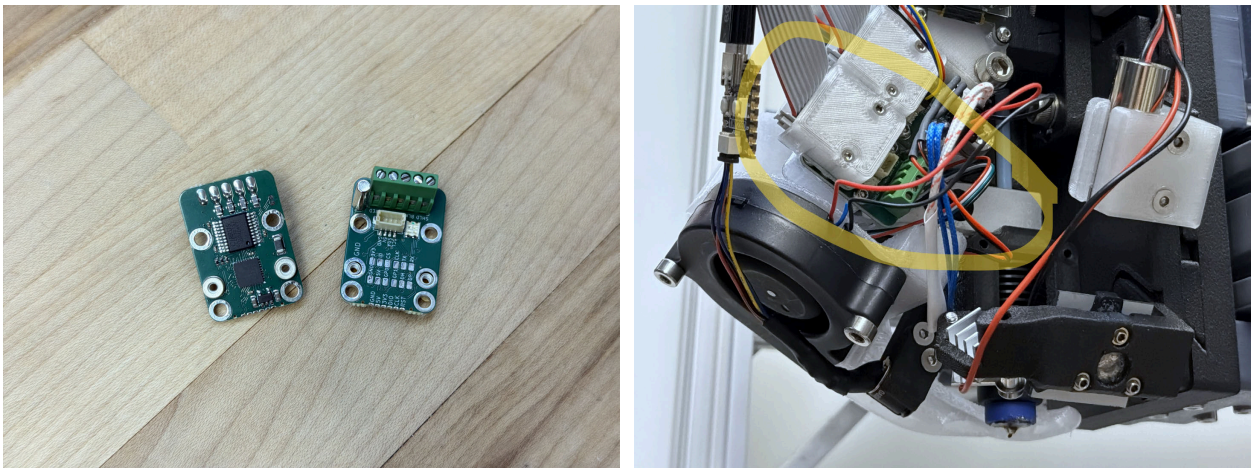


Figure 3.5: The Loadcell Amplifier Knuckle, which uses an ATSAMD21 mcu with the CS5311 wheatstone bridge ADC. The board (top and bottom) at left, and set up in the RheoPrinter (5.4.1) (to measure extrusion pressure) at right.

I developed the accelerometer module recently, but have not completed writing its firmware. I discuss its future utility in Section 4.9.3.

The deadbugger (named after the ad hoc electronics assembly practice of building *deadbug* circuits [201], [202]) is an odd module; I developed it to cover edge cases in modular systems where we simply want to turn something on or off, *and* to drive 5V loads like hobby servos. It is essentially just a SAMD21 microcontroller, with our network interface (below), exposed connectors / pads, and a large 5V regulator. I discuss it as well in Section 3.6.1.

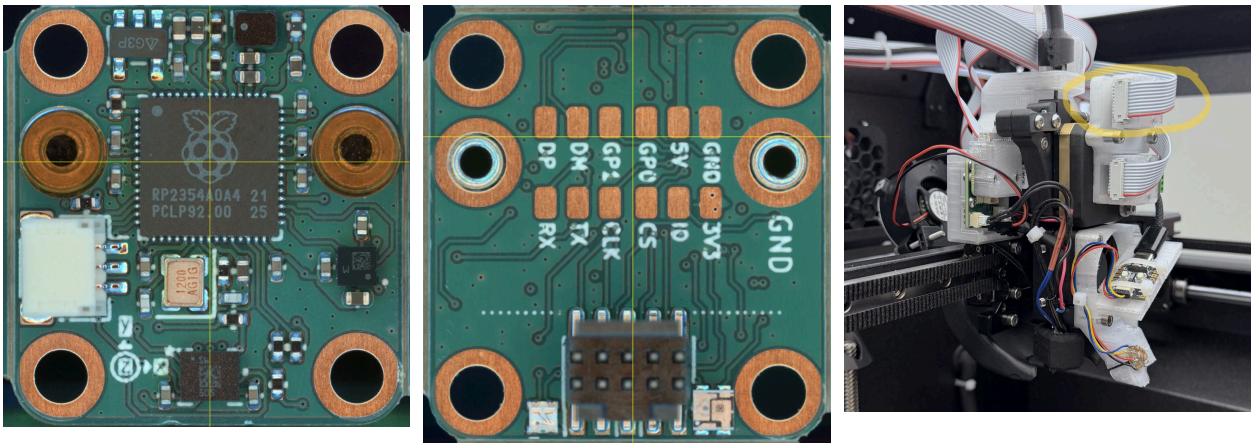


Figure 3.6: The Accelerometer Knuckle, which uses an RP2354A mcu with the LSM6DSV16X inertial measurement unit. The board top (left) and bottom (center), and set up in the FrankenPrusa (5.4.2) at right.

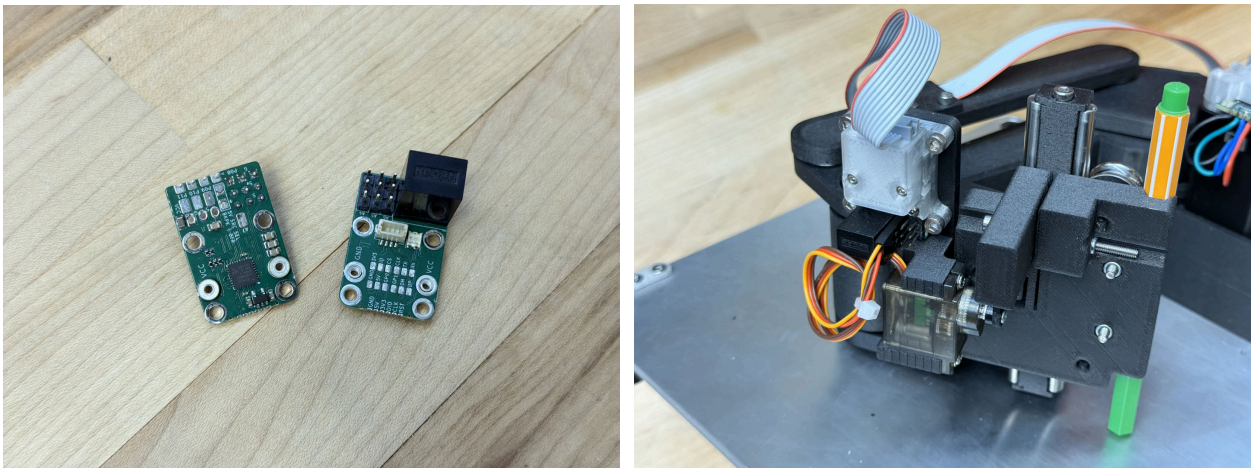


Figure 3.7: The *"Deadbugger"* Knuckle, which uses an ATSAMD21 mcu. The board (top and bottom) at left, and set up to control a hobby servo (right).

3.3 Knuckles Backpacks

One of my ongoing frustrations with circuit development was that the board designs were often coupled to systems design. I would build a new networking layer, design a series of circuits for that layer (including the correct connector and interface hardware), and then I would subsequently improve or modify the networking layer, rendering that set of circuits defunct. This was seriously unproductive, but is representative of something that is true more broadly in networked hardware: link layers are heterogeneous.

That is to say, in different *systems* contexts, we want to use different types of networking. In a high performance machine controller, we probably reach for EtherCAT, ProfiNET (etc) — but these are expensive and complex to integrate. For small systems with just a few devices, a simple USB connection to each module is sufficient and productive, since everyone has USB hardware already "lying around" [203]. In the middle ground we can use simple wired links like CAN or ad hoc UART based busses, and for remote systems we want to implement wireless links like BLE, WiFi or (for long ranges) protocols like LoRa. Ideally, we would be able to mix and match these systems — using some wireless devices, some wired, etc.

To counter this issue, I began developing what I call *backpacks.* These are *PHY* level modules; they interface from our devices' microcontroller to various links. Early backpacks used a simple eight pin header to interface between devices and link layers, some of which I show in Figure 3.8.

Using backpacks means that I can build A : a set of modular circuits and B : a set of modular PHYs, and rapidly combine them depending on the context rather than filling out the $A \times B$ matrix of unique circuit designs.

Backpacks use two 2x3 spring-loaded connectors to connect to exposed pads on the host module. They mount to that module using surface-mount standoffs: through-hole standoffs on the backpack, and threaded (M2 size) on the module. The stand-offs do double duty as power connectors, they are

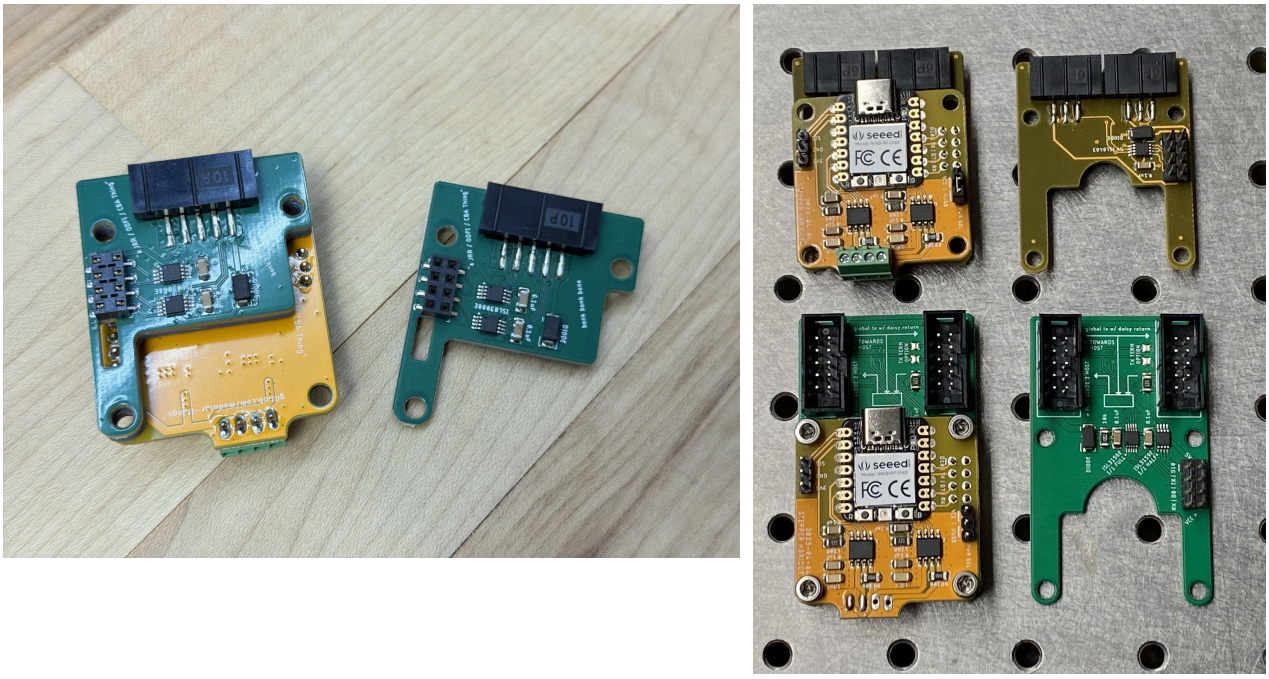


Figure 3.8: These are an early backpack design, showing a motor driver (orange), and backpacks for: Full-Duplex UART over RS485 (at left, the same link technology as I use extensively in this thesis, and in Figure 3.10), Half Duplex UART over RS485 (right, top), and a daisy-chained UART architecture (right, bottom).

rated to carry up to 70 Amps of current. The Backpack interface is $20 \times 17.5mm$, with standoffs $15mm$ apart on center. It specifies a pinout, which is below in Table 3.1.

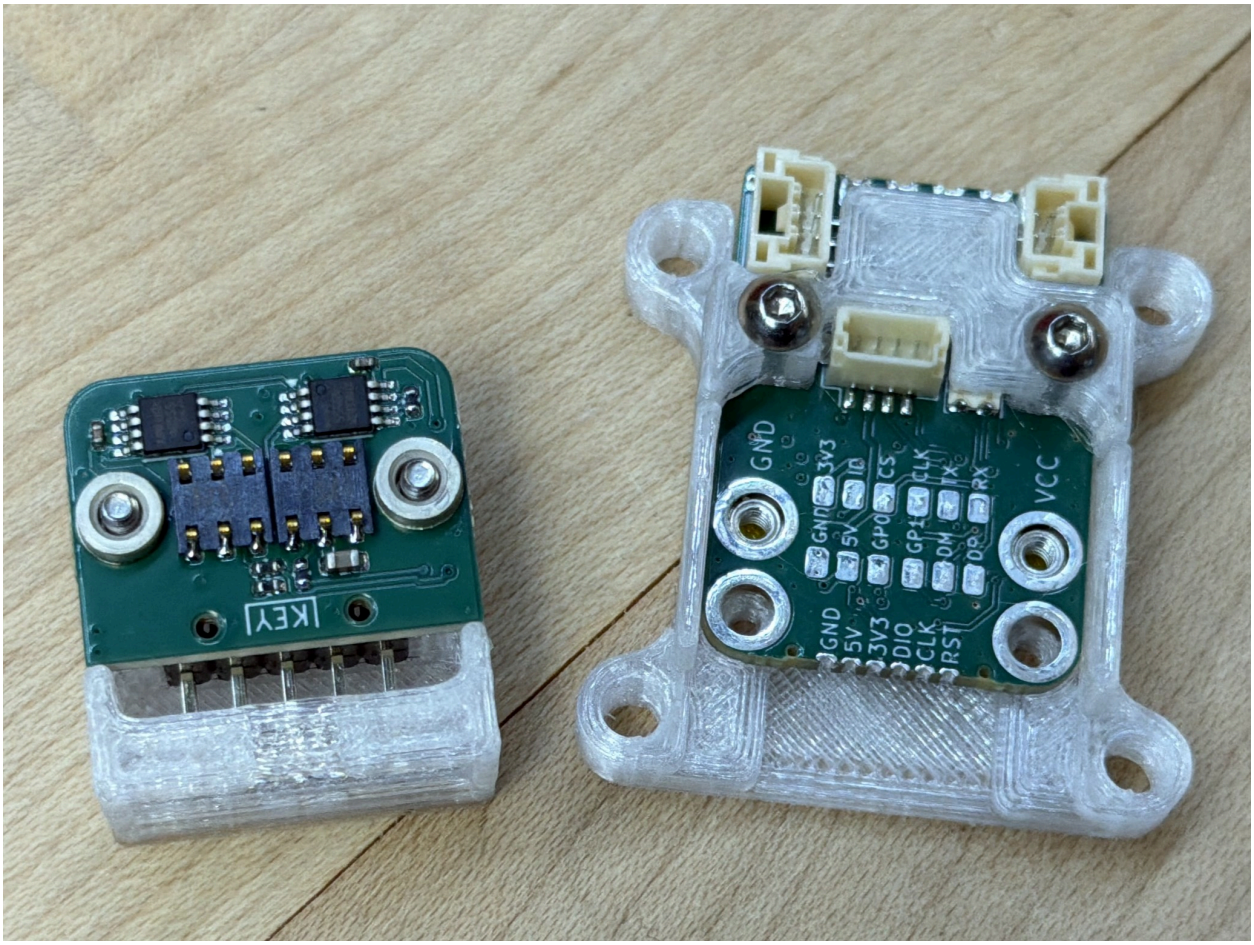


Figure 3.9: Left: a backpack with two 2x3 leaf spring style connectors (black with gold contacts) and through-hole surface-mount standoffs (zinc-plated cylinders). At right, the HBridge Knuckle with the reciprocal interface; GND and VCC connect via contact with the surface-mount standoffs (underneath these holes are threaded standoffs, an M2 screw bolts them together), and leaf springs touch down on exposed pads. A complete pinout is in Table 3.1. The catellated pins (towards the bottom of the figure) are a second interface for programming and debugging of the module, this is not part of the backpack interface.

Pin	Name	MCU Pin / Peripherals	Note
Standoffs	—	—	—
GND	Power Ground	n/a	
VCC	Power Voltage	n/a	
Left Col	—	—	—
GND	Data Ground	n/a	
5v	5 Volts In[a]	n/a	Sourced from Network
GP0	GPIO 0	Any / GPIO	
GP1	GPIO 1	Any / GPIO	
DM	Data Minus	USB_DM, LVDS_LO	
DP	Data Plus	USB_DP, LVDS_HI	
Right Col	—	—	—
3v3	3.3 Volts Out[a]	n/a	Sourced from Module
ID	ID / Type	ADC	Backpack Identifier
CS	Chip Select	SPI_CS	
CLK	Data Clock	SPI_SCLK, UART_CLK, I2C_SCL	
TX	Transmit[a]	SPI_COPI, UART_TX, I2C_SDA	
RX	Receive[a]	SPI_CIPO, UART_RX	

Table 3.1: The Backpack pinout.

[a] Names are relative to the *module,* so e.g. Transmit is data departing the device, Receive is incoming, and 3v3 *out* is sourced from the module, whereas 5v *in* is sourced from the networking device. I discuss this choice in 3.7.2.

A complete spec would need to include more detail like power requirements and ratings, voltage levels and permissible data rates for signals, etc. This is a prototype. A few notes on the design of the pinout:

1. The ID pin is connected to a voltage divider on the Backpack: by specifying resistance levels in that divider (the top of which is connected to 3v3), microcontrollers can read the voltage to ascertain which type of backpack is connected. This is an ad hoc solution to the problem presented by the intended use, where modules can load variable drivers on boot to run backpacks that may have been swapped since the last firmware update.
2. I specify that the *5v* pin delivers "logic" power to the module, and the *3v3* pin returns regulated power to the backpack. In retrospect, this was an odd choice when compared to other modular systems where it is typical that modules include their own voltage regulators, stepping down from their "main" power input, i.e. high voltages in the case of motor drivers. This arrangement assumes a tighter integration between network power supplies and modules. However, it makes it very easy to outfit simple sensors (that only require logic-level power), and to power devices' microcontrollers to configure them before delivering potentially dangerous levels of power.



Figure 3.10: The UART over RS485 Backpack, bottom (left) and top (right).

3. Many pins do double duty, i.e. *CLK, TX and RX* are available for *either* UART, SPI or I2C integrations. I presumed that use of these peripherals would be mutually exclusive on any given backpack. Modules are free to implement as many of these as they like, and replace empty sets with general purpose GPIO *or* connect all peripherals to anticipate variable backpacks. However, the pinout should be modified to clearly reflect which are available on the module.
4. Adding interconnect can cause degradation and crosstalk or very high speed signals and network connectors are carefully standardized for this reason. The latest swappable memory modules for modern laptops LPCAMM2 [204], [205] (for Low Profile Compressed Memory Module) uses a similar design pattern to these backpacks (but much more advanced) and supports clock speeds of 4.8 GHz. These are known as Land Grid Arrays (LGA) and can also be found between personal computer and server CPUs and motherboards; they work well because they keep the total length of the connector small, in LPCAMM2 the face-to-face height is either 0.6 or 1.0 mm, the connectors I use are similar at 1.2 mm. To support extreme data rates however they also interleave ground pins with data pins to minimize crosstalk, which this connector does not do.

The *UART/RS485* Backpack (above, 3.10) connects the module's UART peripheral to others over RS485 (a voltage signalling standard), using a 10-pin IDC connector and cable. This technology is extremely simple and common, but (with well written firmware) can be very high performance.

The *USB Dually* Backpack (above, 3.11) connects one USB-C connector to the module's mcu, and the other to a *USB Power Delivery Decoy* chip, which handles the protocol required to request high voltage power from modern USB-C chargers and power banks. This lets us connect to and power machine systems or modules using ubiquitous power supplies.

In many cases, the host microcontroller contains the hardware required to interface directly into the physical layer — i.e. the link layer logic is implemented in the host microcontroller. For example just about every microcontroller on the market includes one UART peripheral, and probably one SPI, and often a USB block. In these cases the backpack interface lives between OSI layer 2 (Data Link) and 1 (Physical, aka PHY). For example the backpack at the left of Figure 3.9 promotes UART *TTL* (Transistor-transistor Logic, which comes directly out of the microcontroller) to *RS485* — a larger, differential voltage level that improves noise immunity. The microcontroller



Figure 3.11: The USB Dually Backpack, bottom (left) and top (right).

can easily implement the *protocol* that promotes that UART peripheral (which simply sends and receives bytes) into a proper Link Layer (encapsulating packets and catching errors). For some more discussion on how these layers are organized, see Section 2.3.1.

It is common for more complex link layers like EtherCAT and BLE to be integrated via standalone link layer controllers. This is because link layer operation is a time critical task, and microcontrollers that are running application code are not guaranteed to be available to handle packet frames in time. In these cases, it is common for the link layer controller to be interfaced to the host microcontroller using another protocol that is effectively hidden in the OSI layers model. For example EtherCAT and Ethernet controllers are normally connected via SPI, USB controllers via UART, etc. These can be accommodated in the backpack ecosystem, integrating the link controller on the backpack and mounting those SPI / UART connections via the backpack pinout.

I want to note that this sneaky insertion of (what is effectively) an additional network link in the OSI model is a common design pattern in embedded device development. There is even a set of standardized link-controller-to-phy interface for Ethernet: the MII/RMII [206], [207] (Media Independent Interface, or the *Reduced* version) enables Ethernet to be broadcast across various media: these are, again, effectively GPIO specifications. In optical networking, we find things like *Small Form-factor Pluggable* "network interface modules" (SFPs) [208] that allow individual ports to be reconfigured to use different types of transceivers. These are often even hot-pluggable.

Obviously the sticking point is that we would all have to agree on the backpack layout. While this may seem unlikely, I would contend that it is a design pattern worth looking at. GPIO based interfaces have already been productive in open hardware, and even have two apparent hardware-design rivals (Adafruit and Sparkfun) sharing one design pattern in Qwiic / Stemma (they are the same thing). The shared value in such design patterns is the tricky engineering on either side of the interface doesn't have to be duplicated relentlessly.

3.4 Knuckles vs. NICs

In Section 2.2 I surveyed some existing distributed systems architectures. Of course most of those are based on existing network stacks: IP-based or fieldbus-based systems like CAN and EtherCAT. However, simpler microcontrollers tend not to include peripherals to interact with these stacks, even USB peripherals were rare until recently. They do have simpler communications peripherals like UART and SPI: these are not network links, they merely serialize data from the microcontroller's memory into sequences of voltage signals on the device's pins.

To connect to external networks we have NICs (Network Interface Controllers), these are integrated circuits that offload networking routines into a dedicated ASIC and subsequently communicate with embedded devices over a NIC-specific SPI or UART based protocol. This is architecturally similar to how *bridges* work to connect embedded devices to IP-based middlewares, see Figure 2.24 in Section 2.6.1.2.

The advantage of course is to enable operation with these networks where no peripheral exists on the microcontroller to do so. Another advantage is that they effectively add computational power to the microcontroller, which no longer has to compute packet checksums / handle the arrival and transmission of bytes onto the network's PHY etc.

The disadvantage is in cost and complexity *and* in timing: the SPI or UART bridge between the two IC's is another hidden link in the network that adds its own delay and bandwidth limits. It

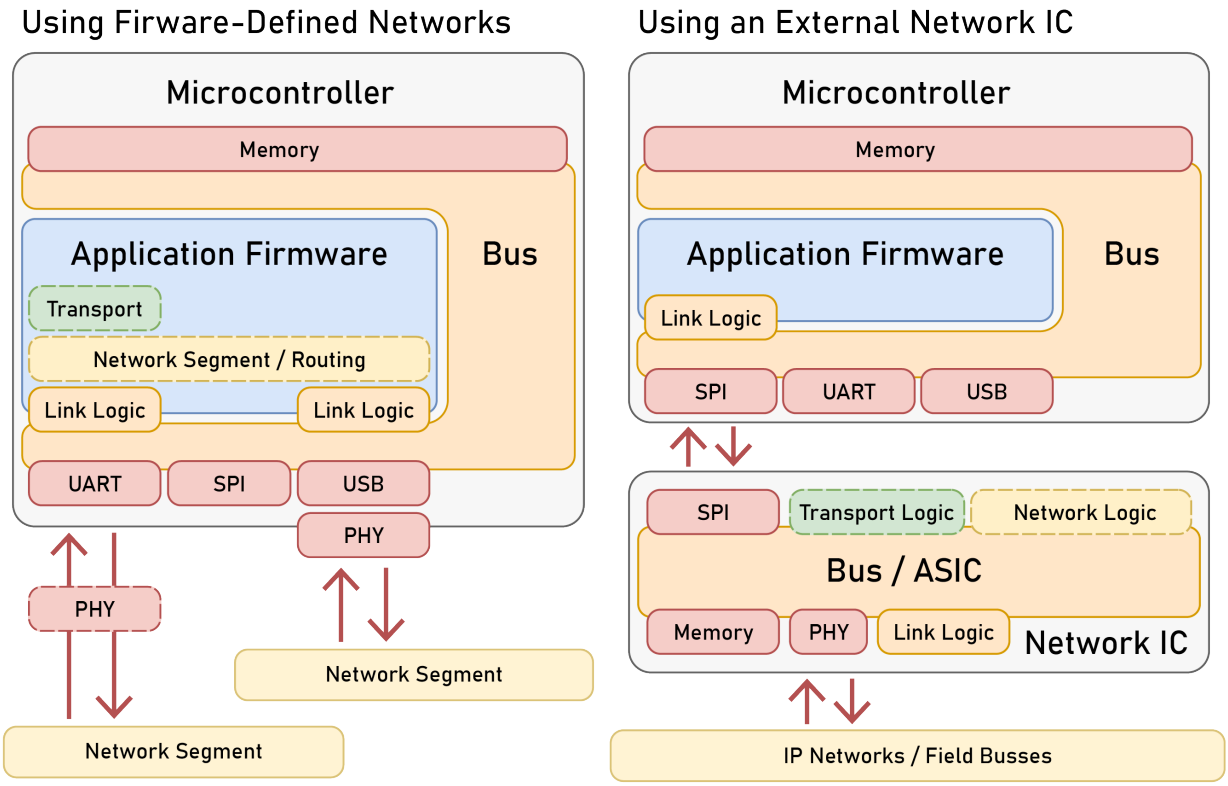


Figure 3.12: NICs (right) vs. Firmware-Defined Link Drivers (left)

can also require configuration of the NIC over the SPI or UART connection, meaning that network state now spans the two ICs. There is still some protocol here which adds complexity, and at the end of the day the MCU still needs to read and write the bytes that will go out on the network, so it is not entirely unburdened.

The alternative is to connect SPI, UART etc. directly to the network and build software-defined fieldbusses and point-to-point links. This is common practice and I used one-such design extensively in this thesis, a UART link that is promoted to RS-485 signalling through a ribbon cable. The strategy burdens the microcontroller with link layer control (error detection and packet framing), but allows us to directly write packets onto network segments. Moving network interface definitions into firmware (for firmware-defined links) also enables flexibility — when combined with the Knuckles backpack, we can reconfigure circuits and software to be used across types of network.

The disadvantage to *this* strategy is that it is not standardized. Although building a custom fieldbus can often be simpler than using an existing solution it of course prevents interoperation with off-the-shelf or other existing devices.

3.5 Knuckles Hubs

OSAP and PIPES are agnostic about device network architectures, i.e. there is no distinction made between "routing" devices and "endpoints." This means that any Knuckles device can implement multiple backpack connectors, enabling it to connect to more than one device and serve dual duty as a device and a kind of network router. In practice, networks are faster overall when their graphs are short, e.g. where we have very fast routing devices with very many ports, keeping the total number of hops between devices low. For this, I developed one hub device (pictured below), that uses a high performance microcontroller and has five Backpack interfaces. In addition to use in message passing, we can implement some firmware here for system control and management.

The hub also provides some power management functions: we can shut off downstream ports' logic (5v) and device power (0-48v). This provides a convenience for configuring systems-level safety: the hub can turn system components off when i.e. watchdog conditions are not met in the case of watchdog hangups or other failures. It is also valuable to be able to remotely reset devices by simply power cycling them.

3.6 Using Knuckles

3.6.1 Device Re-Use

In the introduction to this chapter I discussed how we would like to have hardware modules that are *generalizeable* across tasks. While this is primarily a challenge in the development of module

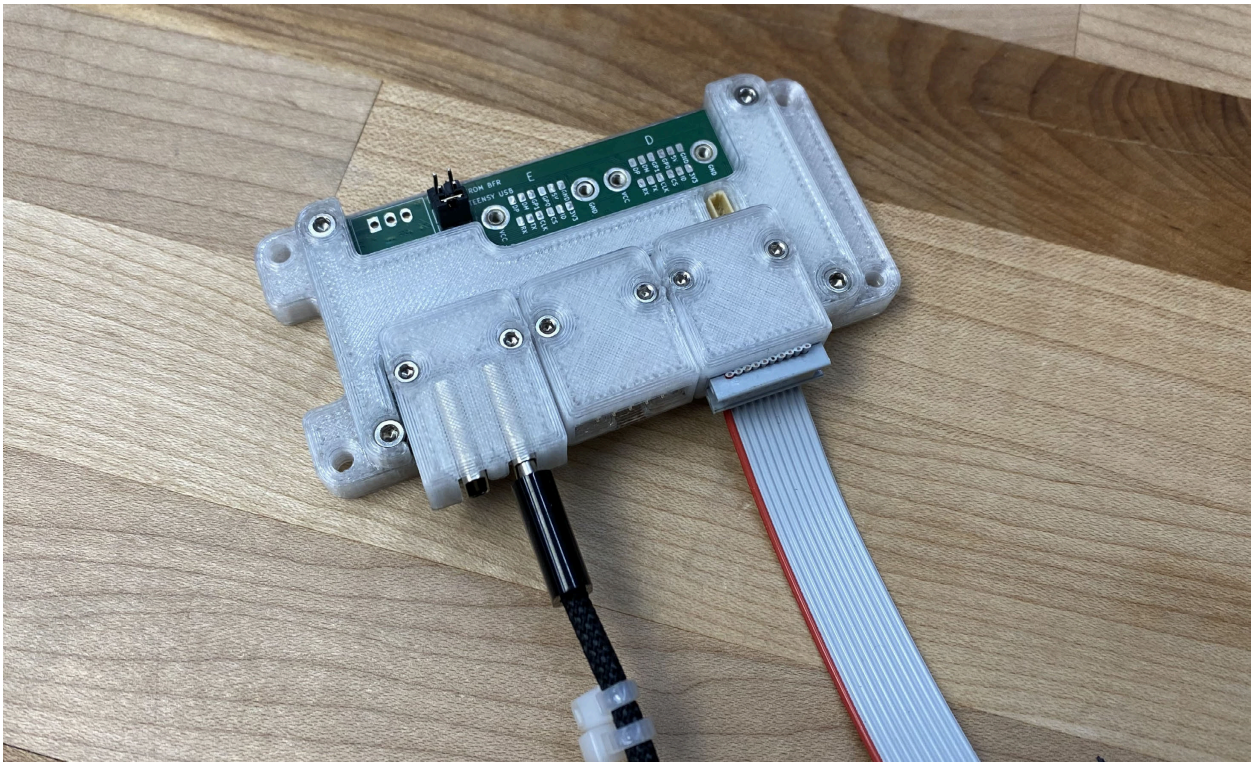


Figure 3.13: The Knuckles Hub circuit, shown with one USB Dually Backpack and two UART/ RS485 Backpacks.

firmware and control, the hardware also needs to be designed with enough affordances that it can be reconfigured.

HBridge connect electrical loads to voltage and ground through either a forwards or backwards path using four switches arranged in an "H" looking configuration, hence the name. They are most often used to drive DC motors, but of course any generic electrical load is valid: heaters, solenoids,



Figure 3.14: The hub with all Backpack interfaces exposed (left) and connected (right) to one USB Dually and four UART Backpacks.

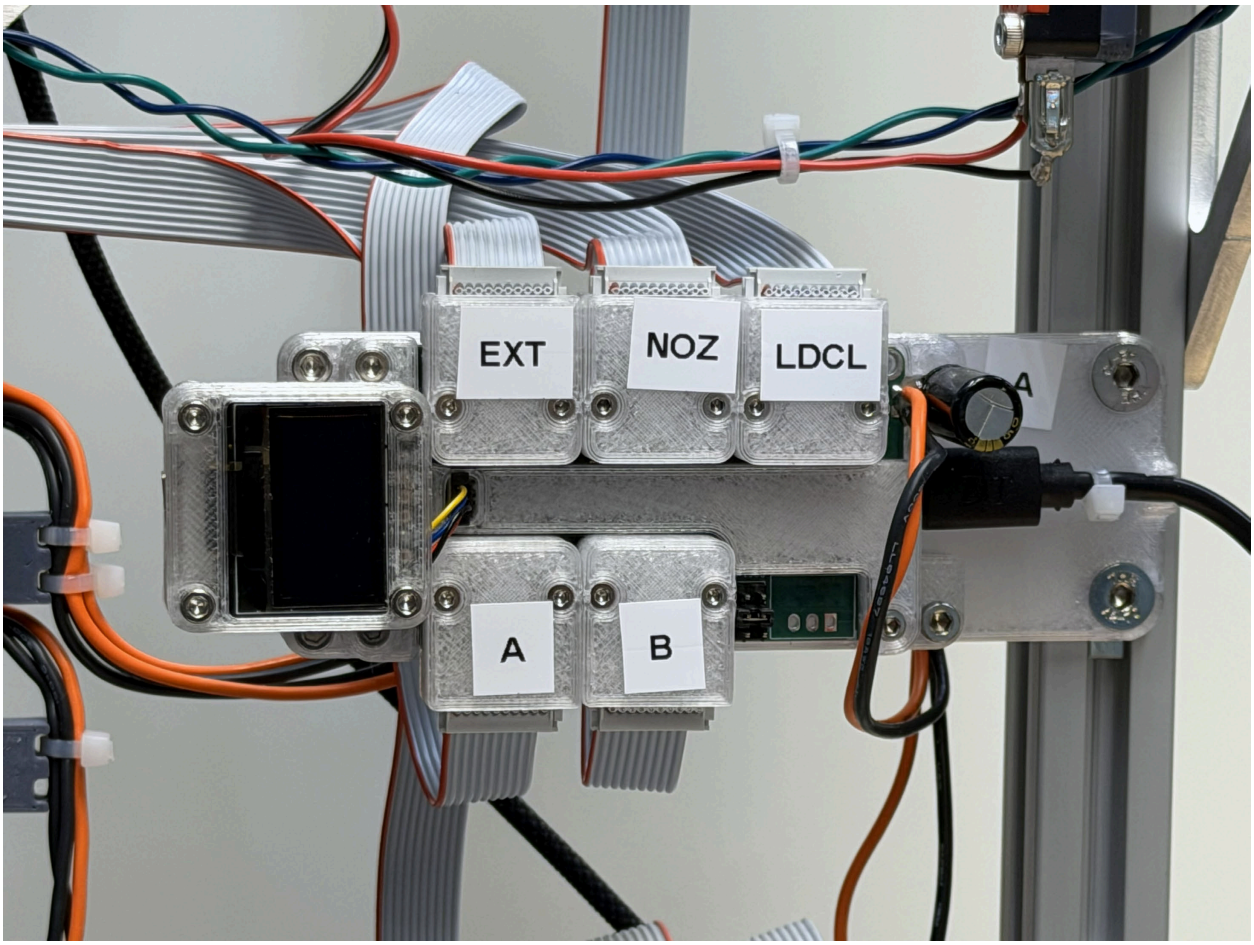


Figure 3.15: The Knuckles Hub as integrated in the RheoPrinter (5.4.1). This is one of two Hubs used there, which is connected to the "top half" of the machine: the motors which control it's motion in XY, and the extruder system.

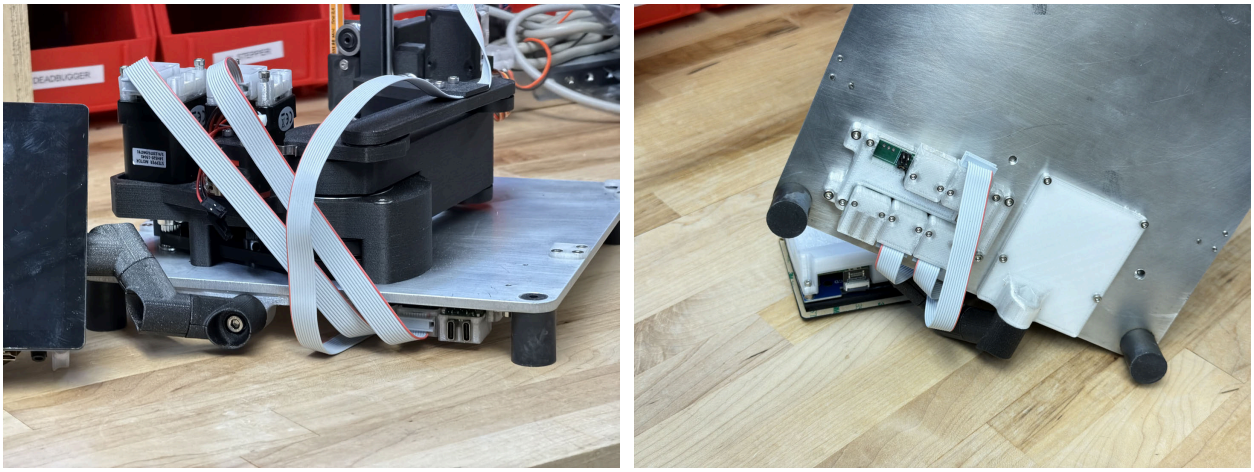


Figure 3.16: The Knuckles Hub as integrated in the *Little Guy,* a small plotting machine that I developed to test MAXL's ability to manage nonlinear kinematics (2.7.5.1).

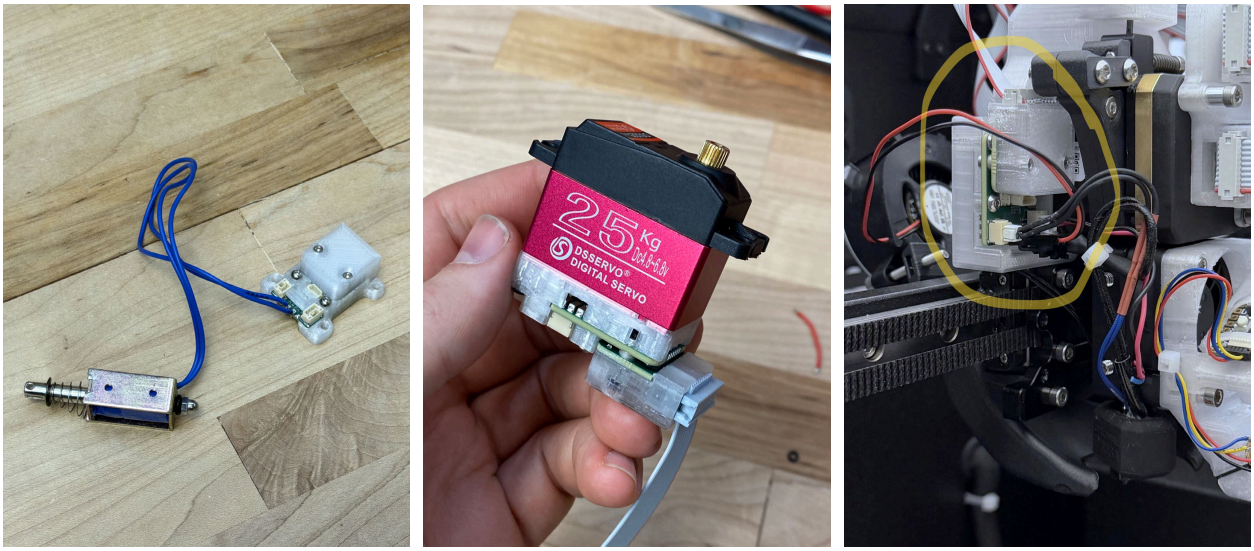


Figure 3.17: Here I show the knuckles H-bridge driver used in three contexts: to drive a solenoid (left), a servo (middle) and to control the FrankenPrusa's (5.4.2) hotend heating element and part cooling fan (right).

etc. We can also use each leg independently to drive separate uni-directional loads (each being used as a "low side switch").

On the HBridge knuckle, I included a connector for the HBridge's outputs, and also an input for a thermistor (to read motor or *other* temperatures), and a *generic* three-pin connector, hooked up to an ADC - we often simply want *one* other pin to connect to the world.

I used the HBridge in three contexts. To drive solenoids for the CNC Xylophone's Mallets (2.7.10.1), we simply connect the solenoid as a DC load across two of the HBridge's outputs. To drive a hobby servo (which uses a DC motor), we connect the HBridge as intended to the two motor leads, and can use the generic three-pin connector to read the servo's position feedback encoder (which is a simple potentiometer). In the FFF 3D printers in this thesis (5.4.1 and 5.4.2), I use one leg of the HBridge to run the nozzle heater, the other half to run the Part Cooling Fan, the thermistor input to read the nozzle temperature, and I use the 5V that is exposed on the generic three-pin header to power the hotend's *cold side* cooling fan.

Because we define machines "across" firmwares, we can also mix modules' functions across devices. For example the Prusa Core One, which becomes the FrankenPrusa in this thesis (again, 5.4.2) has a filament detection sensor (a GPIO, switch) and a hotend *cold side* thermistor (to detect heat creep). The HBridge, which I had tasked here to control the hotend peripherals, was out of spare connectors in this case - but I could simply wire those two sensors into the motor controller's spare inputs. We can then easily rename these inputs in software to reflect their physical configuration. Again, this flexibility helps us to avoid developing new circuits while we go about building new machines.

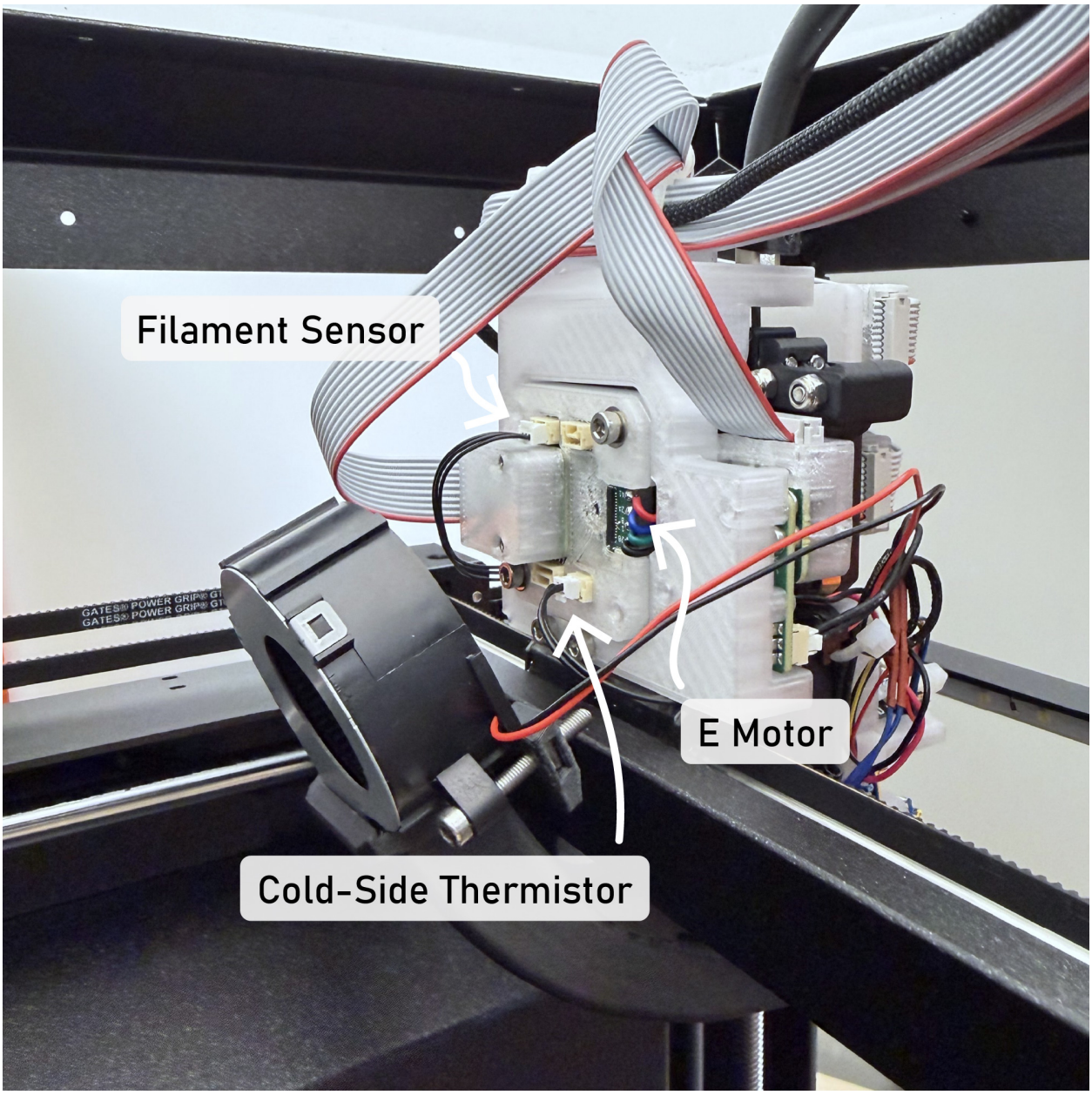


Figure 3.18: The FOC Stepper's GPIO and thermistor plugs used to interface with the FrankenPrusa's filament detector and cold-side thermistor, respectively.

I was able to use the Deadbugger in two contexts as well: to drive hobby servos as they are intended (through a PWM interface), and to control spare peripherals on the RheoPrinter: a line laser and an LED driver, both used for print inspection.

In some of these cases, we can write device firmwares to anticipate re-use by keeping the firmwares themselves *stateless* — just exposing their low-level interfaces so that inputs and outputs to those interfaces can be configured or renamed appropriately in software. However, some require lower level changes. The simple solution to that issue at the time of writing is to modify the firmware for each new context, redefining the hardware as a new device. This is at least simpler than

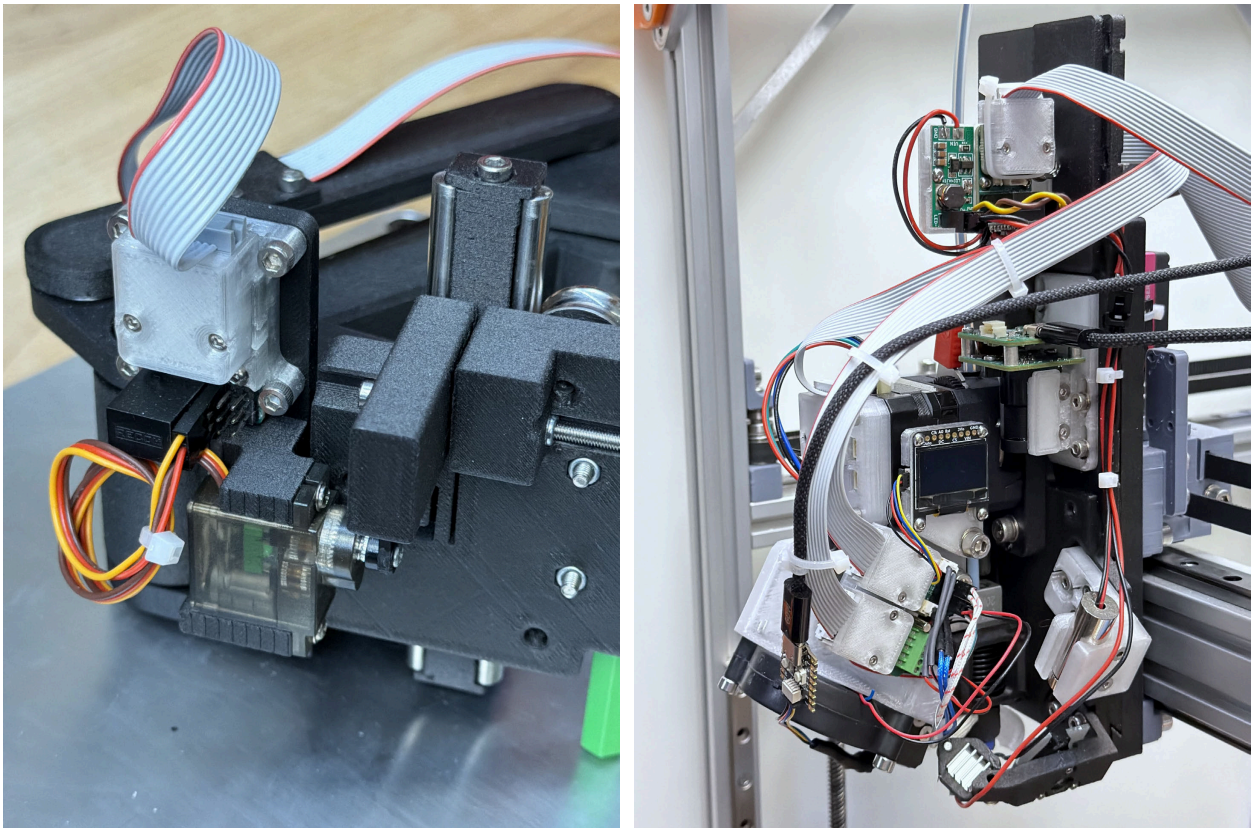


Figure 3.19: The Knuckles Deadbugger used in two contexts: to drive the Little-Guy's Z-Lifting servo (left), and (right) to control an LED driver and a small line laser module in the RheoPrinter (deadbugger mounted near the top of the end effector, beside the LED driver (green circuit, top left) line laser (silver cylinder) and LED (square with aluminum heatsink) near the tip).

authoring an *entirely* new firmware, as the networking code and hardware layers are the same: we essentially just reconfigure the device's API. This serves well enough and has been a productive tool during my time developing machine controllers, but in the future I would like to expand PIPES to enable on-the-fly reconfiguration of embedded codes *using dataflow configurations* as I discuss in Section 2.9.2.1 and as I explored in my masters' thesis [167].

3.6.2 Retrofitting Machines

I used the Knuckles kit to control my own hardware, but also to retrofit the Prusa in Section 5.4.2. Besides allowing me to compare my printing workflow to the state-of-the-art (5.10), it also gives us a one-to-one comparison in the practice of systems integration: an established product design vs. an ad hoc assembly of modules.

I was surprised that the modular set of controllers, which we would expect to mean overall *more* controller, is about equal on total board area. I think a lot of this has to do with wiring: most of the Prusa's control board area is actually taken up by connectors. The same is true of ours. On two-sided boards especially, it is in many cases easy to saturate board area (and definitely board perimeter area) completely with the desired connectors; whereas IC's and passive components



Figure 3.20: The FrankenPrusa a Prusa Core One FFF 3D printer that was retrofit with controllers from this thesis, using the Knuckles control hardware.

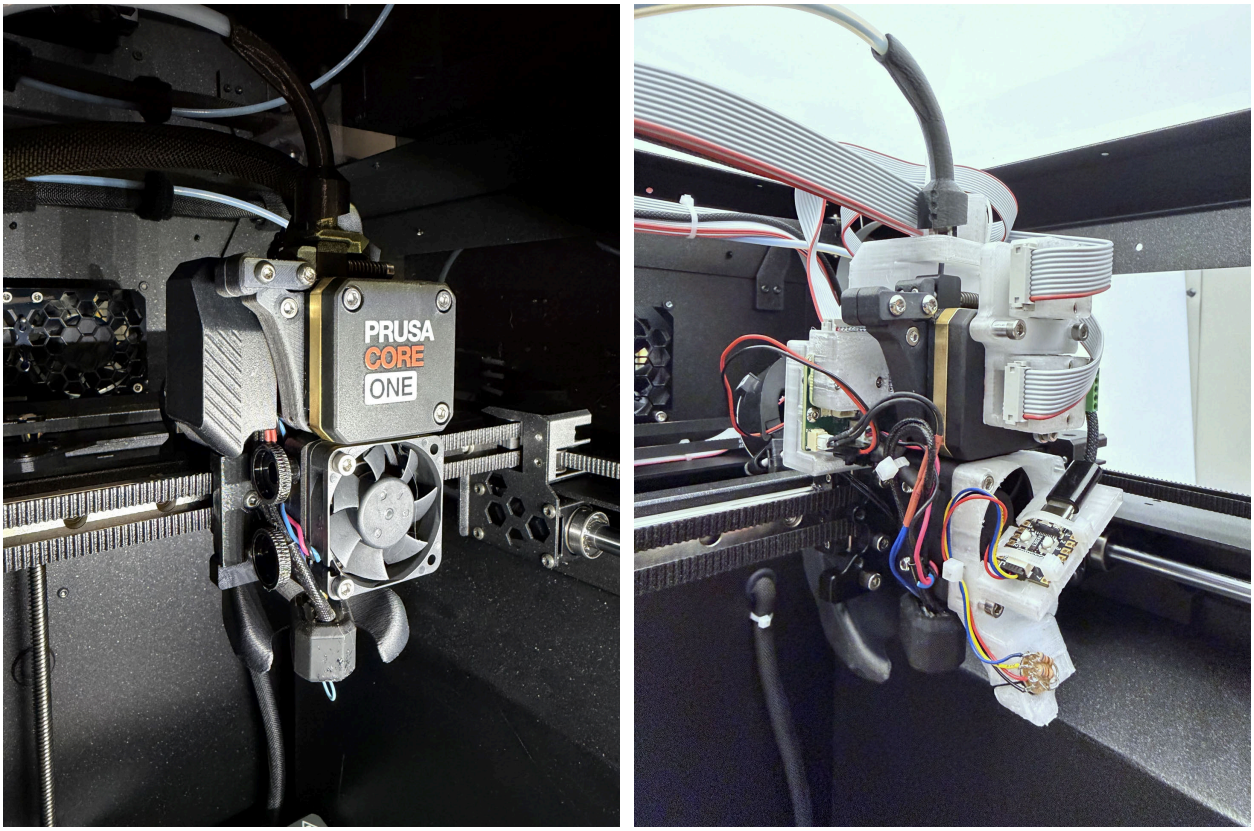


Figure 3.21: The original (left) and retrofit (right) Prusa *Nextruder* Printhead

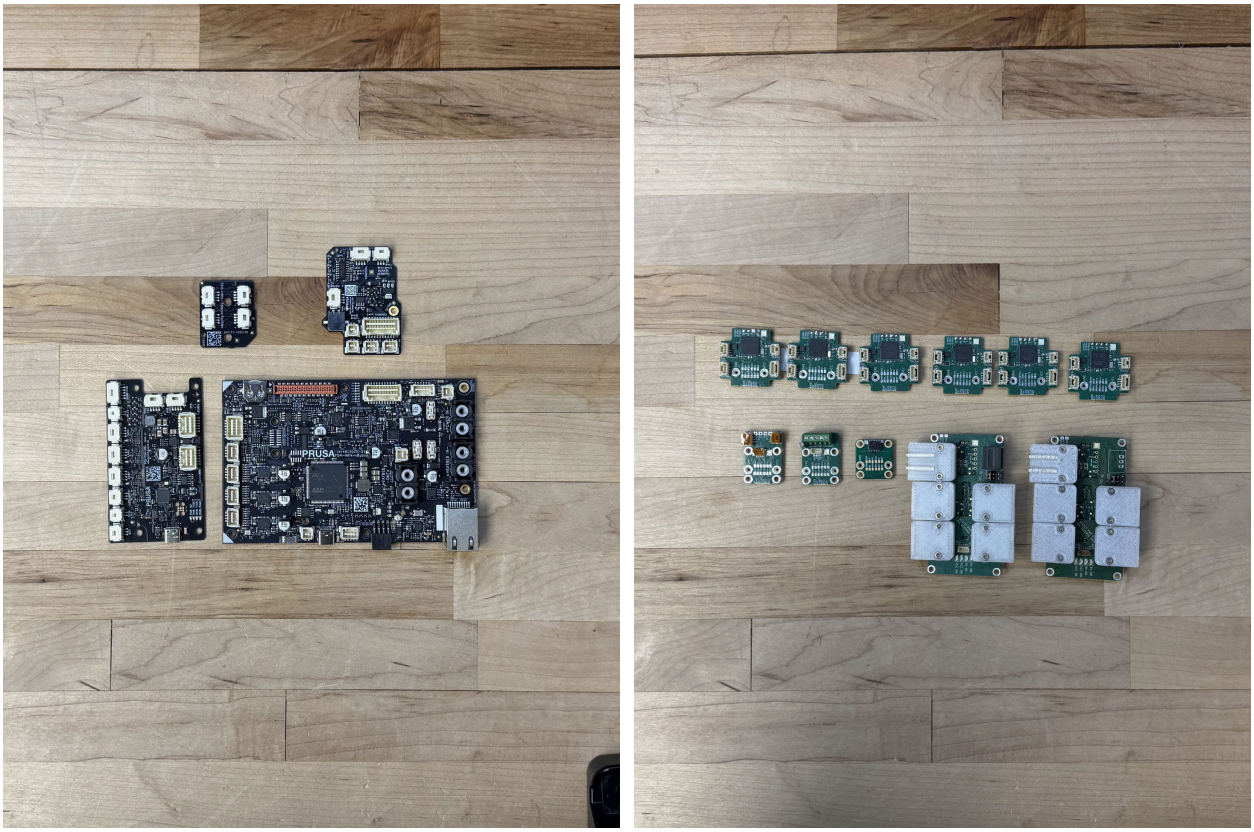


Figure 3.22: Control boards removed from the Prusa (left) and added (right).

continue to shrink, connectors cannot become vanishingly small because we need to manipulate them with our human-sized fingers, *and* because contact resistance of two metal blades has not fundamentally changed, leading to size limits that emerge from heat dissipation and management in power connectors.

The rear end of the retrofit (above) is certainly messier than the shipped product, but not egregiously so. I added to their power supply a bleeder resistor (the orange aluminum block) and large bypass capacitance (blue cylinders), to help linearize the power into the FOC Steppers. The chassis wiring removed (below) is a much more heterogeneous set than in our retrofit, where a network connector lets us homogenize the main wiring harness. This is where IDC connectors and Ribbon Cables shine: we can easily make polarized, terminated cables of any length without the requirement for time-consuming and detailed work of wire crimping.

However, this practice takes for granted that we relocate the control circuits to where the relevant hardware is, as in Figure 3.21. We do not entirely remove the requirement for "terminal" wiring (connectors below), where our modules connect to the plugs and wires that are integrated into the actual pieces of hardware we are controlling.

The actual *electrical interface* to the Prusa is quite thin: four connections to each motor, some heaters / thermistors, fans, and the load cell. The Prusa (and our controller) doesn't use limit switches, instead we simply run the servo into the machine's hard stops; a simple example of turning motors into sensors to reduce complexity.

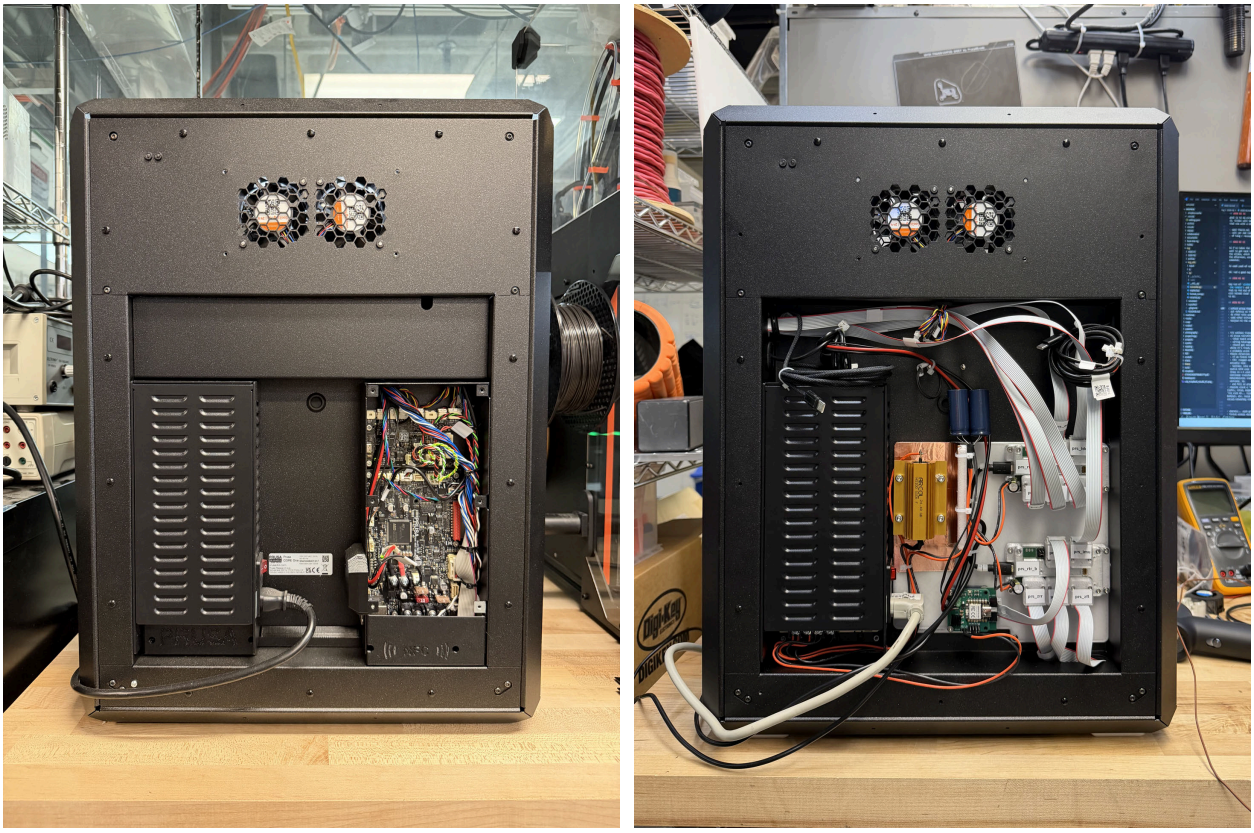


Figure 3.23: Control boards removed from the Prusa (left) and added (right).

The modular controllers let us take over the Prusa hardware with relative ease; I use their power supply and connect it to two Knuckles Hubs, which then each connect to motor controllers, heaters, etc. In all it took me about four days to disassemble the Prusa, design some additional mounting brackets for my circuits, characterize motors (4.3.1), and then re-assemble it and begin printing (5.10.1). Were this step to require that I develop a new circuit board for the task, it may have taken up to two weeks to design that board, one or two weeks to wait for it to return from the fabricators, and perhaps another two weeks to write its firmware. In the case of the Prusa, this may have been easier because their hardware is open source — however, their control board does not have enough processing power for the requirements of this project, nor does it have inputs for motor encoders, or motor *drive* circuitry that can be used in a closed-loop manner.

I also used the Knuckles to control the CNC milling machine from Chapter 6. This system was much simpler than either of the printers, requiring only one Hub and three motor controllers. It also eschews limit switches in favour of homing via motor sensing. In this case, I did not instrument or control the machine's spindle directly, mostly for lack of time.

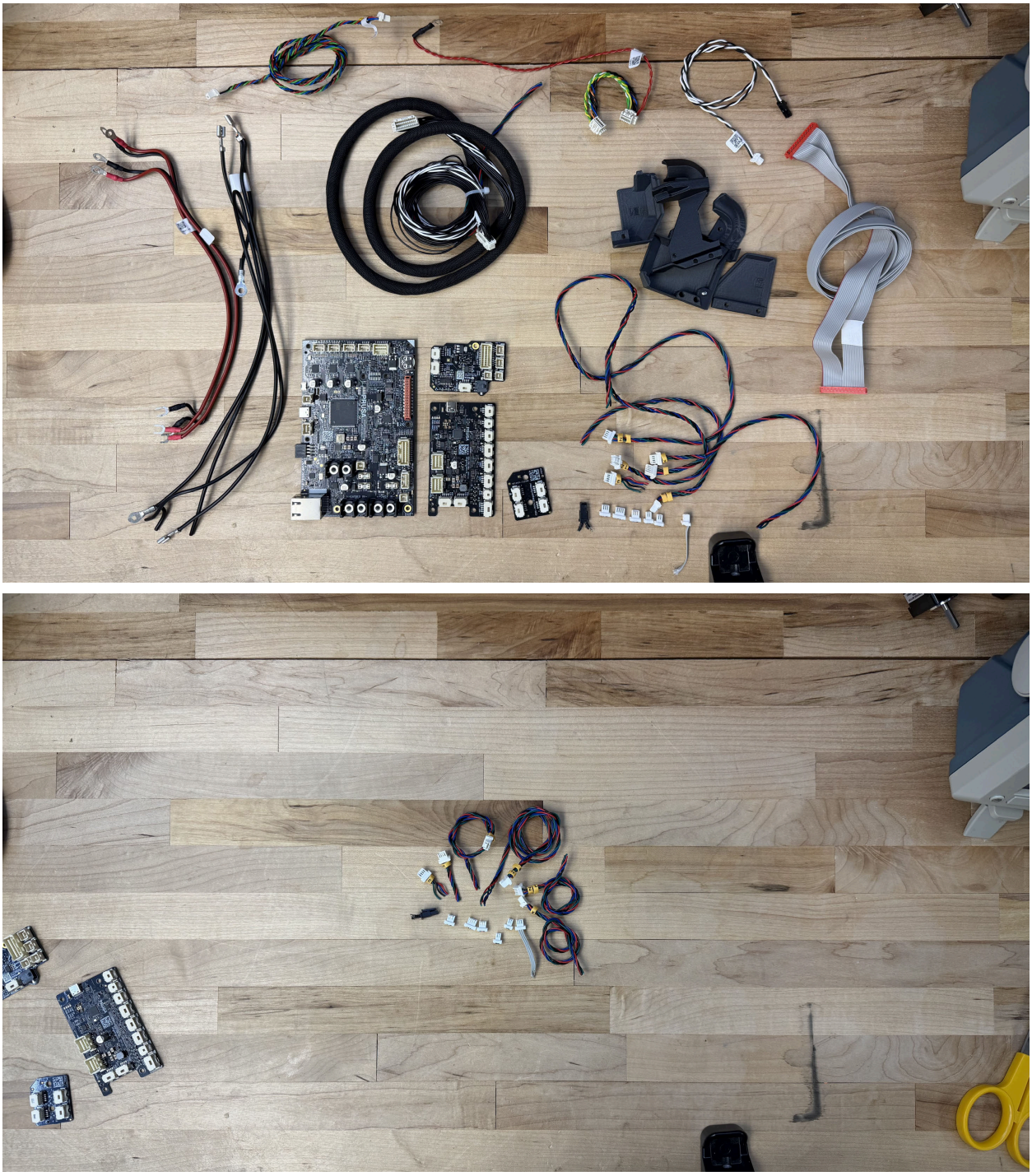


Figure 3.24: Wiring removed from the Prusa (above), and *just* the terminal connectors (below), i.e. plugs that go directly into hardware rather than into another control board or connector interposer.

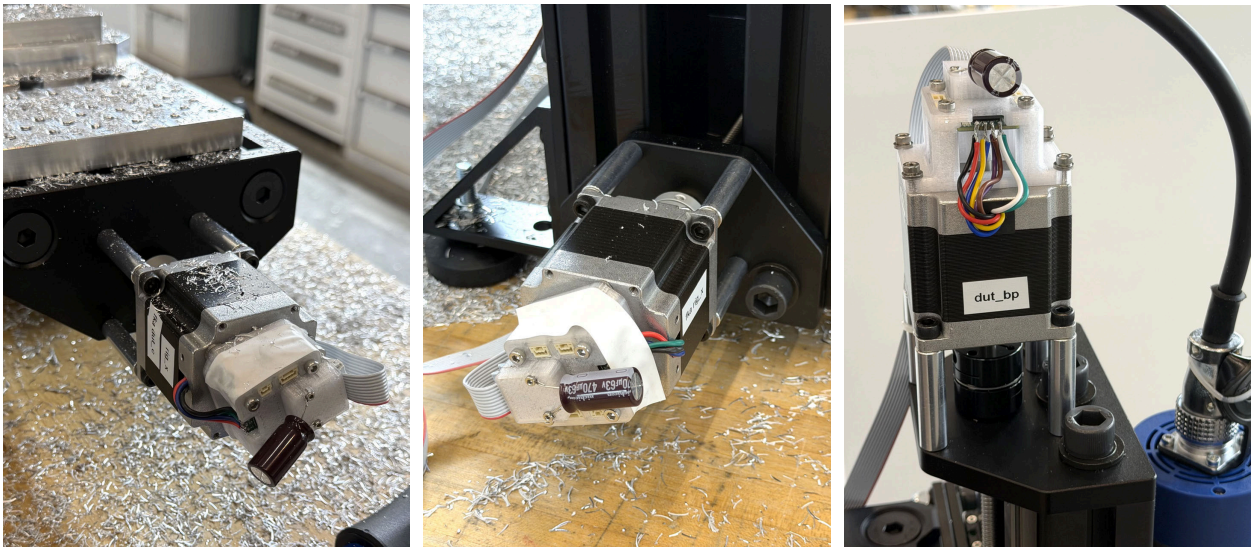


Figure 3.25: FOC Stepper Knuckles mounted on the CNC milling machine's X, Y and Z axes.

3.7 Discussion

3.7.1 The Hardware Partitioning Problem

In some ways, the arrangement of modular circuits and their wiring reflects the partitioning problem that I discussed in Section 1.4.

Here, we mount modules close to the devices they control. In industrial systems, it is more common to co-locate all controllers in an enclosure on DIN rails, and build wiring looms from that enclosure to the hardware itself. Although there are of course cases (e.g. encoders and some other sensors) where the location of some circuitry alongside the relevant hardware is required. Considering that, in some cases, the circuitry required to integrate with a device becomes *about as large* as the connectors required to plug into it (as in Section 3.6.2), locating controllers alongside hardware begins to make more sense, especially where tight integration reduces overall complexity. This would be driven by hardware component *suppliers* integrating their components with controllers. There is some emerging practice of mounting electronics *within* connectors themselves, as in USB-PD cables that are electronically tagged to verify their current carrying capability [209].

Motors are the most common scenario where this is true in the state-of-the-art: the device is quite large, and co-locating the drive circuitry with the hardware has the advantage of minimizing parasitic inductances and resistances that would otherwise exist in long cable runs between a servo controller and the servo's stator coils. There is also — as mentioned — the issue of the encoder.

3.7.2 About Power Routing and Delivery

In practice, routing power throughout a system can be as cumbersome as routing networks. For prototypes, can be valuable to have one-plug-for-both. In other systems, these concerns are often separated. Because power and networking are both heterogeneous, collapsing those layers would require (again) more unique modules. This is another layer to the hardware partitioning problem.

One of the valuable things that I did develop here is that the Knuckles Hubs also include power switches. They expose Pipes functions that can be used to operate those switches such that were the hubs are used also as power distribution boards remote legs of the system can be power cycled in software. This helped to provide a backup safety scheme and to e.g. automatically turn systems off at the end of a job. But merging all of these layers presented some conflicts as well, for example to ensure that network discovery steps work, a script that looks for devices on the network must also know to turn those devices on via the hub.

This is an issue again when systems are deployed at multiple scales. Whereas the hub can manage power requirements from most small machines' motion systems, it is infeasible to imagine that it could supply and manage enough power to run the spindle from the CNC machining work in Chapter 6 (which consumes multiple kilowatts of power).

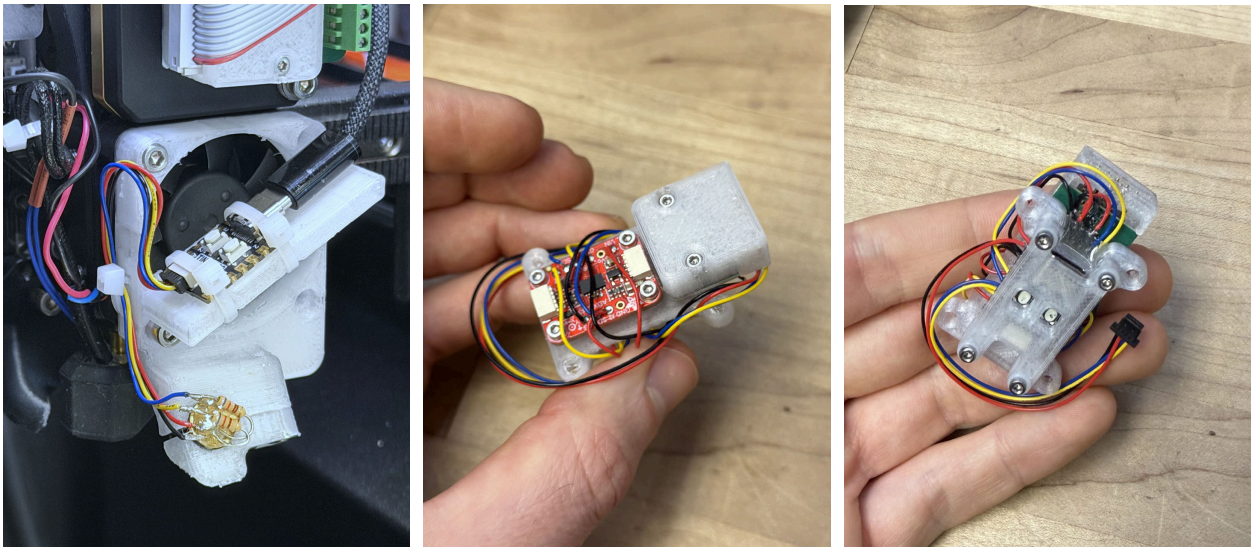


Figure 3.26: The microbolometer on the FrankenPrusa hotend (left) and a prototype accelerometer module. Both were new devices that I integrated in an "ad hoc" manner.

3.7.3 About Hackable Systems

Although the Knuckles represent my attempt to design a set of circuits that would cover the hardware requirements in the systems that I developed here, there is an inevitability of unforeseen requirements in systems assembly. We should aim to develop systems that can easily accommodate these ad hoc inclusions, following the end-to-end principle. This was a concern in OSAP, PIPES, MAXL as well.

One error that I made in the design of the Backpack interface in this regard was not including *breakout-board* equivalents for either end of the interface: one that could mount to modules and let us solder ad hoc networking technologies onto it, and another that could attach to the Backpack allowing us to integrate ad hoc modules to our existing network links.

3.8 Future Work

I took on the development of these modules because I found that existing open source hardware was not modular at the level that I wanted to explore. Some industrial controllers are modular, but they are not open, so I would not have been able to develop new computing systems for those.

These modules do not constitute the complete set of hardware that would be required to realize the broader goals of the thesis, i.e. extending across industrial sectors into all manner of machine control. That is an activity that is taken on by much larger groups of individuals and firms. That is also why I developed most of the systems integration tools in Chapter 2 to be software defined; they should not require that we modify hardware in order to integrate it into new systems.

So for future work in this chapter I think that a key future goal is to enable others to more rapidly include networking layers in their embedded devices, see Section 8.3. If anything, I would expect

that the *Backpack interface,* (or something like it) would be the most productive component of this system to develop. Module themselves will be heterogeneous because they interact with the world, networking is less so. The proposal in the Backpack interface is to enable networking / physical layer experts to develop those circuit components and then *device experts* to easily drop those into their projects. For example having designed a motor controller or sensor, good open source Backpack designs may help them to easily add isolated power delivery, battery-powered capabilities, etc. A similar pattern exists in GPIO interfaces like the Xiao microcontroller modules from Seeed Studio. They enable circuit designers to eschew the complex microcontroller-care-and-feeding steps in their circuit designs, and focus on application circuitry.

4 Motion Control with Machine Models

4.1 Introduction

In this chapter we will depart from the world of hardware and systems architecture and move towards applications in motion control and planning. Before we get started with anything else, a quick disambiguation of terms. *Motion control* can mean many things. Depending on who you ask, it may be the practice of:

1. Tuning a servo's PID[66] gains to generate smooth and precise response to target positions or velocities.
2. Generating single-segment acceleration "ramps" aka *trapezoids* to smooth motion of an individual axis.
3. Optimizing motor commands in a bipedal, legged, or 6-dof robotic system.
4. Generating target curves for a robot or machine to follow, navigating through space.

The field of robotics (of which machine control is a subset) is a mixed bag of all of these activities (and more!). In this thesis, "path planning" is the step where path *geometries* are defined. That is the purview of slicers and CNC CAM tools; I do not work on this layer explicitly. For "motion control" I mean two things:[67] first is the task of "velocity planning" where we already know the path we want our machine to take (our "target trajectory") but must pick the velocities that it should use along that path, generating a "planned trajectory." Given those trajectories, we must actually follow them: this means motor control. Of course there is a natural connection between all of these layers; constraints from motor control and machine kinematics relate directly to velocity planning, but they should also be reflected into path planning algorithms. That remains out of the scope of this thesis, but I do discuss how the work here could be extended into that layer in Section 7.2.2.2 and 7.7.1.

So, about those constraints. Motors produce limited torque, structures deflect with load, polymer extrusion is limited by rheological (and thermal!) dynamics, cutting forces are limited by tool and part stiffness, etc. Implicit to CAM programming is that users should set parameters such that their GCodes don't require the machine to violate steady-state constraints (1.3.1). But what about the dynamic limits? GCodes describe *target* trajectories (see Figure 1.7 in the earlier discussion of the *hidden optimization 1.3.3*) and completely faithful execution of those trajectories would mean making instantaneous changes in velocity. In real-time machine control (where those GCodes are turned into actual motion) we use velocity planners to avoid exceeding dynamic constraints, for

[66]PID, for *Proportional-Integral-Derivative* controller, is a pattern that I will assume readers of this thesis have seen, built, or tuned. If not, never fear: there are excellent explainers about, my favourite of which is here: [210]. They are simple (ish) and surprisingly effective feedback controllers that are found *all* over the place - in simple and in complex systems, slow and fast, etc. You will also see i.e. the noun *PIDs* in referring to the controller's *gains*.

[67]Well, there is the third component to this problem, which is the systems-level implementation of these controllers that I already covered in Section 2.5.

example constraining instantaneous changes in velocity by imposing an acceleration constraint. This is the hidden optimization that I spoke about in the introduction, and it takes place *beneath* the GCode layer, e.g. in a firmware on the machine's control board.

Perhaps the critical word here is *planner;* these account for machine dynamics that must be anticipated, i.e. their main concern is to ensure that a machine starts slowing down well ahead of a corner in order to avoid the sudden change in direction that would over-extend motors and excite the machine's structure. They are often called "look-ahead planners." To disambiguate time scales, planners typically operate over a horizon of tens of seconds[68] whereas a motor controller's main loop may run ten times in every millisecond; recall Figure 1.9 for a map between time scales and control components.

In Chapter 2, I showed how we can bring these planners *up* out of machine firmwares and make them into software objects. That roughly solves the problem of the hidden optimization simply by relocating the algorithm into a higher level system where it can be more easily modified and inspected / observed. But how should we actually describe and operate those planners?

4.1.1 Key Limits to Classical Velocity Planning

MAXL includes a trapezoid-based planning block (2.5.4.4) which implements a common style of velocity planner that I explain in more detail later on in this chapter (4.2.4). These form the basis of the state-of-the-art in machine control, but they suffer from the same similar structural issue that is present in CAM workflows: they are configured using parameters that describe their outputs rather than with models that describe their constraints. For example trapezoid planners are configured using parameters for maximum velocities and maximum accelerations[69] $(a_{\max}, v_{\max})$ for each of the system's degrees of freedom. Those parameters are *related* to physical models for motion, but alignment between those parameters and the real underlying physics must be done by hand. To say it again in one more way: these types of planner require that machine builders work the problem *forwards* (from models or intuition to limits) rather than *backwards* (from constraints to planners and solutions).

For a first order mapping between machine physics and these motion parameters: maximal motor RPM and machine friction constrain speed, motor torque and machine mass constrain acceleration, and jerk is limited by the permissible rise time for current in a motor's stator coils (see Section 4.3.1). That gives us about one physical constraint per motion parameter, but there are other considerations: backlash, stiffnesses, vibrations, and externalities like temperature (see 4.2.2 for more notes on each). Really, each of these aspects need to be considered together to understand our machine's total motion constraints. Collapsing them all into a single set of parameters is

[68]Planners can alternately be configured to look ahead through some amount of *space,* which under acceleration and velocity limits can be pretty well correlated to time. For example automobile braking systems are defined by their *stopping distance* rather than stopping time. Some GCode interpreters though look ahead through a fixed *number of linear segments,* which is not necessarily correlated to either as I briefly touched on near the end of Section 2.2.1.

[69]Sophisticated trapezoidal planners also limit *jerk,* acceleration's derivative.

limiting: we may reduce velocity overall to avoid vibrations that only appear under very particular conditions (i.e. where our machine is going fast *and* exerting high forces on the work piece), or limit acceleration and jerk overall where we really want to decrease low-end motor loading.

Defining these limits outright (one maximum acceleration at all times) is also restrictive because the underlying constraints change based on many of the other (dynamically changing) states. For example if you take a look at a motor's torque curve (e.g. Figure 4.4) you can see that they can exert far more force (bigger accelerations) at low speeds than at high speeds. To maximize performance we would want our planners to be able to exploit that low-end torque!

Motion control is also coupled into process constraints, but trapezoid planners do not account for these naturally. Relying on CAM configurations to encode those limits works for steady-state operation (besides the configuration burden) but the couplings can impact dynamic operation too. Because trapezoid planners are based at their core on the kinematic equations it can be difficult to modify them to accommodate for other physics. This means that we either need to make structural changes to trapezoid planners when we deploy them on new processes (in Section 5.2.3 I show how this is done for 3D printers) or build compensations and modifications to their outputs that work "on top" of them, rather than at their core (in Section 4.2.6 I cover a few examples of this strategy).

But what's the actual difference there? Even though these modifications enable trapezoid planners to better accommodate process physics, they cannot anticipate them or reflect their constraints back into the motion optimization. Instead, process control must follow along with whichever planned trajectories were chosen based on kinematic constraints alone. For two clear examples of this: realtime CNC controllers do not model cutting force, and so cannot anticipate that they may need to slow down before they start a new cut (we use feed rate selections in CAM for that), and 3D printer controllers do not model polymer melt flows, and so they do not know where they may need to slow down in order to avoid overpressure conditions in the nozzle (again, we use slicers for that).

4.1.2 Chapter Overview

4.1.2.1 Key Questions and Goals

So, this chapter answers one of the leading questions from the introduction: *how can we re-frame machine control explicitly as a constrained optimization problem?* Trapezoidal planners *are* rudimentary constrained optimizers, but they don't operate directly on model-based representations of those constraints, so a more accurate framing of the question at this point would be to say "how can we re-frame velocity planning as an explicit optimization *over model-based constraints?*"

The key goal then is to learn how to do this: existing frameworks are limited in their ability to describe arbitrary physics primarily due to computational constraints. So, can we leverage new computational technologies to make a planner that removes these constraints with brute force? That means parallelization, so how exactly do we formulate the planner in a manner where the

computations are matched to the compute hardware, learning from other state-of-the-art parallel formulations? What are the limits and tradeoffs; do we trade one set of machine-related parameters for another set of planner-related hyperparameters, and how much determinism and observability do we give up for the new capabilities that it unlocks?

We will of course need some models. Amongst the many modelling techniques for kinematic systems and actuators, we need to select the formulations that suit the planner best. We would also like to use the same set of models for parameter fitting and for operation — this would let us share computational representations across multiple parts of the same complete system. We also have the goal of fitting those models to data from the hardware that the controller will operate, and fitting them in a way that would allow us to update them with new data from regular machine operation. While none of these modelling techniques are new research, understanding how each of these concerns informs those model choices is an interesting trade study.

It is difficult to capture all the relevant machine physics, especially when we are just beginning to fit them on new hardware. So, where models fail to capture important dynamics, we will want to develop handles into the planner that let us encode heuristics that can still modify behaviour of the system appropriately.

Finally, if we can pull this off, can we also formulate the planner so that its internal states (which constitute as complete a simulation of the hardware as our models would allow) can be used to learn about our machines and processes themselves? In that case, what might we learn, and how do we render and distill those outputs, or combine them with other measurements to make new estimates of values that would have been difficult to measure directly?

4.1.2.2 Background Sections

I look closer at state-of-the-art motion control in Section 4.2.4 (which explains in detail how trapezoid planners work) and then 4.2.6 covers how these are extended and improved by leading industrial motion control firms.[70]

It is fairly well known that model-based control has direct advantages over the state-of-the-art; much progress (particularly in robotics) has been made in advancing these techniques recently. I cover key methods and insights from that field in Section 4.2.7, also discussing how the planner that I develop here differs from common formulations.

There is less research on model based control for machines. One reason may be that machine control typically requires higher bandwidth than robotic control (machines must locate tools more precisely than a robot dog must position its feet). Optimization based controllers are also less inspectable and predictable than heuristics based planners — this means that engineers who care primarily about their systems' reliability tend to prefer the latter, as I discuss in 7.2.4.

Based on the contributions that do apply these techniques to digital fabrication equipment (covered in Section 4.2.8), researchers seem to be limited by GCode interpreters and state-of-the-art machine architecture more generally. This is interesting in itself with regard to the earlier discussion on the

[70]Even though it is difficult to ascertain really how these firms' algorithms work because their secrets are kept quite close to the chest.

partitioning problem (1.4), but also provides some more motivation for the work from Chapter 2 on systems integration: eliminating architectural barriers seems like an important step towards helping these researchers develop smarter controllers.

I also cover background on fundamental and then more sophisticated machine physics (Section 4.2.1 and 4.2.2), and a brief note on reinforcement learning (RL, 4.2.7.2), which I do not rely on or implement directly but which remains relevant primarily as we look ahead to the future of control *in general* and its application to machine control.

4.1.2.3 Key Methods and Contributions

Rather that inventing a litany of sideband corrections, compensations, and parameterizations of these underlying physics and couplings, we can try to develop a velocity planner that uses models more fundamentally as an input to describe constraints. That challenge makes up the contents of this chapter.

This requires a few key methods. Firstly, we need to develop and fit models can will articulate motion constraints. I cover models for electric motors and for kinematics (transmissions, inertias, and friction) in Section 4.3. To fit these models on the hardware where they will be used (4.5), I develop a closed-loop stepper driver and a method for learning some of its key parameters automatically (4.4). I assess the quality of our models in Section 4.7.2 and show how they can be iteratively improved using data from real-world machine operation (4.7.3).

In Section 4.6 I develop a motion planner in an optimization framework that uses these models directly. It has the simple task of maximizing speeds while respecting actuator and kinematic constraints. Doing so with high fidelity models for motion is computationally expensive; I lean on a set of tools that have been recently developed for parallel computing on the GPU, and carefully frame the computation such that it remains parallel in structure (4.6.1).

This planner only uses models of actuators and inertial / friction dynamics, I still rely on heuristics to modify the controller such that constraints which are *not* described explicitly are not violated, but those heuristics are encoded in a way where they work "through" models, as I discuss in 4.8.5.2.

However, the formulation of this controller differs from the state-of-the-art then in a few important ways. The first is that it is carefully set up such that it *can* run in parallel on a GPU, which greatly improves its compute performance and allows us to describe more process physics *alongside* motion optimization alone, which I show in Section 5.9.

The second is that it uses models for motion more *directly* than other formulations; normally these are compiled and run in hardware, whereas our formulation runs in the same software where the models are fit: this lets us use the same computational representation for both tasks and also for high level planning steps (as in Section 5.8) and for machine design (6.4.2).

Using models to describe constraints is helpful because we can build those automatically from physical data: e.g. to set motion parameters optimally we would first want to build models anyway, why not use those directly? This is not just a convenience: in Section 4.7.1 I evaluate the planner

against a trapezoid-based equivalent and show how it increases the dynamic range used by our actuators to improve overall speed.

The planner's formulation has the benefit that we can use internal states from the planner to understand *which constraint* limited performance at any point in a planned trajectory, relating models to operation. In Section 5.11.1 I show how this can help us to understand machine design choices more directly. In Section 5.11.2, I expand on the same method to discuss how planner outputs (alongside process models and model-based heuristics) enable a more direct tuning relationship between user-selected values and fabrication outcomes. These values can also be used in combination with realtime measurements to compute new data like machining cut force estimates (6.3.2).

Section 4.8 summarizes the major lessons learned in this exercise and 4.9 discusses some of its major shortcomings and how it may be improved in the future.

4.2 Background in Motion Control

4.2.1 Essential Physics for Motion Control

At the first layer, the physics of motion control is just *inertial;* to move our machine's masses we must apply forces that accelerate them — changes to acceleration update velocity, and then position: $F = MA$. We do so with motors (aka actuators, which have inertias of their own), whose relationship to our machine's moving parts is described with kinematic models. Of course there is also friction in each of these systems. I cover those basic physics and the models that I use in this piece of work to describe them in Section 4.3.2.

The actuators also introduce constraints: electric motors cannot instantaneously change the amount of force that they generate because they are essentially electromagnets, and it is impossible to instantaneously change the amount of current that is generated in motor windings. This introduces a constraint on our motion systems' *jerk,* which is the third derivative of position (the next three are *snap, crackle, and pop*). I cover motor models in Section 4.3.1, and discuss how these are connected to one another in 4.3.3 such that we can estimate the current requirements — and limits — for our motors' actuators given any velocity and acceleration at the machine's tool tip.

4.2.2 Sophisticated Physics for Motion Control

I will show how kinematic and motor models can be used to optimize motion control directly in Section 4.6, but I also want to describe here some finer grained physics that are relevant for motion. I was not able to manage all these in detail in this thesis (I discuss the value of *heuristics* to overcome that loss in Section 7.2.4), but they are still relevant.

They also motivate our formulation of a planner that would enable us to more rapidly incorporate *new physical models* into a framework where models could be fit to data, simulated, and controlled all using the same core representations. This requires that we apply *big* computing to the problem because models for these phenomena are more complex, time-dependent, and may change over time, aspects of which I demonstrate primarily in Section 5.9. I discuss some future work on the topic in 4.9.3.

These phenomena are also worth understanding so that we get a finer sense of why *in-situ metrology* is valuable: some of these physics change not only because of the machine's own construction (which a manufacturer may be able to control), but due to their particular site of use, and vary also day to day. In many cases — as I discuss in Section 1.3.3 — GCode interpreters prevent us from studying these physics "as they happen" on our machines, motivating us also to build our systems such that controller states themselves can be used as we develop and discover these models.

4.2.2.1 Backlash

Backlash (sometimes just called *lash*) happens where actuators are coupled into machine axes in a manner where there is some dead band between changes in their position and changes in the axes position.[71]

For a simple explanation, this happens when there are even slightly loose connections between the actuator and the moving component. The most common example emerges from gear trains, where a driving gear's tooth is not completely meshed with the driven gear; when the driving gear changes direction, it must first move to close the gap from one tooth to the next before it can actually drive the output gear. The same is true of some belt driven systems and some lead screws, rack-and-pinions, etc.

Backlash is exceptionally common but difficult to model because it is dependent on motion profiles' history and on external loads like gravity, forces from acceleration, and forces from process phenomenology. It is also nonlinear, as we can see.

4.2.2.2 Nonlinear Kinematics

The machines that I deploy using my own dynamics planner (in Chapter 5 and 6) both have *linear* kinematics, meaning that their tool-tip position is just a linear combination of their actuator positions (and vise versa). The CNC machine is the most straightforward: each actuator is attached only to one axis, whereas the 3D printer uses *Core XY* kinematics [176], where two motors (a and b) work in parallel to control the machine's x and y axes (see Section 2.5.4.6). Linear kinematics are simpler to model because their dynamics don't change significantly when their position changes, whereas in a nonlinear machine (like a robot arm, for example) they do: the robot has more rotational inertia when it is outstretched than when it is not, and also requires more static torque at the base to balance against gravity.

[71] There are better technical descriptions of this, but any time you can wiggle a machine axis without changing the position of the actuator, you have backlash.

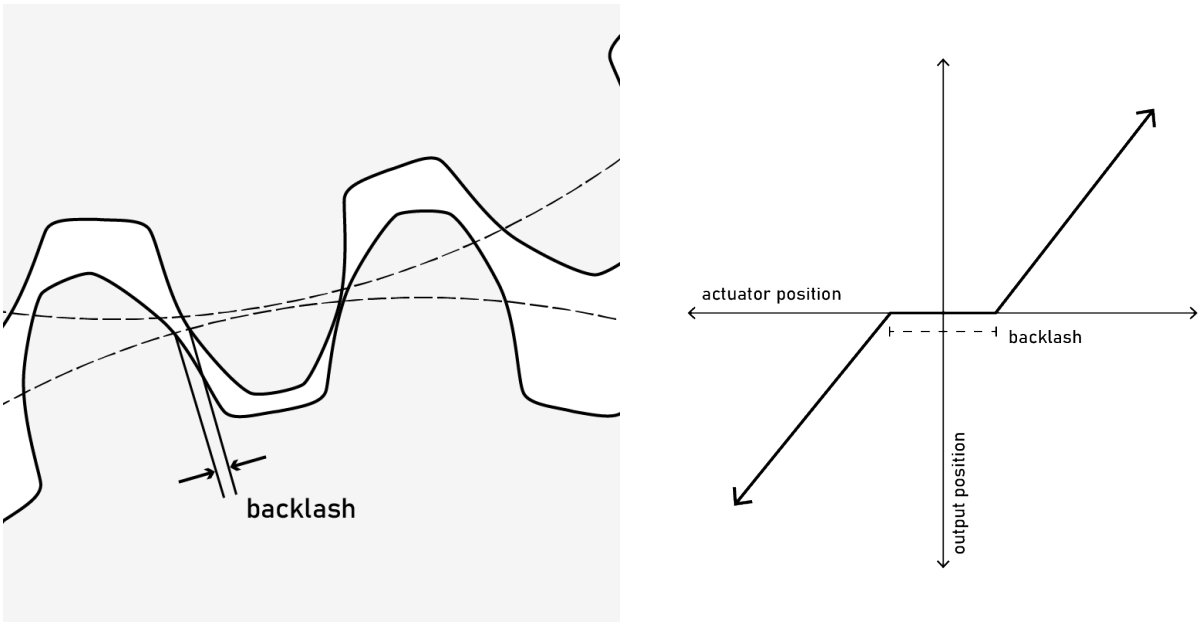


Figure 4.1: Backlash is *strange behaviour around the origin.* On the left is one practical example, on the right is a diagram of the driving gear's position (X axis) and the output gear's position (Y axis). In a zero-backlash system, this would be a simple linear plot, but in reality there is normally some dead band here.

Machines with linear kinematics also have the property that their associated mass and friction terms are simply mapped onto the machine's acceleration and velocity in cartesian (tool-tip) space, whereas defining the kinematics for a machine with nonlinear kinematics would also require that we define motion of intermediate masses and frictions. Take for example the `little guy,` in Figure 4.2, a machine that I developed partially to understand these types of system in more detail.

It is a drawing machine that uses a common kinematic design pattern for robot dog legs, using two rotary actuators linked in an arm, each controlled by a motor that is rigidly mounted to the chassis. Kinematic arrangements like this can have serious advantages: they reduce total moving mass, and they package well (keeping the robot small while its work volume remains large). In this case the motion of the two masses in the system are not in the tool-tip's cartesian space, nor are they in the actuator space — each traces its own path through space. The relevant friction terms (related to rotational speeds at each joint) are also intermediate.

I did use MAXL to control this machine (see Section 2.5.4.7), but that layer only requires that we define the inverse kinematics. I did not extend the dynamics planner in this chapter to account for the nonlinear *dynamics* of those moving masses and frictions. Instead, I did what many machine controllers for these systems do: apply maximum accelerations and velocities for the tool tip using a trapezoidal planner and tune those such that the underlying dynamics seem OK. In Section 4.9.3.1, I explore how the kinematic definitions that I do use in this chapter could be extended to also return these intermediate values and incorporated into the planner.

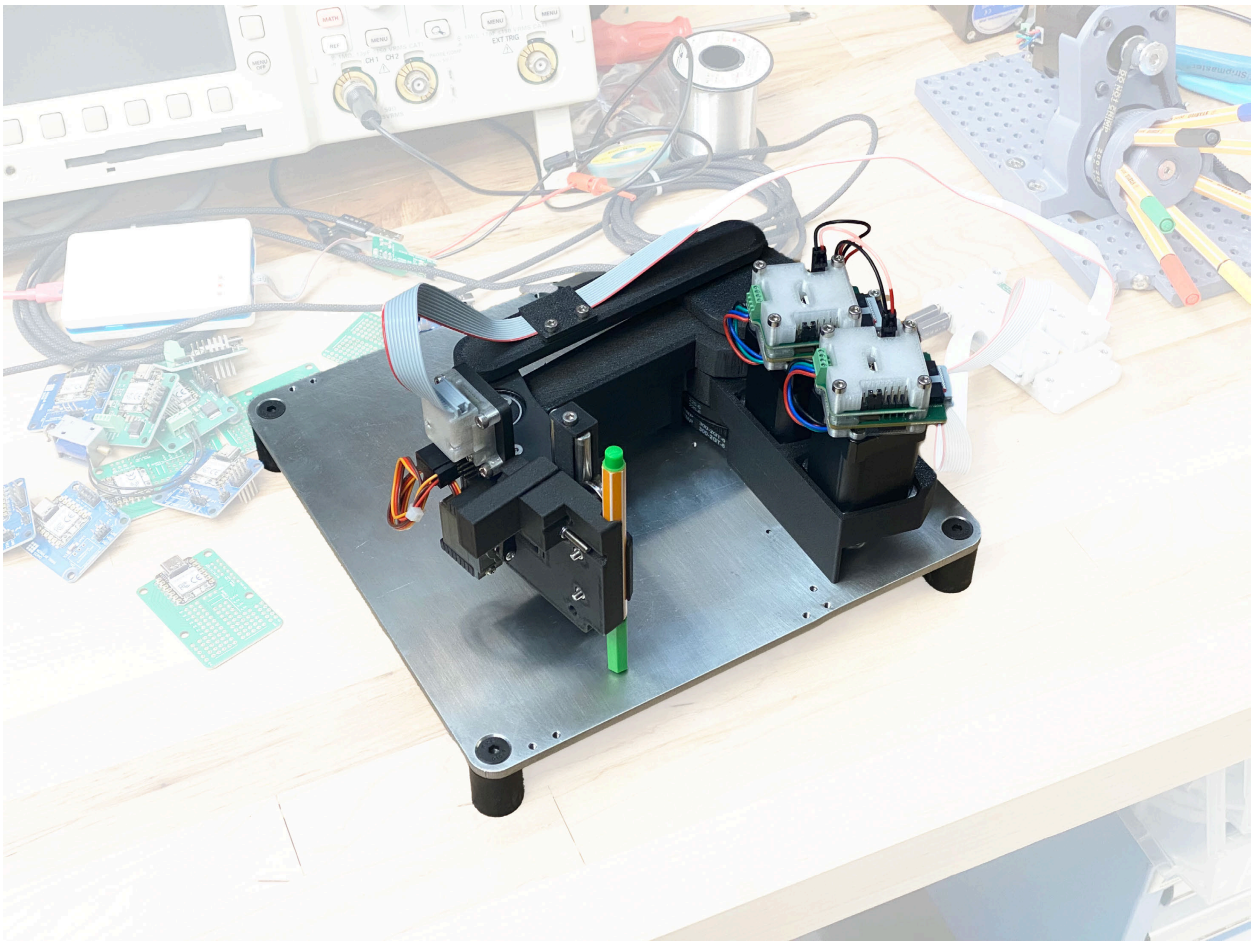


Figure 4.2: The `little guy` machine, and its nonlinear kinematics where motion of the tool-tip in cartesian space is composed of arcs traversed by the machine's two arms, which are connected to one another with rotary joints.

4.2.2.3 Stiffness and Resonant Excitation

Beyond actuator limits themselves (which are effectively force limits), machines also *flex* when we exert those forces. For example one of my old mentors once told me *"there's no such thing as a zero backlash drive."* He was referring actually to machine stiffness: even with a perfect coupling between our drive system and machine axis, the transmission elements will transition from positive, to zero, to negative preload as we change direction — and that preload deflects machine hardware.

So, ideally we would also develop models for machine stiffness that would tell us how much they would deflect under these loads because we want our machines to be precise: if the forces we exert in order to move them around cause them to deflect significantly, we will *miss the point!* Stiffnesses can be more challenging to model than actuators, inertias, and frictions alone because they can change as the machine's configuration in space changes, for example a machine with a cantilevered beam will deflect more when it is extended than when it is retracted. They also require that we measure the *actual* tool-tip position rather than measuring it indirectly with our motor's position estimates (broadcasting those through kinematic models); the kinematic chain between

our actuator and the tool tip is what is flexing, so we need some way to measure that displacement directly.

Many machine tools can be quite stiff, for example linear rails (a common motion component) have stiffnesses that are often measured in Newtons per *micro*meter of deflection ($N/\mu m$). This is one of the reasons (the other being that direct measurement of tool-tip position adds the complexity and expense of adding another set of encoders to the system) that it is common to measure machine stiffness indirectly using accelerometers.

This is done by exciting machines with sinusoidal or step inputs and fitting parameters for stiffness, mass and damping in the frequency domain. I developed MAXL's chirp block (2.5.4.3) roughly for this purpose, but ultimately ended up using it to fit friction parameters alone, as I will show later on in this chapter (4.5.2). In any case we are interested in stiffness *largely* to cancel out vibrational effects[72] on our machine, and so analysis in the frequency domain can be applied directly.

The frequency response of a machine can change significantly due to apparently small or unseen physical changes, e.g. when timing belts becoming slightly looser or tighter, or when masses change (which happens also as printers deposit material, milling machines remove material, or we modify components — for example adding new sensors to a tool head). Even temperature — a longer discussion of which comes up shortly — can change the stiffness of a belt or the preload (friction *and* stiffness) of a linear rail.

4.2.2.4 Excitations from Actuators and Transmissions Themselves

Besides exciting resonant modes of our machine structures by moving it around, there are a few other noteworthy ways that machines can be structurally excited.

Firstly, timing belt teeth mesh and unmesh from timing belt pulleys at a frequency that is exactly related to the pulley's rotational velocity, diameter, and the pitch of the belt. While subtle, those meshing events can cause belts to vibrate [211]. The same is true for gears in general, where meshing of teeth happens periodically, although it is particularly important for belt drives because the belts can act like plucked strings on an instrument [212], and their resonances change as their lengths and tensions change [213], complete models of these phenomena can require complete FEA analysis [214].

Ballscrews and leadscrews are another common type of machine transmission,[73] these can "whip," buckle or vibrate under high speed or high load conditions [215], models have been developed to analyze these properties [216], and predict or analyze their failures using analysis of their vibrations [217].

[72] In 3D printing these produce what are called *ringing* artefacts on part surfaces, and in CNC machining they are related to *chatter,* some more on those later as well.

[73] These are just what they sound like: the motor turns a screw, and the machine axis is affixed to a *nut,* rotating the screw translates rotary motion into linear. In the case of ball screws, the nut is in contact with the screw via recirculating ball bearings to reduce friction and increase precision. Many arrangements exist: the most common mounts the motor to the machine chassis and the nut to the moving axis, but we can also mount the screw rigidly and rotate the nut - actually this second arrangement is more common on longer drive spans where screw *whip* can become important, because it causes the screw to stay taught more often to prevent it from buckling.

So, elements of the transmission can cause excitations — but so can the motors themselves. Stepper motors in particular are driven with *pulses.*[74] Where motor speeds, pulse frequency, and structural resonances line up, this can excite machine structures [218], [219]. Brushless servos of all types produce *cogging torque,* which is a similar phenomenon: internal motor construction is non-uniform, and so particular phases of the motor's rotation (where rotor and stator irons are better aligned) can produce more torque than others. This too can excite systems if it is not corrected for, as in [220] and [221].

Finally, fabrications processes can excite machines structurally — this is most apparent in CNC machine *chatter,* which I discuss in Section 6.1.4.2.

4.2.2.5 Temperatures and External Disturbances

To understand how minute the concerns here can become: national scale manufacturing of extremely precise optics (used mostly for spy satellites and then occasionally for science) is done on machines that can hold position down to *nanometers* across meter-scale parts [222], [223]. Even small changes to their chassis' temperatures can cause deviations of this size, and so they are built within multiple nested thermally controlled rooms. High end CNC mills often have active thermal control on each of their axes: Kern (a company that specializes in ultra-precision machines for small parts in e.g. medical device manufacturing and radar waveguides) elects to build their machine components mostly in aluminum (rather than steel or iron, which is more often chosen for its stiffness, damping and mass properties) because its conductivity provides better thermal stability [224], [225]. In machines with timing belt drives, ambient temperature (and temperature rise due to belt friction) can change those belts' spring rates nontrivially.

National scale metrology equipment like the device that measures the kilogram's mass according to universal physical constants (called a *kibble balance*) are similar [226]. I was lucky enough to visit this machine: to get there you traverse about eight flights of stairs underground (elevators cause too much disturbance), and then into a room, and then into a second room that is structurally separate from the rest of the building (suspended in its own mass of dirt and concrete). You get the point: when extreme precision becomes key, lots of physical gremlins appear.

To back out to more relevant scales, even the structure of the desk on which our machine is placed can affect its frequency response: forces that the machine exerts on its masses must be reflected ultimately into the earth's own mass — if they do so through a flexible desk, the machine system changes! Consider a print farm's shelf with nine 3D printers on it — each of those are now interacting a-la the famous metronome synchronization demonstrations.

4.2.2.6 Interaction with Process Physics

The coupling between process physics and motion control is perhaps the most complex of this set. There are many manufacturing processes and each is internally heterogeneous; tools, materials and other configurations change between jobs. Modelling these physics properly can also require

[74] Actually it is common to see frequency-dependent dips in a stepper motor's torque curve; these are speeds when the pulse frequency resonates the rotor itself, which can lead to motors losing steps more often at those particular frequencies.

that we model the geometries involved and computational representations of variable geometry can be heavy.

I discuss these physics for 3D printing in more detail in Section 5.2.1: those physics are largely rheological and couple into motion control because nozzle pressures must be dynamically changed alongside machine velocity in order to precisely deposit polymers at target track widths. Printer rheology is actually time-dependent because it is also coupled into the nozzle's thermodynamics; completely optimal motion control for 3D printers requires look-ahead over much longer time scales than what would normally be used for motion alone, see 5.11.5 for some discussion on that topic.

CNC machining physics are briefly discussed in Section 6.1.1, and I have made some mention of them in this chapter, mostly about the vibrations that the machining process imparts on milling machines. Cutting force is also a factor: where motion control is fundamentally constrained by the forces we can exert on our machine, sudden changes to those forces (e.g. when a milling machine enters or exits a cut) can of course matter a great deal. Here the modelling challenge is geometric: to account for these step changes dynamically we would need to represent part and tool geometries and also update them as they are machined. In any case, I do not engage in these experiments in this thesis, but I do show how motion data can be combined with motor measurements to estimate cut forces in Section 6.3.2 (this is not a new invention here, I develop it as an additional demonstration for other methods in the thesis).

4.2.3 The Role of PIDs and Hierarchical Control

Controllers of any complexity are normally formulated hierarchically, where lower levels (like current control) are handled with simple PI controllers[75] and more complexity is added as we move *up* in systems scope and *down* in response time, see [227] for an overview of this approach and [228, A. SaLoutos, H. Kim, E. Stanger-Jones, M. Guo, and S. Kim [229]] for more modern hierarchical systems.

The rule of thumb is that the bandwidth of the controller *below* the one you are currently designing should be fast enough that the delay it introduces can be ignored when compared to the current level. A good example of this is shown here in Section 4.4, where I develop a motor controller that uses two levels of control internally: microseconds to close current control loops and milliseconds to close position control loops. The position controller's target trajectory is a basis spline whose control points are output by the velocity planner from this chapter, which covers longer time spans using look-ahead planning.

While hierarchical control is a useful partitioning of the problem they can still be somewhat lossy because the minutia of lower levels are ignored by upper levels. However, this is also the point: considering all the required physics simultaneously is difficult and can get messy. For example trapezoid planners that do not smooth jerk will output trajectories that are not physically realizable by motors that cannot instantaneously change current in their coils. In practice this is OK for most

[75] PI Controllers are Position, Integral, Derivative controllers (PIDs) without the derivative term. These are common for control of systems that don't have any type of momentum.

systems because lower levels of the controls stack work well enough to correct for it; we introduce some error, but it is negligible.

Partitions here are set up by hand / according to heuristics: arranging these partitions is a design exercise that involves the normal trade-offs between complexity, utility, and performance.

4.2.4 Trapezoidal Planners

Trapezoidal planners (or s-curve generators) are the most common and practical type of motion controllers that we find in practice, although the exact configuration of most commercially available machines is difficult to study because their controllers are proprietary, especially CNC machine tools [43]. In digital fabrication research, where machines are often controlled using open source hardware, trapezoid planners are the rule. They are very well understood in texts as single-segment planners [230, sec. 9.2.2.2] but their extension into multi-segment trajectories involves one or two additional steps that I should explain.

Firstly the basics: they work by generating *trapezoids* in the velocity-time plot, which are *s-curves* in the position-time plot.

Trapezoid segments assume fixed acceleration over each component, meaning the kinematic equations can be used, those are below.

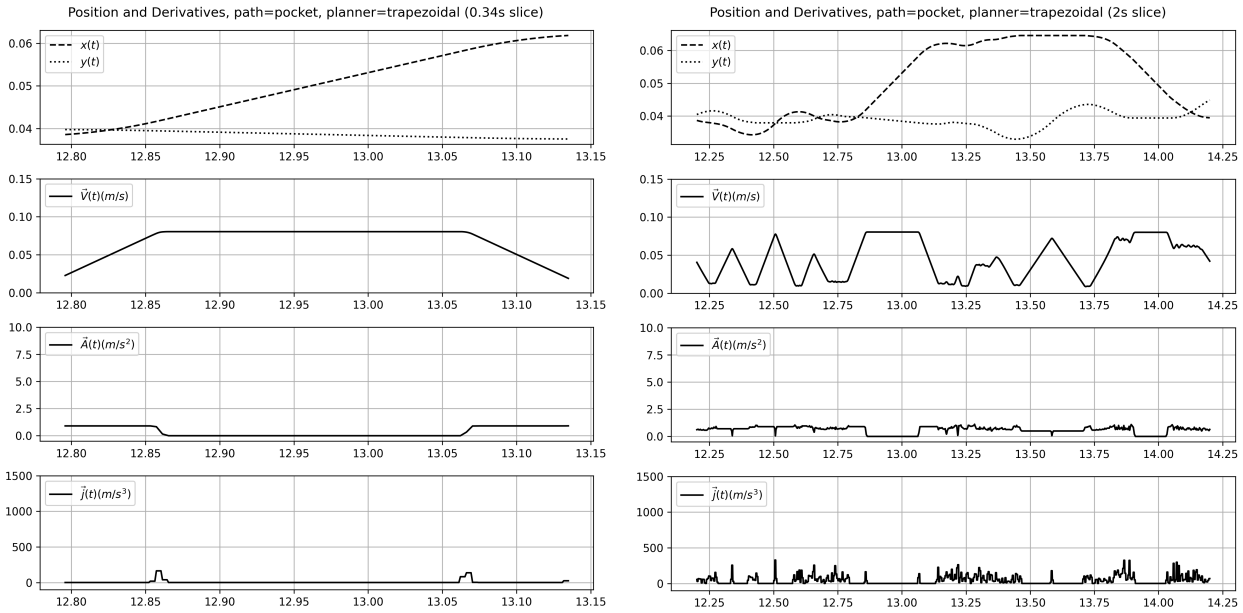


Figure 4.3: At left: A trapezoid in the $\vec{v}(t)$ (velocity *along* the path, in time) is s-curve shaped in position, and has square edges in acceleration, and infinite impulses in jerk (the third derivative of position). In this plot, I am extracting these states from a basis spline that encodes the trajectory (2.5.3); that encoding smooths the infinite jerk pulse, inserting a small ramp in the acceleration trace. At right: trapezoid profiles that have been linked using the junction deviation algorithm.

$$
\begin{aligned}
d &= v_i t + \frac{1}{2} a t^2 \\
d &= \frac{v_i + v_f}{2} t \\
v_f &= v_i + at \\
v_f^2 &= v_i^2 + 2ad
\end{aligned}
\tag{4.1}
$$

Trapezoids planners are configured by setting a maximum acceleration (*slope* in $v(t)$) and velocity (maximum *value* in $v(t)$) for each axis of motion. Some also specify maximum *jerk.* Using these values along with initial and final velocities for each segment v_i, v_f and a distance for the segment d, trapezoid planners use the kinematic equations to determine where acceleration and deceleration should be applied in each. Since motion parameters are set on a per-axis basis but line segments can point in any direction, trapezoid planners also choose per-segment maximum acceleration and velocity *normals* to use based on their unit vectors.

We must also choose a maximum velocity for the corners that sit in each junction between segments. In reality, we cannot move through a sharp corner at *any* velocity without an infinite impulse of acceleration. Most trapezoid planners use an heuristic algorithm called *junction deviation* at this point that approximates maximum cornering velocity using an *imaginary* arc segment between the two lines. The arc radius is chosen such that were it inserted at the corner the arc would not deviate from the actual corner by more than the defined junction deviation parameter (a distance), and the acceptable cornering velocity is set according to the radius of that arc. However, the arc is not actually interpolated, instead the cornering velocity is simply applied to the incoming segment's maximum v_f and to the next segment's maximum v_i. The original blog post on the topic is at [231] for the firmware GRBL [78], and also explained in [232].

So, junction deviation provides an initial maximum v_i, v_f for each segment and per-axis maximum acceleration and velocity are combined with unit vectors along each segment to choose the *segment's* maximum values. We then need to *link* multiple segments together and plan across them. This is important because we often have cases where we need to start decelerating for a corner that is ten or twenty segments away. In this step we use a form of dynamic programming [233]; we compute a *forwards pass* (ensuring each segments' v_f is not so large that it is not achievable with the segment's a, v_i, d) and a *reverse pass,* which is the same computation but from $v_f \rightarrow v_{i,max}$.

Trapezoid planners re-compute junction deviations and linking velocities whenever a new segment is added, and they always plan to come to a full stop at the end of the segments in this queue. The queue size *used* to be constrained by microcontroller memory footprints and compute power because each segment consumes some RAM and re-computing over each requires a few clock cycles. This is less of a constraint with modern embedded computing, but the queue imposes some other limits for e.g. updating plans in real-time that I discuss in Section 7.4.2.

Each segment in the queue is a single trapezoid; one end of the queue can be extended but the other is constantly being interpolated as the machine operates. Trapezoids are useful in this case as well because they can be rapidly evaluated in firmware to e.g. generate stepper motor pulses

or servo target values. When the planner interpolates past the end of each segment, it removes it from memory to free space for the next segment to be added.

It is worth noting in this point is that trapezoid planners are based on *fixed* maximum accelerations and velocities, whereas motor dynamics show us that they are capable of more: larger accelerations at low speeds, and higher speeds where low accelerations are required. Where our machines have nonlinear kinematics, minimum and maximum velocities and accelerations change through space as well; a robot arm requires more torque to move its base joint when it is outstretched because the rotational inertia of the arm is larger when it is in a longer configuration. Because trapezoidal planners are fundamentally based a set of kinematic equations for segments with *fixed acceleration* over their length, they are unable to exploit these dynamics. They are practical for the same reason: entire linear (and arc) segments can be rapidly evaluated in only a few clock cycles. See Section 4.7.1 for the full comparison between this type of planner and the model-based optimization that I develop.

4.2.5 Geometry Smoothing and Filter-Based Methods

The junction deviation step in trapezoidal planners is lossy because it approximates smooth junctions (virtual arcs) between line segments. When machines traverse those junctions (which remain sharp despite having nonzero velocity), they actually perform some instantaneous acceleration. These impulses are normally taken up by motors and other springs in the machine system, but it would clearly be preferable to avoid them entirely.

For this purpose, actually smoothing the trajectory such that it does not contain any corners with zero radii is best. This is the topic of many contributions in machine control research. For example [234] intelligently modifies GCode geometries according to geometric heuristics where (1) it is possible to do so within specified refitting tolerances and (2) it is advantageous to do so for motion control, i.e. smoothing small corners into arcs.

Most other methods in this family are similar: geometric tolerances are combined with varying geometric representations to refit input geometry such that maximum smoothness is attained while tolerances are respected, called "chord deviations". For example Lavernhe [235] does this for surfaces, considering also kinematics of a five-axis machining center. A simpler version of the same idea is implemented into almost every velocity planner: incoming GCodes encode segments which are smaller than the spatial tolerance of the actual system are ignored, simplifying computation over these segments. This is mentioned explicitly in the Delta Tau manual [236] and in Ward's thesis [237], and is implemented in interpreters like GRBL [78] and its many descendants.

This geometric smoothing can be combined with velocity planning itself by articulating the whole problem in terms of filter design; some of these methods apply filters on top of trapezoid planner trajectories to smooth sharp corners [238], [239]. They can also be used to reduce excitations generated by the planned trajectory at a machine's known resonant frequencies. This is known as *input shaping,* [240], [241], [242], and it has become common in even consumer grade 3D printer controllers like Klipper [58], which I covered in the systems chapter (Section 2.2.4.4). Input shapers

are *designed,* they are configured to reduce energy at selected frequencies and a number of different styles exist. Understanding which frequencies should be filtered is normally done against frequency response plots generated using accelerometers mounted to machines.

Finite Impulse Response (FIR) filter-based methods are the basis of another family of velocity planning algorithms that can supplant (rather than simply extend) trapezoidal planners. The method was originally developed in [243]: rather than planning with trapezoids and then filtering corner geometries or modifying velocity profiles to remove certain frequencies, filter parameters are derived from maximum jerk, acceleration, and velocity parameters. Applying a finite impulse response filter thrice to a target trajectory (of line segments) results in a planned trajectory that is continuous in jerk. These have been extended to solve also for time optimality while suppressing other vibrations in [244]. [245] improved the FIR approach for use in human-facing robotics where online replanning of trajectories is important, and [246], [247] made further improvements for five axis machining and compute performance.

FIR-style velocity planners have a few advantages over trapezoid planners: they can smooth geometries and velocities in one step (avoiding the junction deviation abstraction) and since they are already formulated in the frequency domain they can be easily modified to suppress other vibrations. They are also a much simpler way to generate trajectories that are continuous in the jerk derivative when compared to methods that extend trapezoid planners to do the same. However, like trapezoid planners, they are configured using direct kinematic parameters (maximum velocity, acceleration, and jerk — and geometric tolerance), rather than using models of these systems. While those parameters can be *extracted* from good models of a kinematic system, this means that they have many of the same limit that I mentioned in Section 4.1.1.

4.2.6 The State-of-the-Art in Industrial Motion Control

It is hard to know for sure what the internals of state-of-the-art industrial motion controllers do because these firms guard their algorithms closely [43]. To investigate, I read some technical literature that is available publicly from a few purveyors of motion control. From Heidenhain (a leading provider of machine controllers in the machining industry) I found trade literature [248] and a manual that is made available for machine tool builders [249] which was particularly useful because it is written for a knowledgeable audience. I did the same for Siemens Sinumerik ONE controllers [250] but didn't find any equivalent "manual for integrators," presumably one has to enter into a trade agreement with them before they will explain their technology. I also read (not entirely, it is 832 pages) the manual for Delta Tau's PMAC (for Programmable Multi Axis Control) controllers [236]. These are more general purpose controllers as their name suggests, marketed for use in everything from packaging equipment to multi-axis synchronized machine control with velocity planning over multiple segments. As a result, their documentation is the most complete: they cannot really hide their algorithms from their users since the configuration of their

equipment across multiple domains requires that it is understood by those users.[76] I also discuss these controllers' system architecture more broadly in Section 2.2.

4.2.6.1 Fixed Kinematic Parameters, not Models

Based on the velocity-time plots available in each, and on notes in literature for machine tool builders on how they are configured, each example is fundamentally based on kinematic parameters (maximum velocity, acceleration and jerk) for each axis. This means that they must be either trapezoidal or FIR-style velocity planners. For example in the Delta Tau manual [236] around page 738, we see trapezoidal profiles for look-ahead planning through multiple linear segments. There, motion is planned according to kinematic parameters defined for each *motor* in the system. They also note that if the user wants to constrain motion based on limits applied to the end effector position (i.e. motion on the other side of the kinematic chain), they should apply "virtual motors" onto these axes, noting a key limitation across these methods: real physical constraints emerge from motor space *and* cartesian space, but here we can normally only write parameters for one or the other. In Chapter 2 of Ward's thesis [237], he fits measured velocity data from a DMG Mori to an FIR-style velocity controller, showing that their velocity planner is likely based on the same method.

Siemens Sinumerik ONE marketing material claims that it uses a digital twin representation of the machine for control, which could mean model-based control that is more similar to the velocity planner that I develop here. The most comprehensive documentation they provide publicly are manuals for machine interface configuration, these detail steps to attach 3D models to GCode axes definitions, but say nothing about supplying physical model parameters like mass, motor data, or damping terms. This indicates that by "digital twin" they may only mean that their planner is attached to a 3D representation, but not more fundamental models for motion. That said, 3D models do provide immense value for collision detection; Heidenhain controllers seem to have similar capability.

4.2.6.2 Compensations

Heidenhain literature [248] mentions a few different controls technologies that compensate for errors that arise during operation. These all seem to be methods that are applied on top of either trapezoid or FIR-style planners. There are multiple compensations mentioned there, each of which gets a small paragraph below:

CTC for "cross talk compensation" is *compensation of acceleration-dependent position errors at the tool center point,* adjusts trajectories based on machine deflections that arise from acceleration forces. They mention that parameters for this compensation can be found using machine measurements, but no other information is available on the topic. They relay that this "makes it possible to increase jerk by a factor of 2" (in one example), indicating that these stiffness models are not automatically incorporated or fed back into the actual acceleration control step.

[76] A little bit off topic, but remember here the earlier discussion on how technical partitioning interacts with economic partitioning (Section 1.4). In this case, in order to sell motion control systems Delta Tau has to provide a kind of set of *design rules* [37] for systems integrators.

PAD for *position-dependent adaptation of control parameters* adapts control for various kinematic poses (i.e. a beam gantry which has lower stiffness than when it is contracted) through "position-dependent filter settings and control factors," again indicating that the method is likely applied using tabulated values or interpolated parameters through space, rather than on position-varying kinematic models themselves.

LAC for *load-dependent adaptation of control parameters* seems to be the most advanced, it enables the controller to "automatically ascertain the current mass with linear axes and the mass moment of inertia with rotary axes as well as the friction forces," which is very similar to the methods that I describe in this section. It is unclear if these updates are transmitted to the velocity planner itself, based on their language for "feedforward parameters" it seems more likely that they are updating lower-level control gains for axes servos to reduce following errors.

AVD for *active vibration damping:* despite sounding like an online control method ("active") on closer reading sounds like FIR-style filters that are parameterized around vibrations from the machine's structure and from motion components (i.e. torque ripple), not adapted online based on disturbances from the process (cutting, i.e. chatter). Again, these updates are not reflected back into the velocity planning constraints themselves. Rather, their inclusion "allows jerk limits to be increased."

Besides these four compensations, the Heidenhain literature also mentions compensations for temperature and kinematic models. These apply spatial offsets to target trajectories, for example moving machine axes away from other machine components that have expanded due to heat, or correcting motion on axes that are slightly out of square. Each of these is also configured using a set of parameters rather than by describing e.g. models for thermal expansion.

Overall, these controllers are certainly sophisticated equipment; they represent the pinnacle of what is available "on the factory floor" today. However, they are still fundamentally based on directly written motion parameters and their various compensations are added *on top of* trapezoid or FIR-style velocity planners. With more fundamentally model-based control, there is an opportunity to express these operations in a more straightforward manner.

4.2.6.3 Process Coupling and Active Control

Heidenhain also details a few online control strategies that manage interactions between process physics and velocity planning; *ACT* for *active chatter control* and AFT *adaptive feed control* both appear in [249].

AFT is likely based on a patent from 2008 [251] where feed rate targets are adapted based on spindle load measurements. A older Boeing patent from 1987 [252] is related; there feed rate overrides are modified using data from strain gauges mounted to a milling machine's spindle. ACT can be traced to a more recent 2018 patent for monitoring spindle load during machining [253] and one from 2023 [254] for adjusting feed parameters accordingly.

Both approaches connect to the velocity controller's feed rate *override,* so they are similar to research literature that I will cover in Section 4.2.8; they do not connect these concerns *to* the velocity planning step, they run in an outer loop around that step using feedback rather than

predictive planning. Presumably though the active chatter control here is often used in conjunction with the active vibration damping modification described in the prior section, but again both are filter design methods rather than being model-based.

These methods extend well beyond the topics covered directly in this thesis. It is interesting however to see the partitioning problem appear here again: chatter is largely geometry dependent and because planners of all nature lack good representations for changing part geometry, so most methods are based on active control rather than predictive. As we move more control into higher performance computing environments, it will become easier to manage these physics via computation directly against simulateable models.

4.2.7 Model-Based Control in Robotics

So, the industrial state-of-the-art seems to prefer control methods that are more classical in nature, which makes sense in a risk-averse industry with a long history; CNC machines entered service on the factory floor in the early 1960s [8]. It also makes sense where determinism and predictability is of primary importance.

The story is similar for *industrial* robots e.g. the 6-DOF[77] arms that populate automotive manufacturing lines, welding lines, or material handling lines.[78] They have been on the factory floor since about the 1970′s [255] and their controllers are similar in nature to those present in CNC machines. Many of them are even tasked using low level instructions that look similar to GCodes: KUKA robots are programmed in KRL (KUKA Robot Language) and ABB robots are programmed in a language called RAPID. A longer discussion on that topic would be interesting, but it is outside the scope of the present focus.

Here I want to discuss model-based methods that *are* quite distinct from industrial motion control, so we want to look controllers from the more sci-fi instantiation of the word *robotics* i.e. things that have legs and arms. Modern approaches to control these systems are much younger than CNC, and they are often based around some form of Model Predictive Control (MPC).

MPC controllers work in much the same way as you and I do: we use a simulation of the future to optimize future and current control outputs. The classic example comes from driving: when we are entering a corner, we know that we need to slow down before the turn. To pick a braking point, we use a mental model of our car's dynamics (how long it takes to slow down) to simulate (imagine) how long it will take to come to the appropriate entry speed for the corner. With computational MPC, we simulate and optimize control outputs over a horizon of up to a few seconds *at each time step,* but only issue the immediate next control output to our system. These are common in practice to solve controls problems for legged and bipedal robots [56], [256], and other dynamical

[77] DOF: Degree of Freedom.

[78] For a quick aside and motivation towards more unified control architectures, a common application of 6-DOF robot arms is in *tending* CNC machines: loading stock and unloading finished parts. It is common for the robot arm to physically press the CNC machine's *start cycle* button because connecting their system logic together is much more cumbersome.

systems like quadcopters [257]. The best practical overviews of their implementation I have found are in Tedrake's course notes [258] and Brunton's textbook [259].

These solvers are normally constructed symbolically and are subject to the constraint that the optimizations they solve must be quadratic programs, i.e. highly convex with one global minima. Two common tools for this practice are CasADi (2019) and acados [261]. CasADi lets developers author symbolic representations of their system dynamics and cost functions, and then generates C code for evaluating derivatives and integrating dynamics models. Acados is a solver that can directly use those C codes to minimize cost functions in real-time. So for a programming workflow example: we write down our systems dynamics mathematically in a symbolic programming language, write a cost function for that system, and then compile that into a *quadratic program* (QP, [262]); this step is possible only where the mathematic definition of the problem meets quadratic programming constraints. That program is then copied into the controller's firmware and recompiled to run. So these are akin to programming languages and compilers: we write systems and constraints at a high level (mathematically), and then automatically generate programs (solvers).

MPC typically uses the canonical state space representation [263] or [264], which I will include here for clarity:

$$\begin{aligned} \dot{x}(t) &= A(t)x(t) + B(t)u(t) \\ y(t) &= C(t)x(t) + D(t)u(t) \end{aligned} \tag{4.2}$$

Where x is the state vector (what is happening), y is the output (what happens), and u are the control inputs, i.e. the knobs that our controller can set; normally motor voltages, currents, or velocities depending on the controller's hierarchical design. The matrices describe various aspects of system evolution. The $\dot{x}(t)$ function describes how the system evolves from any given state to the next state, with inputs $u(t)$ from the controller. Using this formulation to simulate the system is easy, because that function is already an integrator. MPCs work on this representation and are typically *either* formulated using a *"direct shooting"* method or a *"direct transcription"* method. [258]

Direct shooting integrates states forwards in time. The solver begins with initial state $x(t)$ and picks control outputs over time, some $u(t)$. So the control inputs are the solver's *decision variables.* It then calculates forwards how the system evolves over time given those inputs, compares points on that planned trajectory to an optimal outcome and computes a cost for the solution. The gradient between the decision variables and the cost is used to update the $u(t)$ towards an optimal set. This is perhaps the most comprehensible formulation because it mirrors how we naturally think about dynamical systems. However, it requires serial computation of the system's evolving states which makes it difficult to compute, and gradients can vanish over long integrals.

In direct transcription, the solver picks *the resulting trajectory* $y(t)$ and each step's internal states $x(t)$ directly, and also the control inputs $u(t)$. This breaks the time dependence and allows us to parallelize much of the computation. But it also requires that the solver implicitly link each time step such that control states at each time $u(t)$ for each system state $x(t)$ generates the next system state $x(t+1)$. That is, the cost function compares the planned trajectory to optimal outcomes, but

also penalizes steps where the time series of states doesn't make sense according to the system's dynamics. Because the solver is now choosing outcomes *as well as* inputs, the dimensionality of the problem increases significantly. But, this has advantages for parallelization and also means that the solver doesn't have to compute gradients *through* as many time steps.

4.2.7.1 New Tools for Constrained Optimization and Differentiable Simulation

I am enabled to complete the work in this chapter largely because of a new availability of automatic differentiation (aka *autograd*) tools, that automatically generate gradients for *any given program* (gradients being a key ingredient for optimization, most performant optimizers are based on *gradient descent*). In particular, I am using JAX [265]. With JAX, any system that can be articulated as a purely functional Python code can be automatically differentiated. We can then use the function's gradient to optimize the function's inputs, and JAX conveniently pairs with optimizers from the OPTAX library [266]. The performance of this system is only viable because JAX also includes a just-in-time compiler that turns the whole system into code that can run in parallel on high performance hardware (the GPU).

4.2.7.2 Reinforcement Learning and Policy Control

MPCs are typically used for short-order optimizations over time horizons around one to five seconds, and they usually run somewhere between *50Hz* and *500Hz.* For longer horizons of control and to solve problems that are not differentiable the current best practice uses simulated systems to train *policy controllers* that can issue higher-level control commands. Policy controllers are normally coupled with lower level, faster feedback controllers and classical controls components like Kalman filters and PIDs — recall the note on hierarchical control above (4.2.3). [267] is a particularly clear example of how all of these components come together to produce performant controllers, it includes classical simulations, policy networks, Kalman filters, and PIDs. Policy controllers are often trained without gradients because it is hard to differentiate through simulations that involve rigid body collisions and that happen over long time spans, but recent work deploys differentiable simulation to overcome this issue [268].

Policy controllers are one component of Reinforcement Learning (RL) for robotics: are the "brain" of RL systems. They compute optimal actions (a) based on observed states (s) from their environment and are trained with rewards (r) that are also computed from their environments.

RL is making huge progress recently that extends from control into Large Language Models (LLMs) because it is possible where differentiation of the environment itself is not possible — i.e. they can be trained to manage large, messy tasks that don't have straightforward or analytical mathematical representations. In robotics, RL it is mostly applied to legged or wheeled robots (but again, combined hierarchically with lower level controllers).

Overall the topic is interesting here because progress in applying RL to machines could be enabled by some contributions in this thesis; namely the systems that I develop to more easily connect high- and low-level computing and make sense of hierarchical control systems. RL policies are normally trained offline using countless simulations of their environment in what researchers call "infinite

fun time,"[79] the operation of which requires good ground-truth simulations of those environments. The models developed in this thesis for machine systems would be appropriate in these contexts as well. I discuss how RL could help to connect work in this thesis to *path* planning in Section 7.7.1 and expand on this thesis' connection to RL and AI more broadly in Section 7.7.

4.2.8 Integrating Model-Based Control with Machines

Model-based approaches are less common in machine control than in robotics for perhaps a few reasons. The first is the preference in machine control for deterministic, fast controllers that I mentioned in Section 4.2.6: optimizers are less inspectable than classical control approaches and can sometimes generate unexpected behaviour. The second is related to the partitioning problem: bandwidth requirements for machine control are much greater than in robotic contexts, and so even more computing power is required to run optimization based solvers here.

Although it is impossible to know how common model-based control of machines really is in industry [43], there are a few good examples in the literature that I will look at here.

4.2.8.1 Integration Around Interpolators

In many of these examples, model-based planners are added to existing velocity planners. [270] proposes MPC to maximize feed velocity within dynamical limits involving feed rate and cutting forces. [191], [192] and [271] all make excellent progress for online estimation of cutting forces, detection of chatter, and modify machine velocity control to optimize or correct these physics. But in all three cases the control signal from the model-based controller plugs into an existing machine controller's *feed override* input. This sets the *target* feed rate for the machine, which its own controller is in charge of; in [270] the authors note: *"however in such a scenario, an accurate modeling and control of the feed dynamics become more crucial, as the built-in routines within the machining center induce unknown nonlinearities."* This adds complexity to the author's work and requires that they also model the hidden planner itself in order to anticipate its internal dynamics[80]. The theme appears also in [235], where the authors are concerned with optimizing toolpath geometry against a five-axis milling machine's motion constraints; *"… The actual feed rate is thus locally lower than the programmed on due to the interpolation cycle time."*

[79] Infinite Fun Time is originally a reference to an excellent sci-fi series [269], where superintelligent AIs while away their time simulating universes in their imagination. Training RL policies is quite similar: numerous simulated environments are instantiated where policies are tested, generating rewards to improve those policies.

[80] This style of control system is called a *piggyback* controller, which is pretty self explanatory. I first learned of these in the context of internal combustion Engine Control Units (ECUs) (thanks Dad !). Hot-rodders often want to unlock spare performance in their off-the-shelf automobile's engine without spending much money. Replacing ECUs outright is possible but expensive and difficult to get right; the engineers at i.e. BMW definitely know a thing or two. Instead of a full refit, we can intercept some sensor signals (like an exhaust oxygen sensor, which signals to the ECU what the current fuel mixture is like), read them, and modify them before they go to the stock ECU, to coerce it into doing things (like adding more fuel to the mixture) that it might not be meant to do. This involves some reverse engineering of the computer that we would like to trick - and is certainly a hack - but i.e. here *and* in the case of GCode interpreters, is often the best solution.

In fact, Rob Ward of [191] (whose work was also covered in Section 4.2.5) includes in his PhD methods for the prediction and modelling of GCode interpreters [237] for the purposes of improving the development of these types of "piggyback" control systems for CNC machines.

Using feed rate override is not necessarily a bad approach, it is an important feature that allows machine operators (and these researchers) to quickly change the most critical parameter for machine operation. It is also exactly the kind of separation of scales that we would expect in a hierarchical controller, and these researchers show excellent results. It does introduce some loss and as I mentioned adds complexity; model-based control should enable even more fundamental control of machine dynamics; for example the inclusion of motor dynamics themselves — and direct access to motor torques — could significantly improve each of these controllers' force bandwidths by bypassing their velocity planners' imposed bandwidths (recall that they are essentially filters!). For discussion on how these researchers managed control integration from a systems architecture viewpoint, see Section 2.6.2.

In FFF, Pinyi Wu uses feed-forward modelling of extrusion dynamics to improve control of material extrudate [272], [273]. These methods benefit the extruder system, but the extruder system's dynamics are not fed back into the velocity controller in any manner. That is, the two systems are controlled in separate control paths; Wu's model-based controller tracks a signal that is generated from an off-the-shelf interpreter. Also in FFF, [274] integrates some melt flow modelling with a 6-axis robot controller — their systems architecture also tracks outputs from the robot's hidden velocity planner.

4.2.8.2 Other Optimization-Based Interpolators

[275] presents an optimization based controller for CNC machines whose formulation is distinct from methods that I have covered so far, and it is solved using the research group's own optimizer from [276], which was originally developed for automobiles. In [277] they combine this optimizer with earlier work on chatter control for milling that I mentioned above [271] where the chatter control was implemented using feed override of existing machine controllers.

Their implementation performs the optimization by selecting a time series trajectory directly against a reference trajectory made up of linear segments. This allows them to minimize time for the path against geometric constraints (deviations from those linear segments) as well as motion constraints. But those constraints are formulated as maximum per-axis jerk, acceleration, and velocity values that are chosen ahead of time, rather than being themselves selected by the controller or expressed directly as models for motion.

4.2.9 Comparisons to This Work

As I will discuss, the planner that I develop here is somewhat similar to a *direct transcription* style MPC in that it is parallelizable and that it selects the solution rather than the control values. But really it is somewhere in between MPC style control and classical velocity planners. It is described entirely in Section 4.6.1 and then discussed in 4.8.2. To generalize I would say that it *represents*

the problem in a manner that is most similar to velocity planner, but it *solves* the problem in a way that is more akin to MPC, or to the planners from Section 4.2.8.2.

It is structurally different from each of these in that it does not use direct parameters to constrain motion profiles themselves (maximum jerk, accelerations, and velocities) and analytical representations (as in most velocity planners) nor does it use quadratic programming (as in MPC). It uses a more generic first-order gradient descent method that is borrowed directly from a set of tools developed for large machine learning problems, and it represents constraints not directly on the planned trajectory itself but by broadcasting the planned trajectory back through kinematic, inertial, and motor models in order to arrive at those constraints.

This allows it to exploit more of the underlying system's dynamics because those constraints vary alongside other system states. This is particularly relevant at low speeds where motors can produce more torque and where friction is lower. It can also exploit friction within our systems that actually helps us to decelerate; planners that use fixed values for these parameters apply the maximum acceleration value also during deceleration.

There is also the matter of kinematic transforms: planners that use direct parameters must define them either in the space of the motion path itself (which is most common) or in the space of actuators (which is the default in e.g. the Delta Tau controllers [236]). In systems with trivial kinematics (one motor per axis) this is not so much of an issue, but many machine systems use parallel kinematics where motion of the machine tool is coupled to multiple motors — CoreXY is the most common of these arrangements. In these cases we have distinct inertias and damping terms in both spaces and so mapping limits from each space into one set of direct parameters is not straightforward. This matters for inertial and damping terms but also for motor physics. For example when a CoreXY-based machine moves at a 45-degree angle, only one of its motors is used to move both x and y masses, but when it moves in x or in y alone, both motors are used — this means that the real underlying motion constraints are anisotropic!

Using models directly also simplifies controller authorship and allows us to add dynamics to the system that may be cumbersome to describe symbolically, we can "pile into the cost function". It also enables us to use the same component descriptions in the model fitting *and* model-based control workflows, and avoid the step where we might re-compile and update a firmware when we generate new models (as in with CasADi and acados).

This is significantly less performant than the state-of-the-art in MPC for realtime control in terms of the FLOPs[81] required. I make up for the increased FLOPs requirement by formulating the computation so that it parallelizes well on the GPU, and developing the low-level motion system description and representation in Section 2.5 and Chapter 2 more broadly, which helps to connect the large compute required to run this planner with our hardware.

All of that said, the planner that I develop here is clearly missing the geometric consideration: the abstraction that I use to compute required accelerations along the path (in 4.6.1.1) is somewhat lossy and computed solutions are transmitted using basis splines that introduce some interpolation

[81]Floating-point Operations: how much total math we need to do.

error. Properly, these errors should be accounted for in the optimization loop itself. I discuss this in Section 4.8.5.3, and possible improvements to the lossy acceleration calculations in 4.9.1.

It is also less deterministic, which may be a real issue to larger scale adoption of the technique. That is partially due to the nature of optimization based control and partially due to the *location* of the planner (which runs within a non-realtime OS). However, the same questions on determinism are being answered by researchers who deploy these types of optimizers for human-robotics interaction. It may also be possible to modify lower levels of the controller to enforce safety boundaries in cases where the planner itself fails, as I discuss briefly in Section 2.9.4.3.

4.3 Models for Motion Control

This section covers background on key models for motion, for motors (below) and for kinematics and damping (4.3.2). I then use a closed-loop stepper driver (4.4) to fit those models to data in Section 4.5.

4.3.1 Motor Models

Motor physics are the most foundational for motion planning: as discussed, we need to understand these physics to know what our motion constraints are going to look like. This means electric motors in their many forms: classically we have Brushed DC, more modern control typically happens on Brushless DC (BLDC or PMSM: Permanent Magnet Synchronous Motors) which both rely on permanent magnetic fields interacting with changing magnetic fields, and Induction or Reluctance motors, which operate based on the field's *propensity* to "want" to move through magnetic circuits that have low reluctance. Luckily for us, the models are all very similar and basically involve generating current with voltage, and then figuring out how current (which is a close proxy for magnetic field strength) relates to the motor's torque and speed.

Because they are extremely common in machine control, I take particular interest in Stepper Motors, which are known technically as Hybrid Permanent Magnet Reluctance Motors. They work partially like a BLDC and partially like Reluctance Motors.

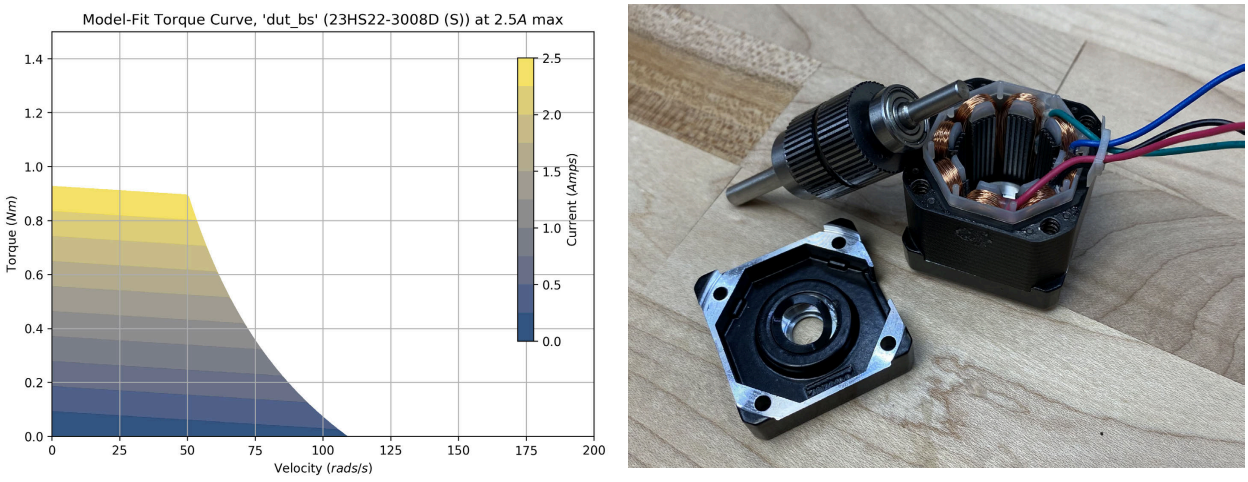


Figure 4.4: At left, a motor torque curve generated by the model in this section, showing torque (normally τ, on the y-axis, in Nm) produced by the motor at varying drive currents (color-mapped), depending on the motor's velocity (x-axis, here in $rads/s$). At right: a disassembled Stepper Motor, showing the stator with windings (copper) and teeth (steel laminations). At left, the rotor on its own showing the tooth offset on the top and bottom laminations — the biasing permanent magnet is in between those.

Our motor has a stator (the static part) with electromagnetic windings that can generate a magnetic field, and a rotor that has a permanent magnetic field. Motors work because these two fields (in the stator and rotor) *want to* line up with one another (which is a way of saying that they *are in a lower energy state when they are aligned*). The teeth in a stepper motor are arranged such that these aligned states are closer together: given some stator field, the closest alignment for the rotor is never more than 7.2° away from the rotor's current position. That is, for every *electrical rotation,* when we rotate the magnetic field through $0, 2\pi$ only $1/50^{th}$ of a mechanical rotation occurs.

When the motor is operating, we switch current in the stator to motivate the rotor to rotate continuously. In *brushed* DC motors, this happens mechanically: the brushes are aligned such that any current passing through them switches onto the windings that will generate current out of phase with the rotor. In brushless motors (steppers are one type of brushless motors), we do this using digital switches and some computing. This process is called *commutation* in all cases.

The motor physics for any electric motor breaks down into a few constitutive equations, which are below: the electrical half of the motor in (4.3), the mechanical equations in (4.4), and a table of each symbol, its units, and meaning in 4.1.

$$\begin{aligned} Z &= \sqrt{R^2 + (\omega_e L)^2} \\ \omega_e &= \omega_m p \\ V_p &= V_a - k_e \omega_m \\ \dot{I} &= (V_p - RI)/L \\ I_{ss} &= V_a / Z \end{aligned} \tag{4.3}$$

The electrical half of the model describes how the voltage that we apply to the windings V_a becomes the rotor current I which will generate torque. The first equation above defines the motor's impedance: $Z, ohms$ — this is important especially in stepper motors where inductance L can be quite large, and is one half of the reason that generating current at high RPM's is difficult: with very many pole pairs (p) in a stepper motor (50), we quickly end up with large impedance in the stator when we get going at high speeds. The second equation shows how electrical and mechanical speeds are defined by pole pairs.

The third, fourth, and fifth lines above describe stator current. We have that the voltage potential across the stator windings — V_p — is the applied voltage less back-EMF ($k_e \omega_m$). Back-EMF is the voltage that the rotor generates as its magnetic field moves through the stator's windings.

The dynamic value of the stator current I is defined by its rate of change (fourth statement): $\dot{I}$ is potential voltage less *voltage already contained in the stator* (RI), divided by its inductance. This is the term we want to use if we want a time-varying estimate of current, or to estimate the maximum $\dot{I}$ which becomes common later on in this chapter.

The mechanical half of the model (below) shows that torque generated by the motor τ is a simple multiplication of stator current with the magical k_t parameter (which we will fit to data later on).

Symbol	Units	Value	Source
V_a	$Volts$	Voltage *applied* to the stator coils	Output(t)
V_p	$Volts$	Voltage *potential* at the stator coils	Product(t)
A	$Amps$	Current *in* the stator coils	Measurement(t)
R	$Ohms$	Stator's DC Resistance	Measurement
L	$Henries$	Stator's DC Inductance	Measurement
Z	$Ohms$	Stator's Impedance	Product(t)
τ (tau)	Nm	Torque	Measurement(t)
p	n/a	Number of Pole Pairs	Measurement
ω_e (omega)	$radians/s$	Rotor *Electrical* Velocity	Product(t)
J	$kg \cdot m^2$	Rotor Inertia	Measurement
Θ (Theta)	$radians$	Rotor Angle	Measurement(t)
ω_m (omega)	$radians/s$	Rotor *Mechanical* Velocity	Product(t)
$\dot{\omega}$	$radians/s^2$	Rotor Acceleration (derivative of omega)	Product(t)
k_t	Nm/Amp	Torque Constant	Model Fit
k_e	$Volts/rad/s$	Back-EMF Constant	Model Fit
k_d	$Nm/rad/s$	Rotor Damping (friction)	Model Fit

Table 4.1: Table of motor model symbols, units and values, including notes on how the motor controller in this chapter 4.4 interacts with them.

The rotor acceleration $\dot{\omega}_m$ is torque, less the torque produced by rotor friction ($k_d\omega_m$) divided by the rotor inertia J, and the rotor's angular position is just the integral of mechanical speed.

$$\begin{aligned} \tau &= k_t I \\ \dot{\omega}_m &= (\tau - k_d\omega_m)/J \\ \Theta &= \int \omega_m \, dt \end{aligned} \tag{4.4}$$

Two more notes on this, which are not required reading:

1. The sign of I *can easily be opposite to the sign of* V_a, i.e. when the motor is generating current in one direction while our controller is driving it in the opposite direction. This is where electric motors enter regeneration. This is very cool and useful when we are in a battery powered system (regeneration simply shows up as a lift in voltage on the supply, which charges the battery) but can cause issues where we use wall-supplied switching power supplies: they shutdown when the voltage at their output terminals rises substantially past their target output voltage. To ameliorate this, I simply mount a big bleed resistor to my power supplies to perpetually drain excess power (one is seen in Figure 3.23). Done properly, we use a switch across this resistor that only opens when the voltage rises above some threshold — that prevents us from spending energy to make the resistor hot.
2. By the magic of SI units, the values of k_t and k_e are equivalent (although they have different physical meanings and units). One way to intuit this is that they both describe the interaction between the rotor and stator's magnetic fields: k_t tells us ~ how much the stator's field *pulls* on the rotor's field, and

k_e tells us ~ how much the rotor's field *drives voltage* in the stator's windings. For a better explainer on that topic, see [278].

Most of these values can be easily measured in the real world: R is a simple multimeter test, and L can be measured using an LCR probe. V and A are measured in the motor controller's firmware, Θ (and its derivatives for velocity ω_m and acceleration $\dot{\omega_m}$) are also measured in firmware (using an encoder and Kalman filter). Rotor Inertia J is typically included in a motor datasheet, but I have at times taken motors apart to weigh and measure this when datasheets seem untrustworthy. That leaves k_t, k_e and k_d to fit computationally, which we will see later.

4.3.2 Kinematic Models

Except in some rare cases, motors don't drive machine components directly. They are connected via belt drives, leadscrews and ballscrews, gearboxes (etc) or combinations of each. Understanding the machine's kinematics enables us to translate between the machine motion we are interested in controlling and the motor models that constrain that motion.

Kinematic models for machines and robots are typically described as rigid bodies connected in kinematic chains using Homogeneous Transformation Matrices (HTMs). These encode translations and rotations. Their effects can be combined by multiplying matrices along the kinematic chain, and to invert a transform we simply take that HTM's inverse, kind of tautological. Parallel kinematics can be more difficult to describe with HTMs, but we can normally simply pick one side of the parallel mechanism to model.

4.3.2.1 Cartesian to Actuator Coordinates

In this thesis, I wanted to use simpler kinematic descriptions that could easily be articulated as code. For machines, these are also often all we need. In fact for linear kinematic systems we can mostly get by with only the inverse kinematics. I describe these as simple Python functions, an approach that is shared by Gestalt [71] and is common also in open source machine control firmwares (where functions are instead written in C [78], [195]).

So, to start with the model formulations I use here, we have a vector of actuator positions $\vec{x}_{\text{actu}}$ and positions in cartesian space, $\vec{x}_{\text{cart}}$. For any given machine system, we have a function that describes there inverse kinematics - from the cartesian position to the actuator position. Generically, this is just:

$$\vec{x}_{\text{actu}} = f_{IK}(\vec{x}_{\text{cart}})$$

Importantly for our models, $\vec{x}_{\text{actu}}$ units are in *radians* and $\vec{x}_{\text{cart}}$ units are in *meters*. We will be looking at two systems in this chapter and the next two, one is a CoreXY based FFF 3D printer and the other is a (kinematically) simple milling machine, their kinematic functions are below.

Listing 4.1: Inverse kinematics definition for the FrankenPrusa (5.4.2), with conversion between motor units (*radians*) and machine coordinates (*mm*), and the corexy transform.

```
# a, b: 32mm travel per rotation
# (and inverted direction)
# z: 8mm per rotation

rads_per_m = np.array([
  -1.0 * (2.0 * np.pi) / 0.032,
  -1.0 * (2.0 * np.pi) / 0.032,
  (2.0 * np.pi) / 0.008,
])

def cart_2_actu_corexy(xyz):
  abz = np.array([
    (xyz[:, 0]-xyz[:, 1]) * rads_per_m[0],
    (xyz[:, 0]+xyz[:, 1]) * rads_per_m[1],
   (xyz[:, 2])           * rads_per_m[2],
  ])

  return abz
```

Listing 4.2: Inverse kinematics definition for the CNC milling machine from (6.4.1), with conversion between motor units and machine coordinates, and a trivial linear kinematic function.

```
# x, y: 8mm travel per rotation
# z: 4mm per rotation

rads_per_m = np.array([
  (2.0 * np.pi) / 0.008,
  (2.0 * np.pi) / 0.008,
  (2.0 * np.pi) / 0.004,
])

def cart_2_actu_mill(xyz_cart):
  xyz_actu = np.array([
    xyz_cart[:, 0] * rads_per_m[0],
    xyz_cart[:, 1] * rads_per_m[1],
    xyz_cart[:, 2] * rads_per_m[2]
  ])

  return xyz_actu
```

4.3.2.2 Applying Coulomb Friction

We need to add friction to these models. I use an approximation for Coulomb friction that combines a viscous term $\mu_{f,v}$ that is multiplied by speed, with an offset term $\mu_{f,static}$. The output from this model (defined with 4.3) is shown below in 4.5. In the rest of the section, I use the symbol $F_c(v)$ to define this friction function.

I had initially tried to fit models with viscous damping alone, but found that most machines have significant static friction. The function that describes Coulomb friction is simple but nonlinear. It is also *sharp;* its derivative briefly goes to infinity at the origin. These are two properties that make classical controls methods twinge, the jump from linear to nonlinear control representing a substantial step in complexity. Our motor models' saturations (we can easily increase voltage applied to the motor, until we hit the maximum available from our power supply) are also simple but nonlinear and have sharp derivatives (dropping suddenly to zero at the saturation).

Listing 4.3: The coulomb friction model as implemented in Python, using *tanh* to smooth the function near the origin, and an heuristic parameter to define the interval over which *tanh* is applied.

```
def coulomb_friction_force(offset, slope, speed, smooth_interval=0.001):
    return np.tanh(speed * (1.0 / smooth_interval)) * offset + slope * speed
```

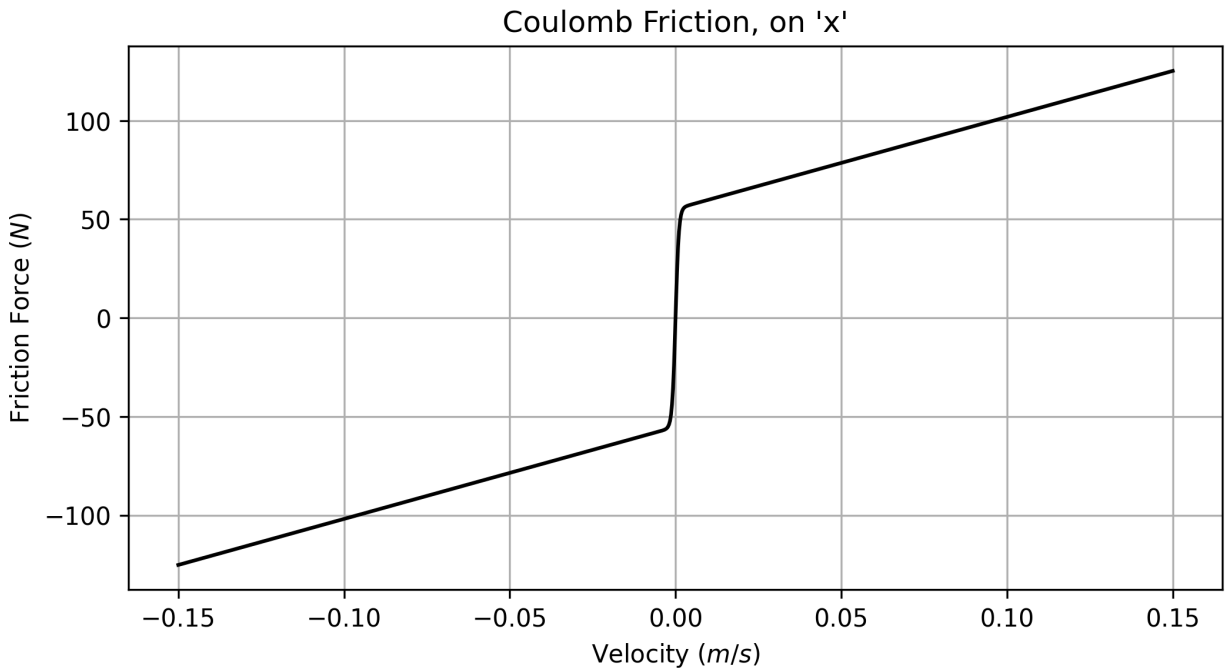


Figure 4.5: Coulomb friction, rendered with parameters from the x-axis on the milling machine in this thesis. We can see subtle smoothing of the function near the origin, which prevents the function's derivative from jumping to infinity at that point.

In the velocity planner that I develop in Section 4.6 the nonlinear property is easily managed numerically, but the sharp derivatives are not OK because the optimizer relies on gradients to make choices about how to update the solution. This is why *tanh* appears in my implementation of coulomb friction; it approximates static friction but *smoothes* it near the origin. I apply a similar trick with motor model saturations (and in a few other places in the planner, see Section 4.6.3).

4.3.2.3 Inertias

We also need inertias / masses. I *measure* these in kg for axes (M_{cart}) and $kg \cdot m^2$ for rotary bodies e.g. motor rotors, leadscrews and couplers (J_{actu}). Measurements are easy enough to make when we are assembling a new machine (a kitchen scale suffices, alongside radii of parts for rotary inertias) and given the uncertainty in system components that are more difficult to measure (like friction) I adopted this practice instead of trying to fit mass estimates alongside other kinematic parameters — but see Section 8.2 for some discussion on how we might estimate all of these parameters in a more "end-to-end" manner, which would be important for retrofitting cumbersome or difficult to disassemble equipment.

4.3.3 Combining Models for Constraint-Based Solving

To calculate actuator constraints in the planner, I combine these models such that we can predict the current required to drive our machine given any velocity and acceleration state at its tool tip,

and our actuators' current limits in those configurations. I show these steps in Equation (4.5) (for the CNC machine) and (4.6) (for the 3D printer) below.

4.3.3.1 Required Actuator Current from Cartesian Velocities and Accelerations

The equation below shows how I combine machine kinematics f_{IK} with known system states $\vec{a}_{\text{cart}}, \vec{v}_{\text{cart}}$ and a damping model $Fc(v)$ to estimate required current for the CNC milling machine. In this model we have damping terms on the cartesian system, but not on the actuator system, because those parameters would be degenerate if we tried to fit both sets. Instead, we just fit one set of damping terms, which is sufficient since damping of the motors is directly proportional to damping in the cartesian system: the one term effectively captures both.

$$\begin{aligned}
\vec{F}_{\text{req,cart}} &= \vec{a}_{\text{cart}} \cdot M_{\text{cart}} + Fc(\vec{v}_{\text{cart}}) + 9.8 \cdot M_z \\
\vec{a}_{\text{actu}} &= f_{IK}(\vec{a}_{\text{cart}}) \\
\vec{\tau}_{\text{req,actu}} &= f_{IK}(\vec{F}_{\text{req,cart}}) + \vec{a}_{\text{actu}} \cdot J_{\text{actu}} \\
\vec{I}_{\text{req,actu}} &= \vec{\tau}_{\text{req,actu}} / a_{kt}
\end{aligned} \tag{4.5}$$

Below is the same but for the 3D printer. They both work in a similar manner. First we calculate the force required to move the cartesian system around, which is just $F = ma + F_f$ we then take cartesian states and transform them to actuator states: for linear kinematics, we can use the same $f_{IK}()$ to transform velocity, accelerations, or forces (so long as we use SI units). We add to the required torque ($\vec{\tau}_{\text{req,actu}}$) the requirement for spinning the motor rotor and (in the case of the 3D printer, where rotor velocity is not a 1:1 combination of one axis' cartesian velocity), we fit a set of damping terms in the actuator space as well. Current estimation is then simple, dividing required torque by the actuators' k_t.

$$\begin{aligned}
\vec{F}_{\text{req,cart}} &= \vec{a}_{\text{cart}} \cdot M_{\text{cart}} + Fc(\vec{v}_{\text{cart}}) \\
\vec{a}_{\text{actu}} &= f_{IK}(\vec{a}_{\text{cart}}) \\
\vec{v}_{\text{actu}} &= f_{IK}(\vec{v}_{\text{cart}}) \\
\vec{\tau}_{\text{req,actu}} &= f_{IK}(\vec{F}_{\text{req,cart}}) + \vec{a}_{\text{actu}} \cdot J_{\text{actu}} + Fc(\vec{a}_{\text{actu}}) \\
\vec{I}_{\text{req,actu}} &= \vec{\tau}_{\text{req,actu}} / a_{kt}
\end{aligned} \tag{4.6}$$

This step uses machine parameters that have been loaded in the planner's configuration: kinematic functions (authored as Python functions), along with damping and mass parameters and motor model parameters. I use the same formulations *also* to fit models to data because we can easily collect time-series velocity and acceleration states of the machine using our motors and planners. This is a useful property of the planner's formulation. Especially because the planner is authored in Python (which has an excellent set of function fitting tools) we can actually use the exact same block of code for both tasks.

Value	In Motor Models	For Limit Calculation
Applied Voltage, at supply limit	V_a	$V_{\text{drv,max}}$
Driver current limit	n/a	$I_{\text{drv,max}}$
Limits to Potential Voltage	V_p	$V_{\text{lim}^+}, V_{\text{lim}^-}$
Limits to Motor Current	I	$I_{\text{lim}^+}, I_{\text{lim}^-}$

Table 4.2: Some new symbols for this step, to define boundaries of motor model parameters, and driving states.

4.3.3.2 Actuator Current Limits from Actuator Velocities

The step above includes calculation of the actuator velocities $\vec{v}_{\text{actu}}$ which we can use to calculate each actuator's real current limits $I_{\text{lim}}, \dot{I}_{\text{lim}}$ for each state. In the motion planning step, calculating these limits is important because they represent the actuator constraints.

The *power supply* voltage and *driver* current limits[82] are both saturations not described by the motor models themselves. Supply and driver saturations are one of the most common sources of nonlinearity in real systems because they pop up even in where the dynamics are otherwise linear — nobody has a power supply with infinite available voltage! To apply these saturations *without* causing gradients of the models to vanish (see Section 4.6.3), I use tanh to clip values. This also leads overall more conservative use of actuators than a simple clip would.

So, combining some parts of the electrical motor equations (4.3) with those saturations, we drive the motor models with maximum supply voltage to find the current limit. These functions are also applied with negative-signed supply voltages to produce lower bounds for each, V_{lim^-} and I_{lim^-}.

$$\begin{aligned} Z &= \sqrt{R^2 + (L \cdot \omega_e)^2} \\ V_{\text{lim}^+} &= \tanh\left(\frac{(V_{\text{drv,max}} - k_e\omega_m)}{V_{\text{drv,max}}}\right) \cdot V_{\text{drv,max}} \\ I_{\text{lim}^+} &= \tanh\left(\frac{V_{\text{max}}/Z}{I_{\text{drv,max}}}\right) \cdot I_{\text{drv,max}} \end{aligned} \tag{4.7}$$

We also need limits for the rate of change of current $\dot{I}_{\text{lim}}$. In steady state, this is just the applied voltage divided by the speed-varying impedance (fifth line of Equation (4.3) from 4.3.1). This is the value that I use in the planner (4.6.1), again with positive and negative limits.

$$\dot{I}_{\text{lim}^+} = \frac{V_{\text{drv,max}}}{L} \tag{4.8}$$

This is technically only true when the motor is at a standstill. Looking back to the *third* line of Equation (4.3) shows that our real limit actually changes with speed (via the change above to V_{lim}):

$$\dot{I}_{\text{lim}^+} = (V_{\text{lim}^+} - RI)/L \tag{4.9}$$

[82]This saturation is just a system configuration that I set by hand per actuator based on my knowledge of that device's amperage rating. In the fullness of time this could be defined in motor firmwares and read from *their* configurations on the fly (or saved alongside motor model definitions) to save a feed-forward configuration step.

There is nothing structural in the planner that prevents me from using the second, more complete definition for $\dot{I}_{\text{lim}}$ — I simply did not implement this because the steady-state approximation is good enough for our purposes.

4.4 Field-Oriented Control on (almost) Any Stepper

Developing this Field-Oriented Control (FOC) motor controller was a key enabler of the research in this thesis because it turns motors into equal parts actuator and sensor; the controller can both drive and measure stator currents, and measures rotor position for commutation[83] and for position / velocity control. When we combine those readings with motor models, the current measurements become torque measurements via $\tau = k_t I$ from Equation (4.4). This is known as *proprioception* and is a common method in the field of robotics [279].

Most proprioceptive actuators are based on BLDC (Brushless DC) motors, but I elected to develop this controller for stepper motors. In Section 4.3.1 I gave a brief overview of these motors: they are ubiquitous in all types of machines and especially common in simpler equipment because they do not *require* closed-loop control. Because they have very many electrical rotations for every mechanical rotation (i.e. they have many "pole pairs"), a controller that sequentially "steps" through those electrical rotations can position the motor using dead reckoning. However, when stepper motors are loaded beyond their limits they "miss steps" and higher level controllers loose track of the machine's position. Although some modern drivers can detect skipped steps, for many other reasons it is advantageous to drive them under "full" closed-loop control: this can reduce errors, improve precision, and above all provide us with dynamical feedback; open-loop controllers do not track the internal states (like current and rotor angle) that are most useful in higher level controllers.

So, to drive steppers as closed-loop motors I am using an architecture called FOC (Field Oriented Control), which is *also* more commonly used for BLDCs but which can be deployed on almost any motor architecture. To implement the FOC algorithm I relied heavily on Dave Wilson's excellent blog posts and video seminars on the topic [280]. It is not common to drive stepper motors in this way, but there are a few examples of other implementations in open source hardware communities but none that couple the FOC algorithm to motor modelling methods and motion control etc. The main motivation to develop this type of control for steppers arises from their ubiquity and cost, but in Section 7.5 I share some more thoughts on why this type of motor architecture may actually be better *overall* for machine control.

Driving steppers with FOC is especially difficult because their high pole pair count means that commutation must be performed very quickly even at relatively low rotor velocities. But, high performance embedded computing is becoming cheaper. This circuit uses a SAMD51 microcontroller, two DRV8251A H-Bridges (which integrate current sensing hardware) and an AS5047P magnetic encoder with 14 bits of resolution. The encoder is contactless and reads field direction from a diametrically magnetized disk that can be attached to the rear shaft of a stepper motor. Despite the SAMD51′s relatively high performance (at 120 MHz and including an FPU), its cycles are nearly entirely used up to operate the control interrupts in the firmware that I developed here.

[83] Commutation is the process of rotating a motor stator's magnetic field such that it is properly aligned to the rotor's magnetic field.

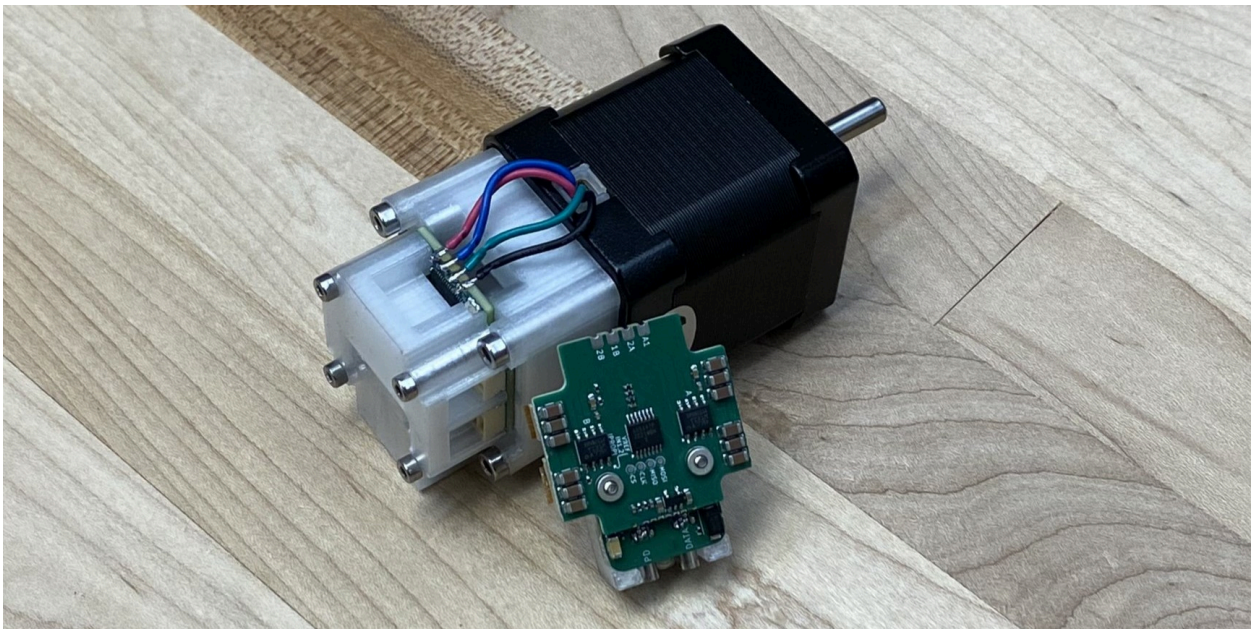


Figure 4.6: The motor driver I developed in order to complete the work in this thesis. The board can be mounted onto "almost any" stepper motor with the small addition of a diametric magnet being attached to the shaft.

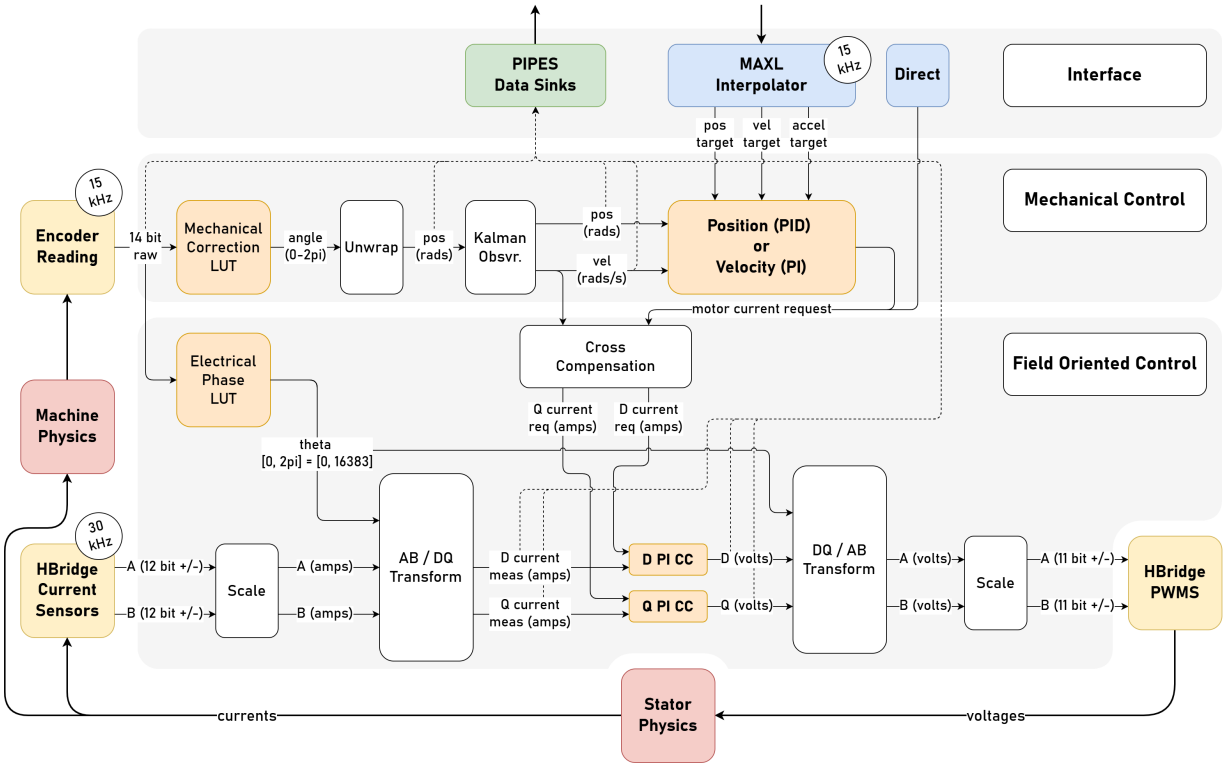


Figure 4.7: A diagram of the FOC control algorithm as implemented on the motor driver, showing the main hardware inputs and outputs (yellow), the MAXL interpolator (blue) that drives its closed-loop position and velocity controller, key control components (orange) and internal states that are available as PIPES outputs (dashed lines towards the green block).

The circuit is also limited in total drive power by the DRV8251A's maximum current rating (at $4.5A$ each) and the board's small footprint, which limits heat flow out of the driver.

The controller's layout is diagrammed in Figure 4.7. FOC manages the lowest level: current control and commutation. At the heart of FOC are two PI current controllers for the D and Q axes. These are *aligned to the rotor's reference frame* such that D is *direct* (aligned to rotor north) and Q is *quadrature* to the rotor (90° from D). Only the current that is aligned with Q produces torque, so typically the D controller works to drive current there to zero and the Q controller works to drive whichever current is requested by any higher level controller. Key to the FOC algorithm are the AB/DQ and DQ/AB transforms: current is measured by the H-Bridges in the AB frame, which is static. By transforming these readings into the DQ frame, the lower level current controllers don't have to track rapidly oscillating sinusoidal commands which greatly reduces their bandwidth requirements. They generate voltage signals in DQ which are then transformed back into the AB frame before being applied to the H-Bridges. There are a handful of other implementation-specific details that would be important to reproduce this work, but they are not novel research. I included them in an earlier blog post [281] but here I will stick with the details that are immediately relevant to the rest of the thesis.

One layer up, the controller implements *either* a second PI controller for velocity control, or a PID controller for position control. The velocity and position controllers can be controlled directly or configured to track the output of a MAXL spline interpolator. The spline additionally produces target velocities and accelerations, which can be combined with feed-forward terms for either to improve performance. For example in a system with known dynamics, the amount of current required to generate a target acceleration or overcome friction at a certain speed is known. To improve estimates of the rotor position, velocity and acceleration the controller implements a Kalman filter [282], [283].

4.4.0.1 Encoder Calibration

The magnetic encoder used here has some nonlinearity because the magnet is never perfectly aligned with the hall sensors in the encoder. Especially because the motor's pole pairs are so close together, calibrating these nonlinearities away is important. Luckily the encoder provides absolute readings, so one calibration can be performed for each motor. This produces two look-up tables (LUTs); one relates encoder readings to rotor electrical angles, and the second relates encoder readings to rotor mechanical angles. Both apply the same offset, but the electrical LUT has 50 times as many rotations (for the 50 pole pairs) and outputs 16 bit values whereas the mechanical LUT outputs floating point values in $0 \to 2\pi$ — angular measurements in radians. I use integer values for the electrical LUT because the AB/DQ and DQ/AB transforms themselves use look-up tables to perform their trigonometric calculations. This is faster than relying on `sin` and `cos` functions, which can take many cycles each; the LUTs trade memory for computation. Equally, the encoder calibrations could be applied using computation but the LUT is faster here too.

This is another place where reconfigurable access to the motor is valuable. To fit the encoder LUTs I configure a MAXL / PIPES script to drive the motor, attaching the spline to the motor's position target but driving the motor using *open loop* current control, and use a MAXL OneDOF

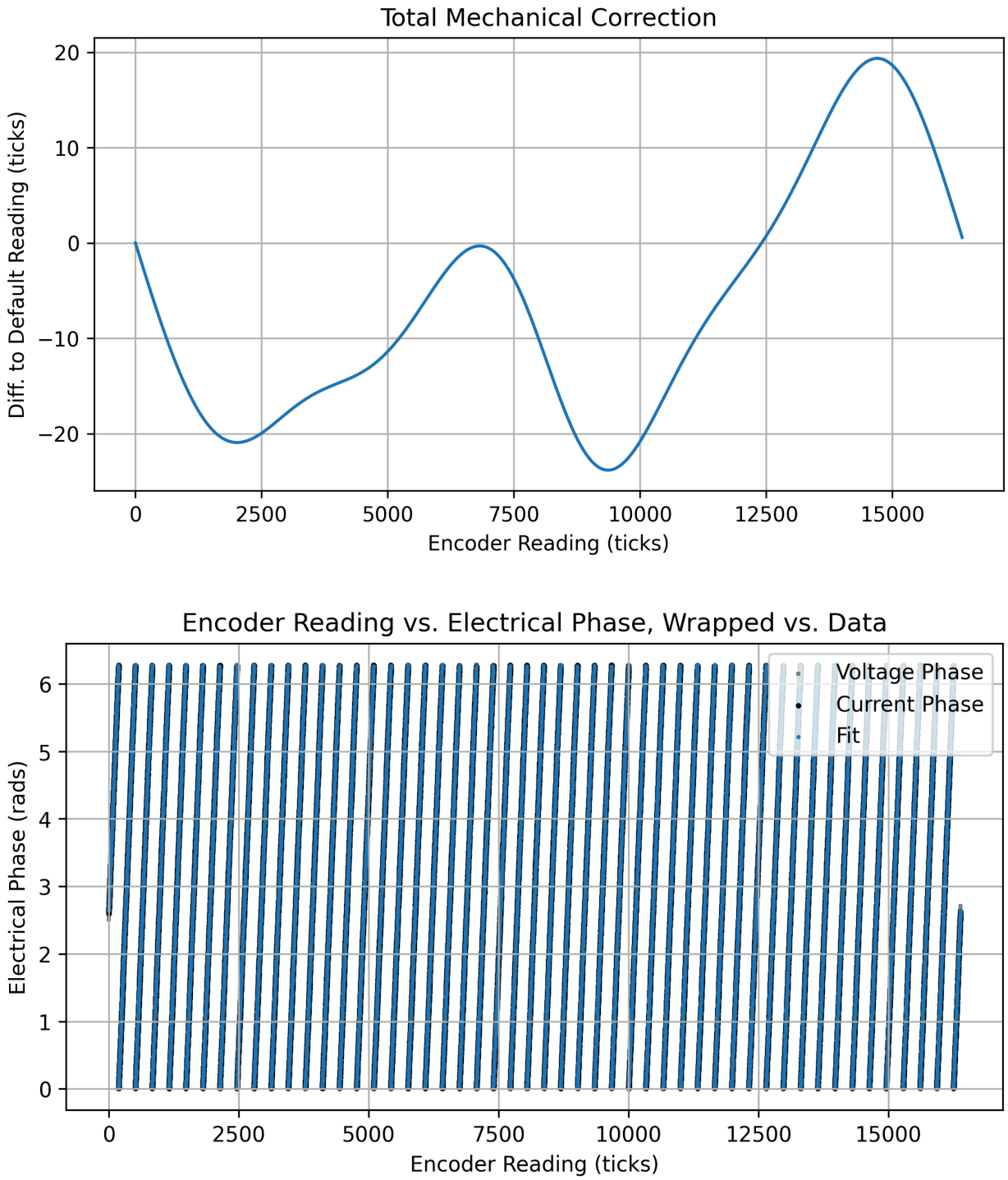


Figure 4.8: The encoder calibration is critical to the motor's operation because the encoder's readings are nonlinear and FOC requires that we accurately map between rotor angles and electrical angles. Above is a plot of the total mechanical correction applied to an encoder, and below shows how this correction is applied to produce an electrical angle for each mechanical angle.

block to rotate the motor at a set velocity while reading raw encoder, current, and voltage signals back through a Pipes Function that transmits low level motor states. I can then analyze these

readings in a script that fits a set of correcting function parameters to produce the interpolations in Figure 4.8. This then writes look-up tables as `.h` files that are loaded into the motor's firmware. This last step could presumably be automated as well, but the simpler approach works well for the time being. Low-level control of the motor is then also used to drive quadrature current requests directly in order to fit motor models in Section 4.5.1.

4.4.0.2 Current, Velocity and Position Control

In the motor's lower level control, gains are tuned by hand. There are actually quite a few parameters here, in Section 4.9.3.2 I discuss how this is one of the key "missing components" to the motion control architecture overall, but tuning these using models is well understood and so would presumably be a simple addition.

Firstly there are PI gains in the current controllers. Luckily they can be set directly via the motor's R and L using the method developed by Dave Wilson [280] (R, L themselves are easily measured, see Section 4.5.1).

$$k_{p,PI} = R$$
$$k_{i,PI} = 2\pi f_c \cdot L$$

The PI / PID gains for the velocity or position controllers are a different story; tuning them properly requires a complete model not only of the motor's electrical half but also how the rotor interacts with the rest of the machine. I do collect and fit all of those models in this chapter, but I did not connect them to automatic PID tuning — instead I simply tune these by hand. To do so I use a PIPES / MAXL configuration using a chirp generator that drives machines across a range of velocities and accelerations and cycle through tests, analysis and gains updates; this is described in more detail in Section 4.5.2.

These controllers have bandwidth limits because they rely on *error* signals to produce outputs. The feed-forward terms that I mentioned above can help, but the bandwidth limits do impose additional constraints on control overall. Luckily here they are quite fast; the current controllers run at 30 kHz which is well above the electrical bandwidth of most stepper motors and the velocity / position controller runs at 15 kHz.

One of the key limits for the velocity controller is that it uses a filtered velocity estimate that has a time-constant of about one millisecond. Velocity control for the extruder system poses some problems in the 3D printer's extruder system (see Sections 5.9.1 and 5.11.7.5)

4.4.1 The Motor's Software and Data Interfaces

The motor is attached to higher level controllers using PIPES / MAXL via the spline interpolator, and a set of Pipes Functions are used to configure its control modes (i.e. whether the spline outputs are used to drive the velocity controller, position controller, or other lower levels).

The driver also streams internal control states back to data collectors and controllers using PIPES. I normally do this by configuring one of these functions to run at a fixed interval to produce

Value	Units	Data Type	Name
System time at sample generation	μs	`uint64`	timestamp
Position from the interpolated MAXL spline	$rads$	`float`	maxl_position
Velocity	$rads/s$	`float`	maxl_velocity
Acceleration	$rads/s^2$	`float`	maxl_accel
Target current from internal pos / vel controller	$Amps$	`float`	cur_cmd
Measured current on the Q axis	$Amps$	`float`	cur_q
Kalman estimate of unwrapped angular position	$rads$	`float`	obsv_position
Kalman estimate of angular rate	$rads/s$	`float`	obsv_velocity
Kalman estimate of angular acceleration	$rads/s^2$	`float`	obsv_accel
Controller error flags	bitmask	`uint8`	ctrl_flags
MAXL error flags	bitmask	`uint8`	maxl_flags

Table 4.3: Motor motion states that can be transmitted via PIPES at runtime.

Value	Units	Data Type	Name
System time at sample generation	μs	`uint64`	timestamp
Raw encoder reading $\in [0, 16383]$	quantized	`uint16`	encoder_raw
Position from the interpolated MAXL spline	$rads$	`float`	maxl_pos
Velocity from the spline	$rads/s$	`float`	maxl_vel
Acceleration from the spline	$rads/s^2$	`float`	maxl_accl
Kalman estimate of unwrapped angular position	$rads$	`float`	obsv_pos
Kalman estimate of angular rate	$rads/s$	`float`	obsv_vel
Kalman estimate of angular acceleration	$rads/s^2$	`float`	obsv_accl
Measured voltage on the supply lines	$Volts$	`float`	vcc_rails
Voltage applied on the D axis	$Volts$	`float`	vcc_d
Voltage applied on the Q axis	$Volts$	`float`	vcc_q
Target current from internal pos / vel controller	$Amps$	`float`	cur_cmd
Contribution of the P term to the current command	$Amps$	`float`	pid_p
Contribution of the I term	$Amps$	`float`	pid_i
Contribution of the D term	$Amps$	`float`	pid_d
Measured current on the D axis	$Amps$	`float`	cur_d
Measured current on the Q axis	$Amps$	`float`	cur_q
Controller error flags	bitmask	`uint8`	ctrl_flags
MAXL error flags	bitmask	`uint8`	maxl_flags
Index of last-inserted control point in ring buffer	index	`uint8`	maxl_insert
Timestamp of closest (in the past) control point	μs	`uint64`	last_eval
Timestamp of last control point wiped from buffer	μs	`uint64`	last_tail

Table 4.4: A more extensive list of states (and debugging values) that I use during motor modelling.

streams of data in time-series. A short list of the most important states in one of those streams are listed in 4.3 below. States are collected in each loop of the controller's main step (at 15 kHz) and the bundle of them is timestamped against the OSAP system time so that the exact measurement time can be reconciled with other system states.

The actual data rate that these streams run at is configurable using PIPES, and the maximum possible rate is a function of underlying network performance and traffic; I typically set this at 500 HZ which is above most machine's physical bandwidth — see i.e. Figures 4.25 or 4.32. In Section 2.6.2 I compare two other state-of-the-art machine control efforts; they collect smaller data streams (just motor position, velocity, and current) at lower data rates (250 and 100 Hz). I write a more complete systems-level timing analysis in Section 2.7.9.

I also include here Table 4.4, a more extensive set of values that I use during motor modelling, PID tuning, and motor firmware debugging.

Finally, the driver's control parameters and configurations can all be read or written remotely using Pipes Functions (normally as RPCs). Besides the encoder calibration, all of these values are remotely updated when systems are initialized and so motors do not need to themselves store or "remember" any of these values in their own memory; configurations are tracked at a systems level.

4.5 Fitting Motion Models

Given the models from 4.3, and our motor above, this section describes how I fit free parameters (and which ones I simply measure directly).

4.5.1 Fitting Motor Models

Table 4.1 renders the motor model parameters, and denotes which of these are measured directly (labelled *Measurement*) or measured by the controller and available as time-series measurements or easily calculated products of other measurements (labelled *Measurement(t), Product(t)*).

Measurements of static values are often straightforward. I measure R and L with an off-the-shelf LCR probe, the pole-pair count p is always 50 for stepper motors (and easy to count otherwise). The rotor inertia J can be measured, but only by disassembling the motor, taking dimensions of the rotor, and weighing it. This is cumbersome, and so I mostly rely on motor datasheets to include this value. In 4.9.3.2, I discuss how it is possible to add estimation of these parameters to this step.

However, that is for later. Here, we make them manually. Then we are left with the three "key" model parameters, k_t, k_e, k_d. To estimate those, I developed a simple program that runs the motor in a *direct current* mode at a number of pre-determined current targets. It runs each for a period that is long enough to get the rotor up to its steady-state velocity (where damping and back-emf overcome the motor/driver's ability to generate current). I run this program with the motor fixed on a desk, and mounted to a flywheel whose inertia has been measured.

Test data stores all the values output by the motor in Table 4.3 in time-series, as well as the test parameters. We can then reconstruct this time-series and fit our model against it.

I do this in two steps, using Python and scipy's *optimize* routine [284], [175]. First, we can fit the mechanical parameters k_t, k_d using the motor's measurements of its velocity, acceleration, and stator current:

$$A = (k_t I - k_d \omega_m)/J$$

Parameters are minimized against a cost function that compares parameters' acceleration estimates (from above) against acceleration measurements from the motor data, using sum squared error. The motor/actuator's rotor inertia J_a was manually measured previously, and is combined with the inertia of the test flywheel for this step. The flywheel does two things that are valuable here. First it increases the time that the motor spends accelerating and decelerating, increasing data density in high acceleration, high current, and low velocity states. It also has much more inertia than the motor rotor, which reduces overall model error that may come from a rotor inertia estimate miss because its contribution to inertia is so large compared to the motor rotor itself.

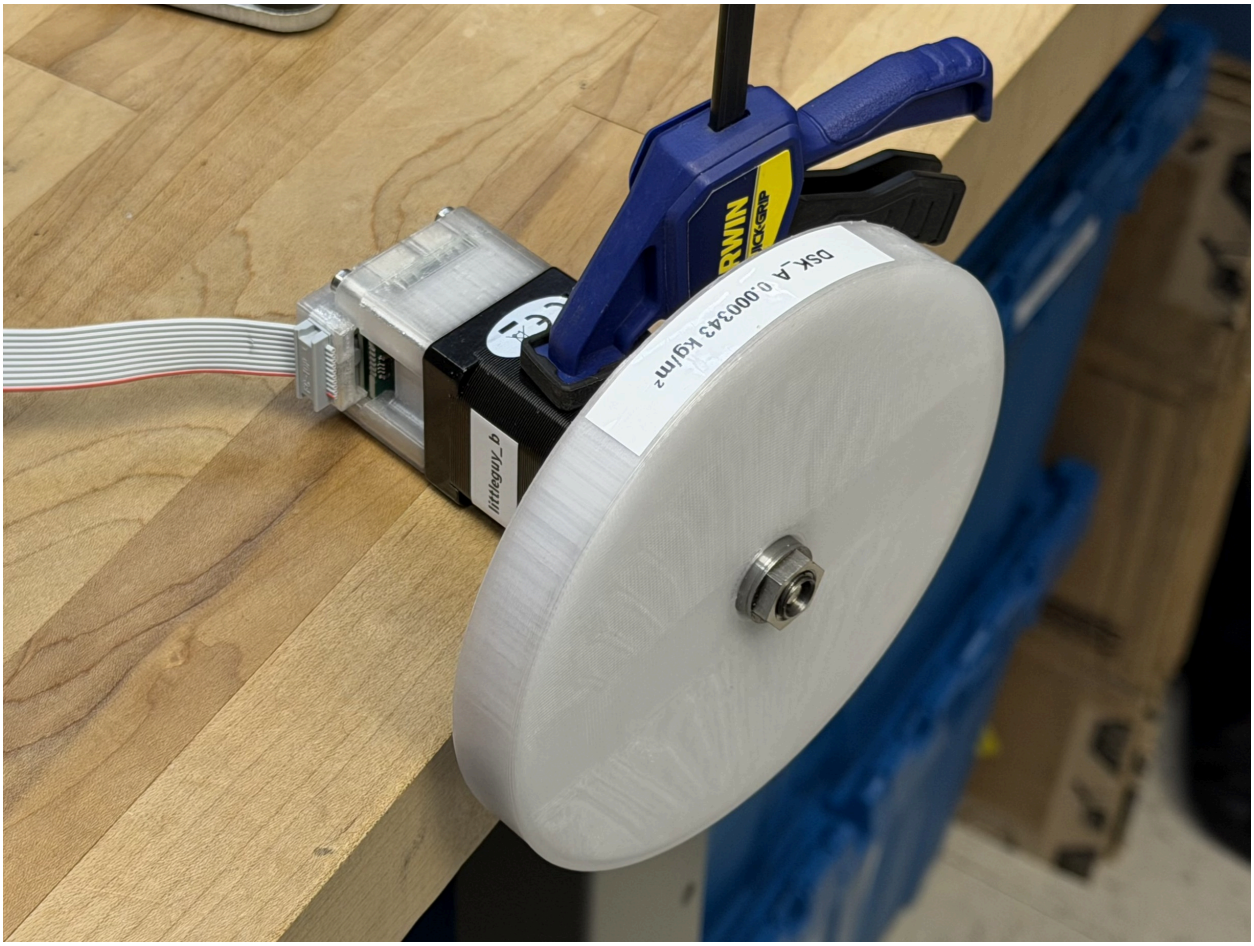


Figure 4.9: The FOC-Stepper mounted to the desk and attached to a disk with a known inertia.

To improve the fit, I also weight measurements by their distribution in velocity. This is useful because most measurements are taken when the motor is at it's steady-state (terminal) velocity, but we want to ensure the fit is equally biased across velocity states.

I estimate k_e separately, despite my note earlier (4.3.1) that the values for k_t and k_e should be the same. This is a bit of a hack, but is a valuable step because we can use this parameter to encode some more information about the motor and controller - that is, that the controller's commutation is not perfect and introduces a small amount of phase lag. When it sets voltage on

Listing 4.4: The motor's mechanical model, and loss function used to fit k_t, k_d.

```
def model_kt_kd(params, q_current, vel):
    kt, kd = params
    accel = (kt * q_current - kd * vel) / J
    return accel

def loss_kt_kd(params, cur_data, vel_data, acc_data):
    acc_guess = model_kt_kd(params, cur_data, vel_data)
    return np.sum(np.square(acc_guess - acc_data) * weights)
```

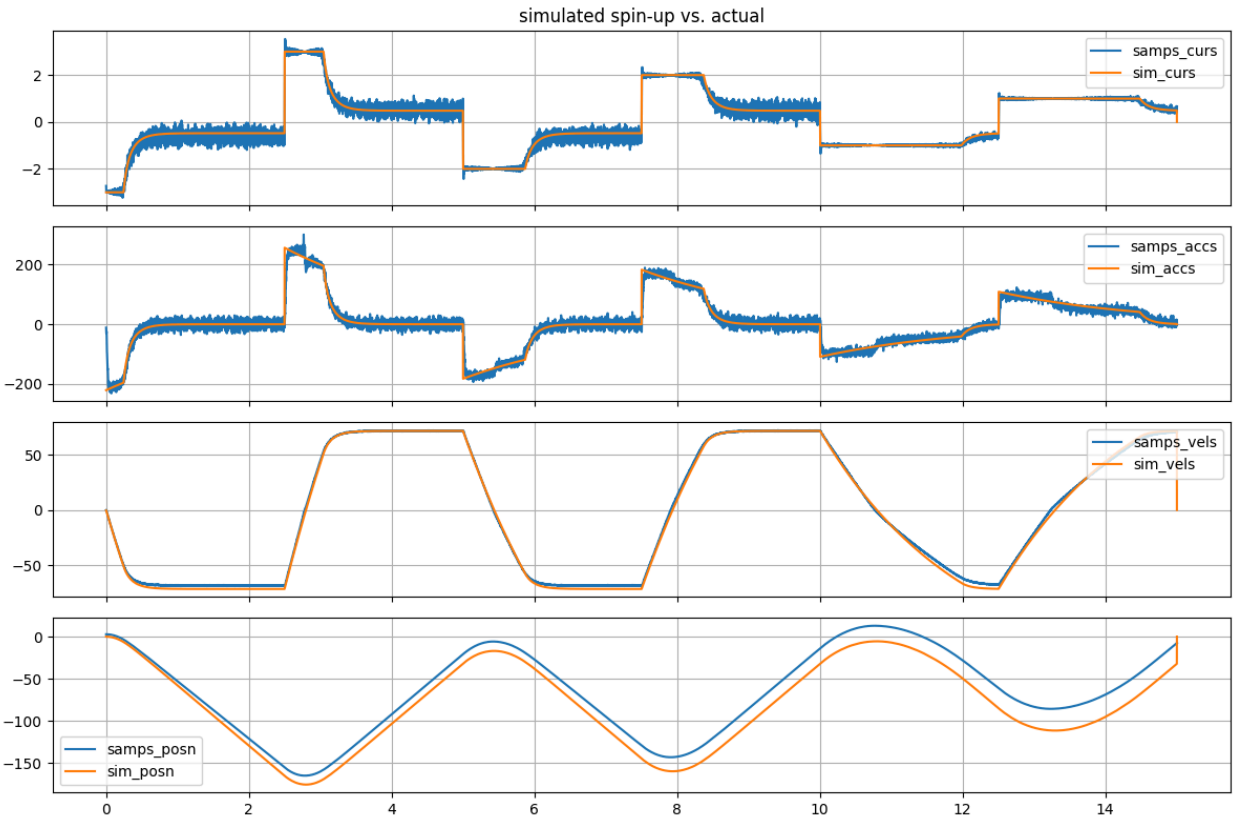


Figure 4.10: Simulation (orange) and data (blue) of time-series data from our model and motor during a "spin up" test.

the HBridge's PWMs, those voltages take some microseconds to be actualized in the drive. During those microseconds, the rotor continues spinning. This means that at high RPMs, it seems as though we have *more back-EMF,* that is, applied voltage becomes less effective.

Fitting this is similar, the model estimates the *current* measurement given the rotor velocity, some estimated k_e, and the voltage applied on the q axis. The estimated current is compared to the measured current, making a sum-squared error, which is used to fit the parameter using scipy's minimize routine.

To verify that models explain the motor system, we can plot time-series data against a simulated system, where the recorded control input (a target current) is used to drive the simulation.

Models can also be used to generate torque curves across a range of driver currents, as below:

Limit to torque generation in these models comes from two places. Firstly, the driver's current limits (or current set points) saturate. Above, you can see that the motor's torque is limited to $0.8N$ at zero velocity, simply because we cannot drive more than (in this case) $2.5A$ through the driver's H-Bridges. In practice this limit can be set also due to thermal limits in the motor. As the motor comes up to speed, current generation is limited by back-EMF and the motor's impedance, which grows as speed increases.

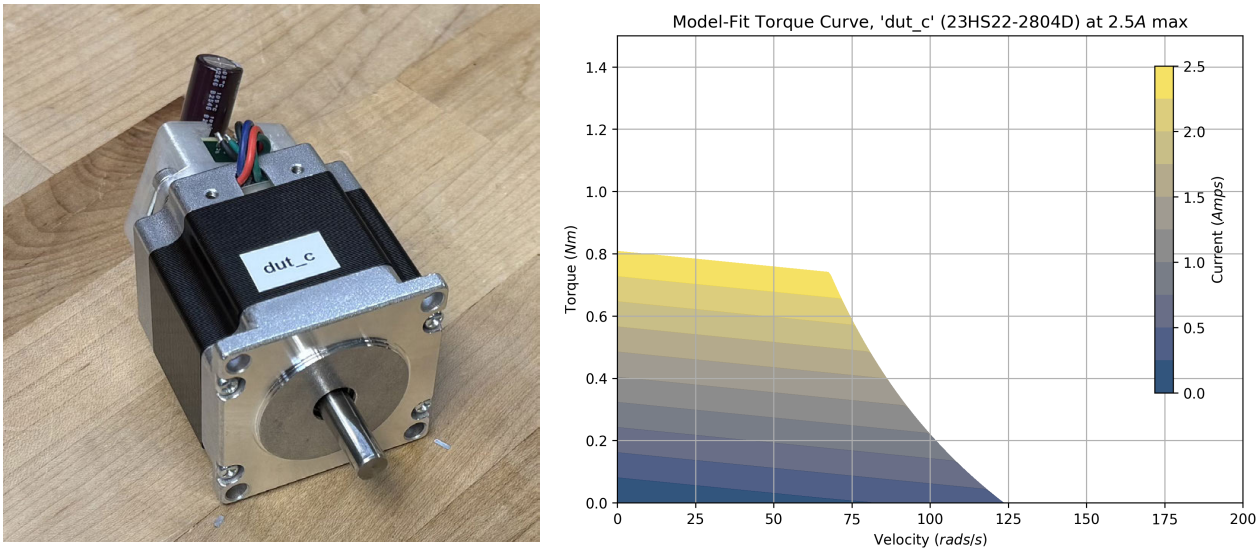


Figure 4.11: The torque curve from a motor-model fit (right), and the motor (left).

Torque curves are an excellent analysis tool, and they are commonly used by machine designers to select actuators. For example if we know that an axis of our machine will mostly be responsible for fast accelerations at low speeds, we will pick a motor with more torque "in the low end" whereas if we have a large gear ratio and need our motor to work at large RPMs, we will look for motors where the torque does not fall off drastically as speed increases. In Section 6.2.1, I show how we can combine motor and kinematic models to virtually test multiple combinations of each to pick motors using data from our hardware (rather than datasheets), and select velocity controller parameters (for trapezoidal planners) using the same.

4.5.2 Fitting Kinematic Models

Fitting kinematic parameters follows a similar pattern. Before we start,

1. Models for each of the machine's actuators are fit using the method above.
2. System inertias J, m are measured manually.
3. Inverse kinematics $f_{IK}()$ are written down as Python functions.
4. An initial set of PID gains are written for the actuators' position tracking low-level loop. These do not have to be well tuned, in fact I used the same starter gains for each actuator in all the systems that I developed in this thesis.

We then have a controllable system, and need to excite it in order to generate some test data. I use MAXL and PIPES to connect a chirp generator (2.5.4.3) to the axes that we are interested in fitting, using it to drive their positions. *Chirp* signals are time-varying sinusoids that slew from an initial to a final frequency (typically low to high) over a fixed time. Parameters for the chirp are configured manually. The system for one of these tests was shown in the systems chapter (Fig. 2.31). We can then collect time-series data for the test, an example of which is figured below in Figure 4.12.

I first use this data to tune the motors' PID gains, which I do iteratively. This is a somewhat involved process, because improvements to gains allow us to drive the system at higher chirp frequencies — so during this process I am heuristically learning about the systems' overall limits. However, it is a well understood process, so I will not elaborate except to note that Section 4.9.3.2 discusses how this could be automated in the future.

Even with data from poorly tuned gains, we can update our models. The only free parameters at this point in the kinematic models are the damping terms. As in the motor models, I set up a function that estimates the motor current using the motion states (acceleration and velocity) for each axis under test and a loss that compares that estimate to measurements from the dataset. This is the same system of equations as in Equation (4.5) or in Equation (4.6) when we are sampling data from a CoreXY based machine.

Fits can be visually verified by comparing their current estimates to measurements across time series data, as shown below. We can also extract global metrics from these time-series like average model estimate error (in *Amps*), which I expand on in Section 4.7.2.

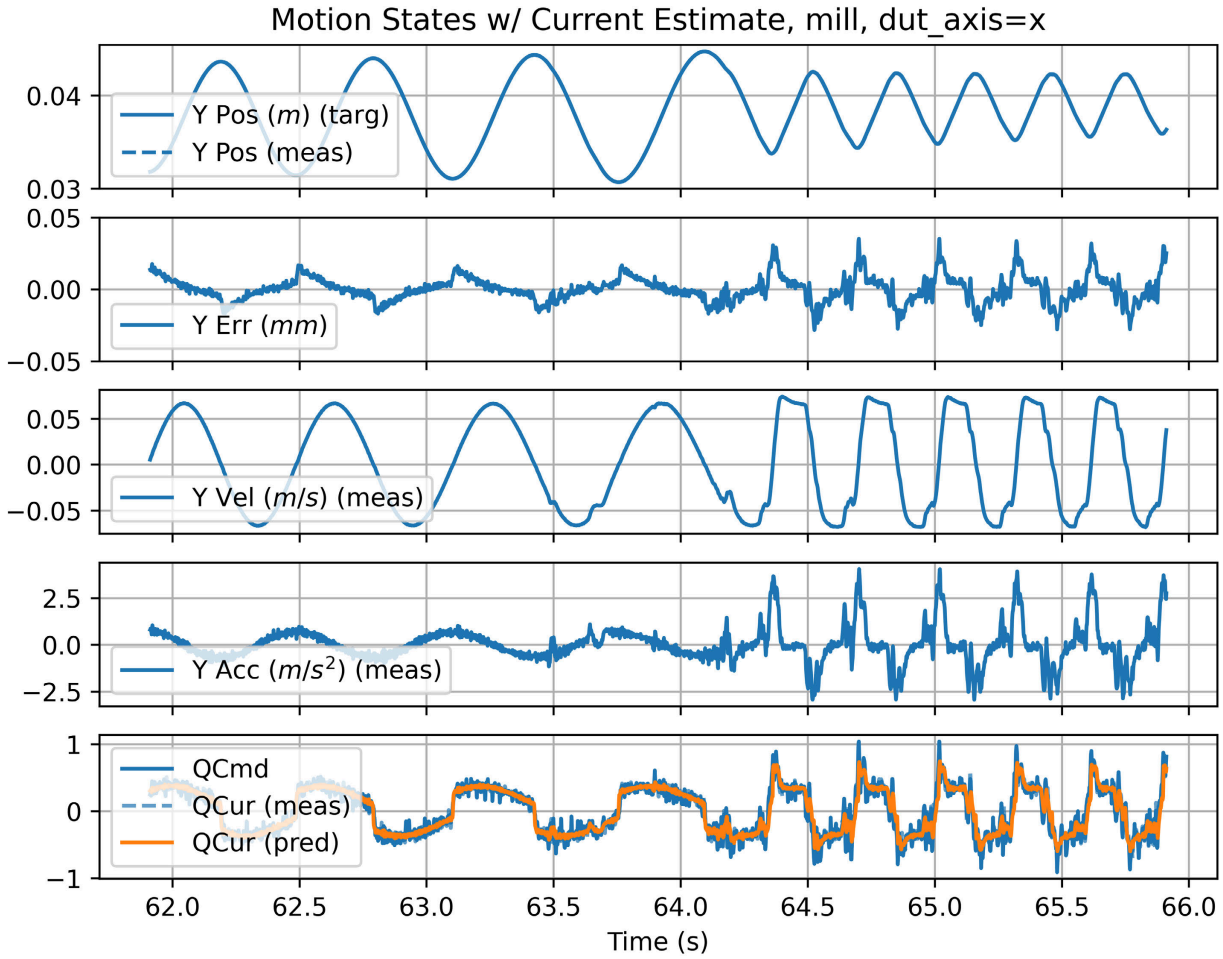


Figure 4.12: Time-series data from a test trajectory through a CNC milling machine's X axis, using measurements (in blue) to produce model-estimated motor currents (orange).

4.6 Optimization Based Velocity Planning

With models for our machines, we can move on to velocity planning. The goal here is to develop an optimizer that will maximize velocities along a target trajectory without exceeding actuator constraints. The planner that I develop here differs from trapezoid planners in a few key ways.

First, it uses machine models to describe constraints — this makes the planner configuration more direct: we describe the system, rather than interpreting the system's constraints as a set of parameters for maximum acceleration and maximum velocity. This also allows the planner to use more of the machine's underlying dynamic range, and lets us describe constraints that emerge in the actuator coordinate space *and* those that emerge in the machine's tool-tip (cartesian) coordinate space — this is valuable in kinematic systems where the two spaces are not identical, e.g. CoreXY. I discuss those improvements in Section 4.7.1. To tune the planner within those constraints, I use *actuator current deployment scalars* that de-tune the system as a percentage of total available actuator power[84], rather than as direct parameters.

Second, it can be more easily extended to capture constraints that emerge from our process descriptions, something that I explore at length in Chapter 5, in particular 5.2.3 (which discusses the state-of-the-art in this regard) and 5.9 (which discusses how this planner is extended for 3D printing processes). Section 4.7.1 shows how the same idea is applied (albeit in a much simpler manner) to the CNC machining process.

Optimization-based controllers are themselves *not* new to the world. They are popular in robotics, where MPC (Model Predictive Control, see Section 4.2.7) has become quite common over the last decade, and have been applied to CNC machines and one polymer extrusion system. More recently, RL (Reinforcement Learning, 4.2.7.2) is emerging as a dominant practice. RL is more appropriate for more complex tasks, and typically run at much lower bandwidths and resolutions than are required for machine control.

In this section I describe the motion control component of the planner, in Section 5.9, I describe how it can be extended to include process dynamics.

4.6.1 Optimization Formulation

This formulation is similar to the *direct transcription* method that I described in 4.2.7, but

[84]I discuss this in more length in Section 5.11.1, where it relates also to process limits. But the gist is like this: on a guitar amp, we can "turn it up to 10," this is straightforward: 10 is the maximum that the amp is capable of (11 would be cooler, but might break things). In trapezoidal planners, we write a maximum acceleration directly, e.g., setting $1500mm/s^2$ for the X axis, $1000mm/s^2$ for the Y axis etc. We then run the machine to check if this works well: if stepper motors skipped steps, the machine vibrated too much, etc. At least, this is the most common practice in *open source* machine design, we can also use models to find these parameters, as in Section 6.2.2, which is more common in professional engineering contexts. Nevertheless, expressing this tune as a *scalar* of the total available power is not a feature of any state-of-the-art motion controllers that I have surveyed or used.

different in two key ways. The first is that the planner picks velocities in *space*[85] rather than at fixed time intervals. I take input trajectories and break them up into very many, very small line segments (as $\vec{x}(s)$ below, with a distance of $50\mu m$ between each, although the value is tune-able, see Section 4.8.3).

Then, rather than choosing control inputs at each step as well as those velocities (and having the planner align those implicitly), we work backwards to calculate the control inputs *that would be required* to achieve those velocities, and compare those to actuator limits in our cost function: violating limits is much more expensive than going slowly.

This is also not *really* Model Predictive Control, although it does use models to predict optimal control outputs. Normally MPC feeds measured system states back into the start of new plans at each cycle, but here we are purely feed-forward. At a higher level, we fit models to hardware — so we could say that it is configured with feedback. Lower levels of the controller are feedback systems: the motors servo to position targets that the planner outputs. I discuss the fact that "feed-forward" does not mean "bad" in Section 7.5.

The planner is designed so that each step can be completed as a parallel computation over the whole trajectory. This structure is what enables the planner to be deployed on parallel computing hardware which greatly improves performance. Below (and diagrammed in Equation (4.10)) are the basic steps.

First we provide a target trajectory as input that includes the 3D positions $\vec{x}(s)$ where s is the length along the trajectory at any given point and then trajectory settings: maximum velocities $v_{\max}(s)$; in many cases we want our planner to travel at some *set* velocity, that goes here. The target trajectory also includes the deployment scalars for current and the rate of change of current $I_{\text{prop}}(s), \dot{I}_{\text{prop}}(s)$. In an important first step this trajectory is segmented from its incoming representation into many small line segments each of which is $0.05mm$ in length (configurable).

The planner *selects* $v(s)$, a candidate set of velocities along the trajectory for each point. We then calculate the accelerations required ($\vec{a}(s)$ in cartesian space) to accomplish these selections, see Equation (4.13) in 4.6.1.1 for some details on that calculation.

We then have the key states for acceleration and velocity in cartesian space ($\vec{a}(s), \vec{v}(s)$). Other planner constrain motion based maximum values for those states alone. In this planner, I broadcast those back through the models that I described in Section 4.3.3 to generate the planner's intermediate values and then compare those to actuator constraints. Those values are: the current *required* in each actuator, current generation *limits* for each (including nonlinear motor model saturations and velocity dependencies) and then (using a simple discrete derivative) requirements for the step change in current at each step $\dot{I}_{\text{req}}$ and limits for the same from Equation (4.8) in the motor modelling section 4.3.1.

[85]The *(position, velocity)* plot is known as *phase space* for these types of problems, see Section 4.8.2 for some discussion on why this is a particularly useful approach for this type of optimization.

$$\overbrace{\begin{bmatrix}\vec{x}\\ \vdots \\ s\end{bmatrix}\begin{bmatrix}v_{\max}\\ \vdots \\ s\end{bmatrix}\begin{bmatrix}I_{\text{prop}}\\ \vdots \\ s\end{bmatrix}\begin{bmatrix}\dot{I}_{\text{prop}}\\ \vdots \\ s\end{bmatrix}}^{\text{Input Trajectory}} \rightarrow \overbrace{\begin{bmatrix}v\\ \vdots \\ s\end{bmatrix}}^{\text{Select}} \rightarrow \overbrace{\begin{bmatrix}\vec{a}\\ \vdots \\ s\end{bmatrix}\begin{bmatrix}\Delta t\\ \vdots \\ s\end{bmatrix}}^{f_a()} \rightarrow \overbrace{\begin{bmatrix}\vec{I}_{\text{req}}\\ \vdots \\ s\end{bmatrix}\begin{bmatrix}\vec{I}_{\text{lim}}\\ \vdots \\ s\end{bmatrix}\begin{bmatrix}\vec{\dot{I}}_{\text{req}}\\ \vdots \\ s\end{bmatrix}\begin{bmatrix}\vec{\dot{I}}_{\text{lim}}\\ \vdots \\ s\end{bmatrix}}^{\text{Intermediate Values}} \tag{4.10}$$

Those intermediate values are sent into the cost function (see details in 4.6.1.3), where exceedances of actuator constraints I_{2big} are penalized. The cost function also receives all other values: the input trajectory, a copy of the planner's selections, accelerations, and velocities in the actuator space, etc. The total sum over Δt is added to the cost as well to encode time optimality (small number fast, good). If we like, it is straightforward to also add terms to the cost function that constrain any of these other states directly.

The cost function is subject to minimization via updates to the planner's selection of $v(s)$ using a generic gradient descent optimizer; I explain the actual update step and solving framework in Section 4.6.1.4.

$$f_{\text{cost}} = \sum_{s}\Big[I_{2big}(s)\Big] + \sum_{s}\Big[\Delta t(s)\Big] \tag{4.11}$$
$$\underset{v(s)}{\text{mincost}}$$

The planner iteratively improves the solution until a fixed number of cycles are completed (again, see 4.6.1.4 for details), after which time the planner needs to send new values to hardware and intermediate values to a data store for later analysis. That is diagrammed below and explained in some more detail in Section 4.6.1.5 on converting from the spatial representation to a time encoded representation. Integration of the planner with the remainder of the machine system is via PIPES/ MAXL (Chapter 2), explained in 4.6.2.

$$\overbrace{\begin{bmatrix}\vec{x}\\ \vdots \\ s\end{bmatrix}\begin{bmatrix}v\\ \vdots \\ s\end{bmatrix}\begin{bmatrix}\vec{a}\\ \vdots \\ s\end{bmatrix}\begin{bmatrix}\vec{I}_{\text{req}}\\ \vdots \\ s\end{bmatrix}\begin{bmatrix}\vec{I}_{\text{lim}}\\ \vdots \\ s\end{bmatrix}\begin{bmatrix}\vec{\dot{I}}_{\text{req}}\\ \vdots \\ s\end{bmatrix}\begin{bmatrix}\vec{\dot{I}}_{\text{lim}}\\ \vdots \\ s\end{bmatrix}}^{\text{Spatial Trajectory and Solution}} \rightarrow \overbrace{\begin{bmatrix}\Delta t\\ \vdots \\ s\end{bmatrix}}^{\forall s\ in\ t_{\text{cycle}}} \rightarrow \overbrace{\begin{bmatrix}\vec{x}\\ \vdots \\ t\end{bmatrix}\begin{bmatrix}\vec{a}\\ \vdots \\ t\end{bmatrix}\begin{bmatrix}\vec{I}_{\text{req}}\\ \vdots \\ t\end{bmatrix}...\begin{bmatrix}\vec{\dot{I}}_{\text{lim}}\\ \vdots \\ t\end{bmatrix}}^{\text{Time-Based Solution Outputs}} \tag{4.12}$$

I show samples of the planner's visualizer in two figures below, these include 2D trajectory visualizations and spatial- and time-series outputs of some intermediate values, as well as a planner convergence plot.

4.6.1.1 Calculating Required Accelerations from Velocities

The planner's velocity selections are for motion along the trajectory at fixed spatial steps. To calculate other system states, we need to transform those into accelerations at each point in all

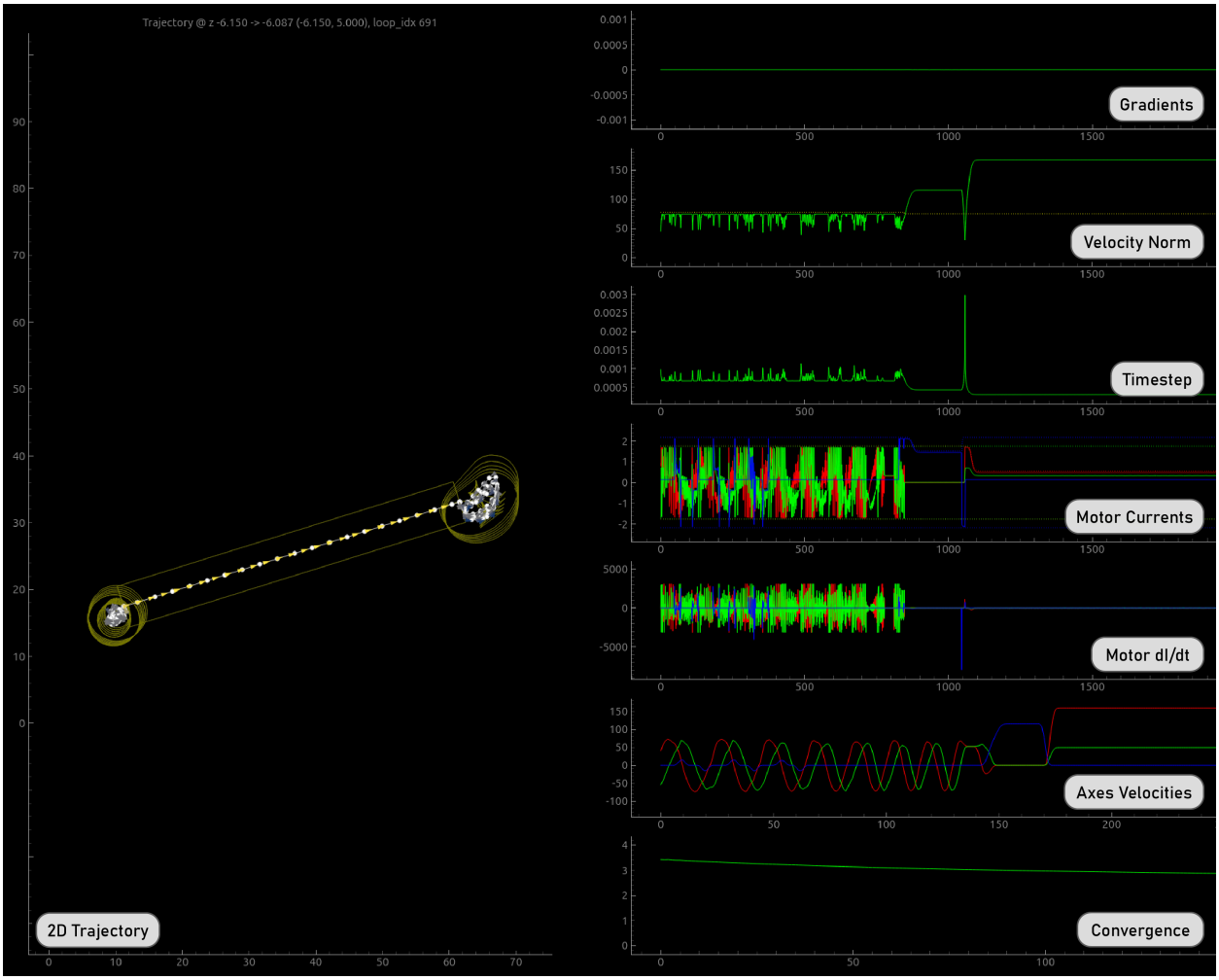


Figure 4.13: A frame from the planner's output visualizer, which can run in realtime along with the machine. This is from an "adaptive" CNC machining path. The trajectory is shown in 2D on the left side of the frame, and spatial-series data is on the right-hand side. $v(s)$ is second from the top, with corresponding $\Delta t(s)$ in the third plot. Fourth and fifth are $\vec{I}_{\text{req}}(s)$ (with their limits, dashed lines) and $\dot{\vec{I}}_{\text{req}}(s)$. The sixth plot contains *time*-series velocities for each axis, which are computed for transmission to hardware (4.6.1.5). Finally, the top and bottom plot are related to the planner itself: top shows the gradient of $v(s)$ with respect to the cost function (the direction that the planner thinks it should update velocities in the next iteration), and the bottom plot shows convergence over planner iterations.

axes that would be required to achieve the selected velocities $\vec{a}_{\text{req}}$ and time steps Δt for each spatial step.

This is a somewhat lossy abstraction because we are approximating cornering through a representation (line segments) that in reality has sharp corners. Properly described, maintaining any velocity through a sharp corner generates an infinite acceleration pulse on axes that are not aligned with the initial line segment — this is the same issue that junction deviation (used in trapezoid planners, 4.2.4) addresses.

So, this step is $f_a()$ from Equation (4.10):

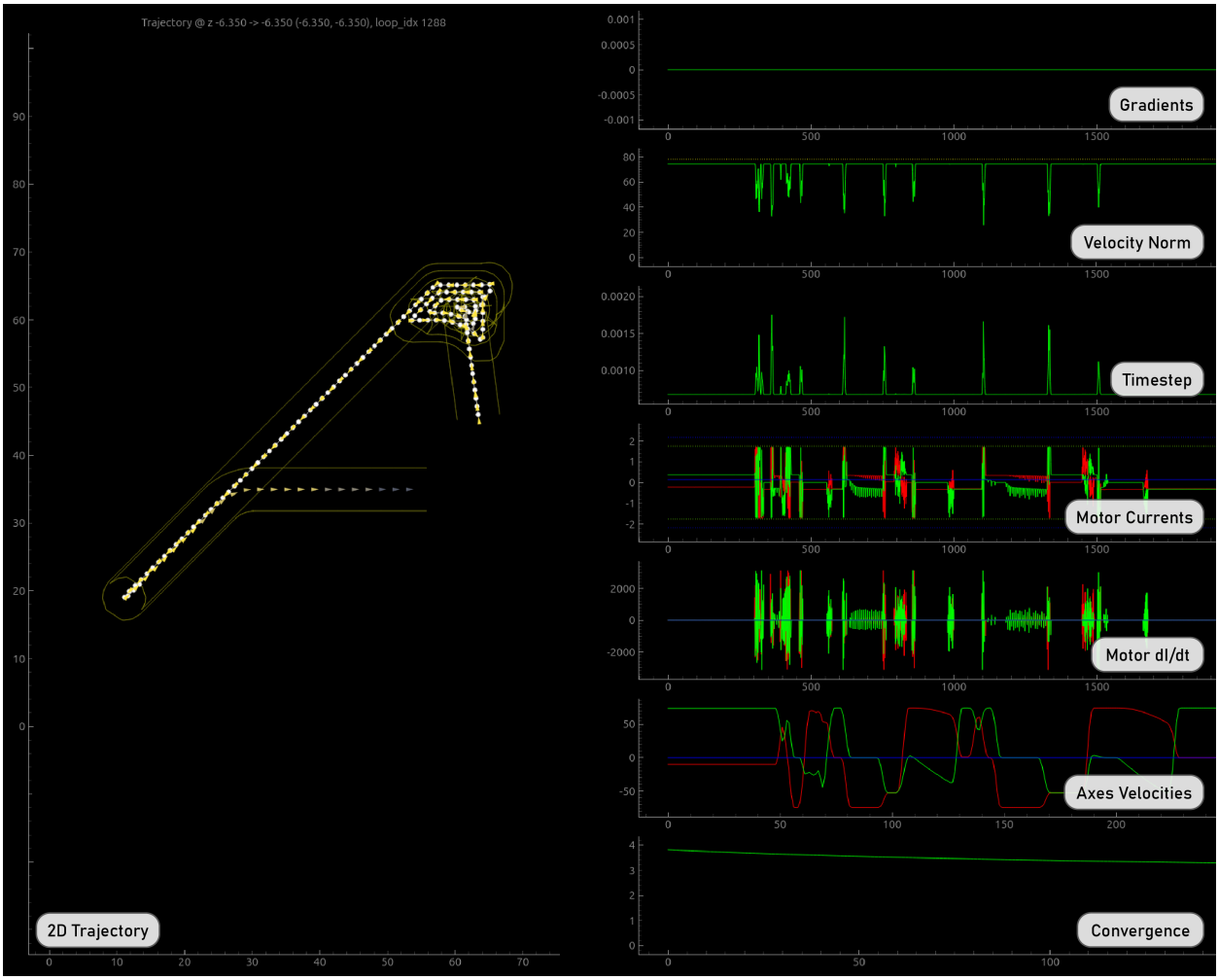


Figure 4.14: Another frame from the same planner run, now in a section of the path that features a 2D pocket machining path.

$$\Delta t, \vec{a}_{\text{req}} = f_a(v_{(s)}, \hat{v}_{(s)}, \Delta d) \tag{4.13}$$

There are two representations that we care about: velocities along the trajectory v and in cartesian space $\vec{v}, \vec{a}$. First we compute initial and final velocities in cartesian space, projecting $v(s)$ along unit vectors (from the trajectory's $\vec{x}(s)$) to make a $\vec{v}_i, \vec{v}_f$ for each linear segment:

$$\begin{aligned} \vec{v}_i &= v_{(s)} \cdot \hat{v}_{(s)} \\ \vec{v}_f &= v_{(s+1)} \cdot \hat{v}_{(s+1)} \end{aligned} \tag{4.14}$$

We have a fixed Δd step between out two points that covers the distance along the trajectory, but these two velocities are in cartesian space. To compute the time step we want to look at the component of motion that is aligned with the trajectory only because the actual arc traversed between these two velocities has a variable arc length depending on the (changing each planner iteration) velocities. The initial velocity at this point is just the selected $v(s)$ and the final velocity is the projection of $\vec{v}_f$ onto the unit vector at the start.

$$
\begin{aligned}
v_i &= v_{(s)} \\
v_f &= \text{proj}_{\vec{v}_f} \hat{v}_i
\end{aligned}
\tag{4.15}
$$

This has the important caveat that corners sharper than 90° have negative v_f along the trajectory, which complicates the next step (causing divisions-by-zero or negative time steps). I will note that I am working to develop an improvement to this step, see Sections 4.8.5.3 and 4.9.1. In the implementation that I used throughout the thesis, I instead assert that $v_f == v_{\min}$ for those acute corners, where $v_{\min}$ is a configurable planner parameter that ensures time steps do not become too wide.

Caveats aside, the time step for motion along the segment can be had now via the kinematic equations for line segments with fixed acceleration:

$$
\begin{aligned}
\Delta t &= \Delta d / \left(\frac{1}{2}(v_f + v_i) \right) \\
\Delta t &= \text{softclamp } (\Delta t, t_{\min}, t_{\max})
\end{aligned}
\tag{4.16}
$$

To ensure that time steps lie within acceptable boundaries, I clamp them after this computation using a smooth clamping function, discussed in Section 4.6.3. Once we have time steps, cartesian acceleration is just the discrete derivative of the two velocities.

$$
\vec{a}_{\text{req}} = \left(\vec{v}_f - \vec{v}_i \right) / \Delta t \tag{4.17}
$$

That's the end of this step. As I mentioned, it is a marginally lossy abstraction that is permissible because the line segments are so small. I work out an improvement to it in Section 4.9.1, using quadratic Béziers.

4.6.1.2 Calculating Required Actuator Currents

Given motion states along the trajectory, we need to broadcast these states back through the machine's kinematics and into the motor models. I explained these formulations alongside their associated models in 4.3.3.

To recap, we have two steps. The first uses inverse kinematics and friction models to sum current required in the actuator along two paths, one of which accounts for forces required to move the cartesian system around and the other accounts for forces required to move the actuators' own mass (rotor inertias) through friction present in the actuator space (rotor bearings and i.e. belts). This is described in full in 4.3.3.1 but also summarized in Equation (4.18) below.

$$
\begin{aligned}
\vec{I}_{\text{req}} \leftarrow \vec{k_t} \leftarrow \vec{\tau}_{\text{req,actu}} \leftarrow \text{Rotor } J, f_c \leftarrow \vec{v}_{\text{actu}}, \vec{a}_{\text{actu}} \leftarrow \text{IK} \leftarrow \vec{a}_{\text{cart}}, \vec{v}_{\text{cart}} \\
\leftarrow \vec{\tau}_{\text{req,cart}} \leftarrow \text{IK} \leftarrow \vec{F}_{\text{req,cart}} \leftarrow \text{Axes } M, f_c \leftarrow
\end{aligned}
\tag{4.18}
$$

The second step calculates actuator current *limits* based on their velocities, recall that our motor's ability to generate current decays at high speeds and that they have important saturations for maximum voltage and drive current. Again this is explained in more detail in 4.3.3.2 but summarized below:

$$\vec{I}_{\text{lim}^-}, \vec{I}_{\text{lim}^+}, \leftarrow \text{Electrical Models} \leftarrow \vec{v}_{\text{actu}} \leftarrow \text{IK} \leftarrow \vec{v}_{\text{cart}} \tag{4.19}$$

4.6.1.3 The Cost Function

The cost of the solution has two components. The first calculates exceedances of the actuator's current limits — remember that I_{lim} includes positive and negative limits (for maximum and minimum applied voltages), so exceedances are cases where these criteria are *not* met:

$$\begin{aligned} I_{\text{lim}^-} < I_{\text{req}} < I_{\text{lim}^+} \\ \dot{I}_{\text{lim}^-} < \dot{I}_{\text{req}} < \dot{I}_{\text{lim}^+} \end{aligned} \tag{4.20}$$

These costs are calculated using a smooth saturation (which also normalizes and scales them, see Section 4.6.3) so that gradients don't change suddenly as exceedances are approached.

I mentioned in the introduction that even when we develop optimization-based control, we are likely to want to retain some heuristics-based *knobs* for our systems. Recall also that $I, \dot{I}_{\text{lim}}$ have been calculated using motor models 4.3.1 and then scaled by $I, \dot{I}_{\text{prop}}$ - values encoded into the trajectory (which can vary by segment) to scale deployment of each actuator's total available power — this is where those knobs (which are in $0, 1.0$) are implemented here. In Section 5.8.2, I discuss how these scalars are set as part of the 3D printing workflow.

The second component encodes our preference for fast trajectories, it is much simpler: we sum all Δt and add that value to the cost. This provides a comprehendible basis for trade-offs that are expressed by cost function component normalization: when no other errors are present, the total cost represents the time that the machine will take to traverse the spatial horizon. In normalizing other error terms, we can then reason about *how much error in this regard would I permit in order to reduce the trajectory time by* $1s$.

4.6.1.4 Updating the Solution

The planner is structurally organized around the cost function, which returns the cost for the planner's current $v(s)$, given a target trajectory and initial states along that trajectory. The cost function is wrapped in JAX's `value_and_grad` function, which returns the cost value (to plot and debug) and the gradient. The gradient is passed to Optax's implementation of the *adam* solver [266], [285], which generates and applies updates for $v(s)$. The solver iterations are all wrapped into one jit-compiled[86] function, which is below (4.5).

There are two important parameters here, `params_solver_rate` (which correlates to the size of the gradient descent step, but is further tuned by the adam solver) and `params_solver_steps` (the number of iterations to run). I found that tuning these across different systems was required, see Section 4.8.3.

[86]JIT: Just In Time compilation, where sections of a program are compiled just before they run (or in JAX's case, after the first time they run).

Listing 4.5: The outer loop of the planner, where the cost function's gradients are used to generate updates for the solution.

```
optimizer = optax.adam(params_solver_rate)
cost_func_wrapped = jax.value_and_grad(cost_func)

@ jax.jit
def outer_loop(params, params_i, traj_bundle):
    # params = v(s),
    # params_i = v(s=0), the initial state from the prior solution
    # traj_bundle: x(s), v_max(s), i_prop(s), didt_prop(s), ...
    def step(carry, _):
        _params, opt_state = carry
        cost, grads = cost_func_wrapped(_params, params_i, traj_bundle)
        updates, opt_state = optimizer.update(grads, opt_state)
        _params = optax.apply_updates(_params, updates)
        return (_params, opt_state), (cost, grads)

    # jax.lax.scan causes the entire cycle of optimizer iterations
    # to be compiled together, i.e. this is just a jaxlike loop
    (params_f, opt_state_f), (costs_f, grads_f) = jax.lax.scan(
        step,
        (params, optimizer.init(params)),
        xs=None,
        length=params_solver_steps
    )

    return params_f, costs_f, grads_f
```

4.6.1.5 From Space to Time

In the final step, we need to deliver part of the solution to hardware, and roll the trajectory horizon into the future; the hardware operates on a fixed *time* interval, whereas the planner's internal states are all defined in fixed spatial steps. I.e. we need for each value $x(s)$ to generate an $x(t)$.

This is done with a set of interpolations. First we calculate the time at each point $t(s)$ using a cumulative sum of our $\Delta t(s)$, and then make a new time series t :.

$$t = 0 : \Delta t_{\text{spline}} : t_{\text{cycle}} \tag{4.21}$$

Where t_{cycle} (also appears in Equation (4.12)) is the interval within which we generate new planner solutions, and Δt_{spline} is the interval between basis spline points consumed by the MAXL system within which the planner is wrapped (see the MAXL Section 2.5, splines in particular 2.5.3 and 4.6.2 for the integration).

We can then find each parameter for s that corresponds to each t in the new series, and interpolate the trajectory positions $\vec{x}(s) \rightarrow \vec{x}(t)$ alongside all the other intermediate states that are selected to exit the planner.

Finally, we "roll" the trajectory along: segments that are within the transmitted time window are removed from the head of the trajectory, new points are added at the tail, and the planner begins work on the next cycle's solution.

4.6.2 Pipelining the Planner

OK. As I mentioned earlier, the planner is quite compute intensive and we need to rely on parallel computing in order to operate it *fast* enough that new chunks of the solution emerge in time to keep up with the real machine. The planner itself runs on the GPU via JAX, but the script that interfaces with it blocks during cycles when the GPU is producing a new solution. To parallelize that as well, it is connected to a MAXL / PIPES system through an *interface* class (that lives in the system's OSAP runtime) via an OS-level socket in Python.

In Figure 4.15, I show a PIPES system for the CNC machine system from 6 in a configuration where the planner is used to plan velocities.

The interface class also saves stores planner outputs directly for later evaluation. These are time synchronized with the rest of the system, which lets us align its internal states and outputs with data collected from sensors; this becomes important in Section 5.11.1 (and is used to analyze results from this chapter).

A note is that that while this planner *runs* in realtime, it does *not* receive realtime feedback from hardware. It produces new chunks of solution at 10 Hz only. The planner is a feed-forward trajectory optimization step, *not* what controls engineers call a Model Predictive Controller — although it does use models to predict optimal control outputs. The key difference is that

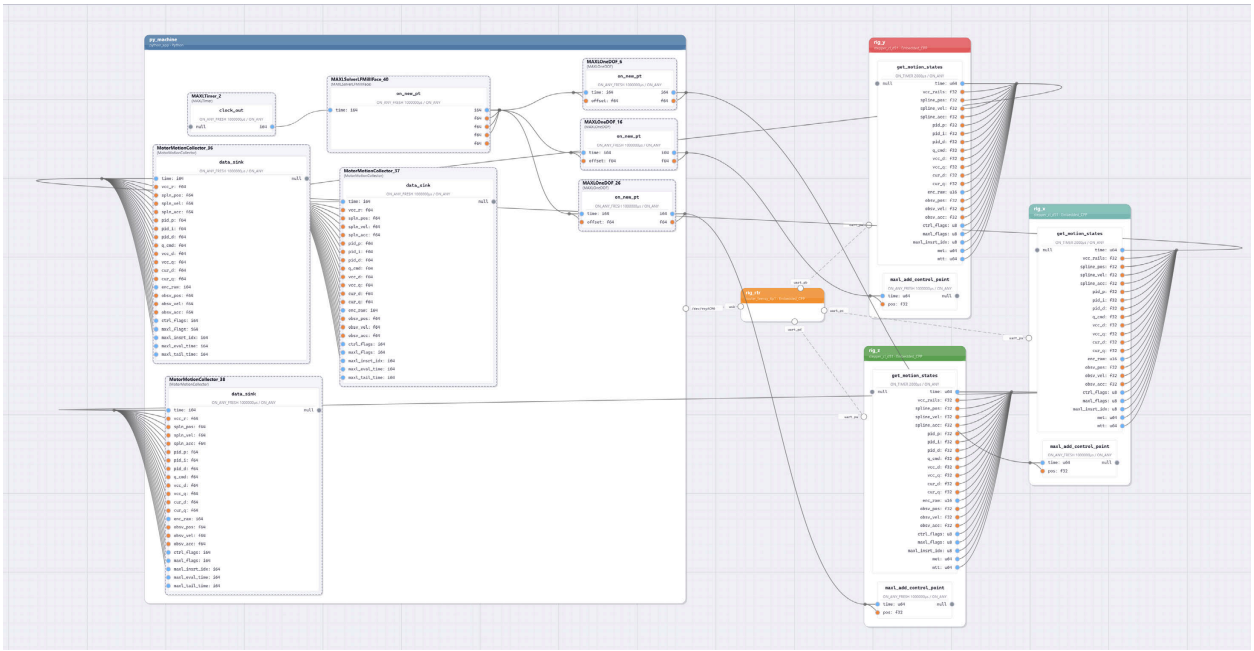


Figure 4.15: The CNC milling machine from Chapter 6, configured to use the planner for motion planning. Motors are rendered on the right side of the frame.

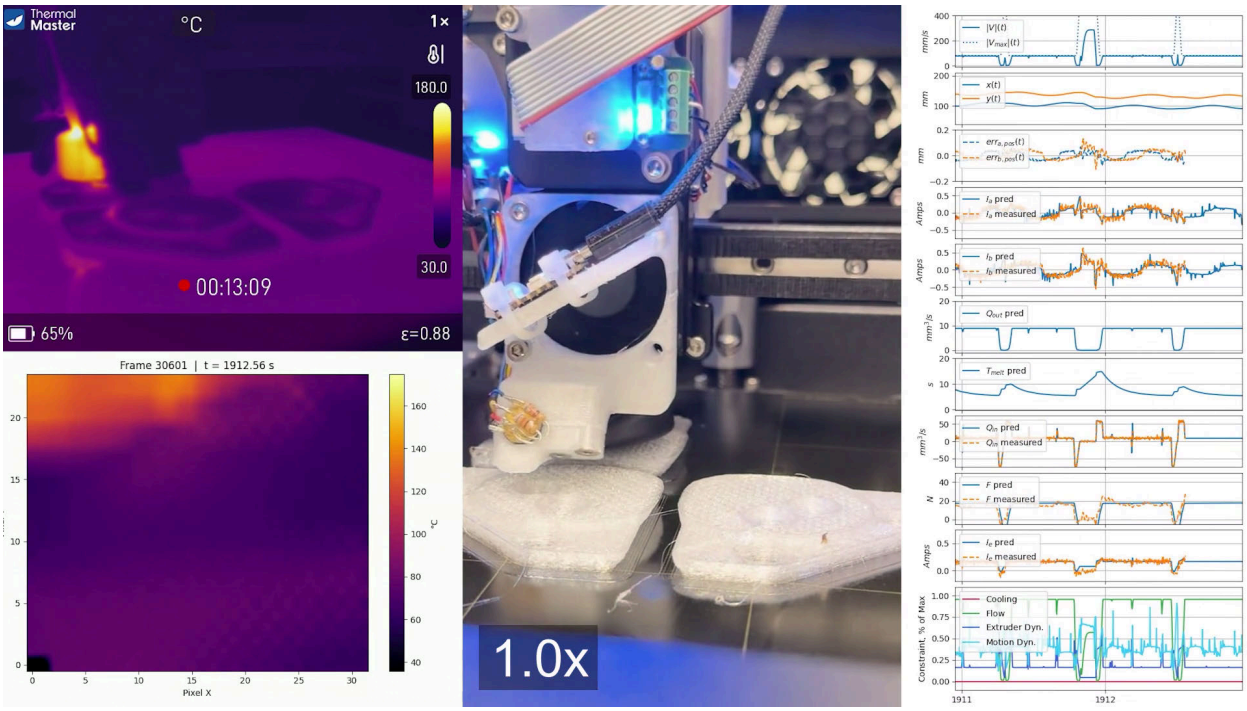


Figure 4.16: One frame of a video that overlays optical video (middle) with thermal video (top left: from an off-the-shelf thermal camera, bottom left: from the FrankenPrusa's integrated microbolometer 5.4.2) and time-series solver predictions (right, in blue) with measurements of the same values (orange), and a plot of system saturations (bottom right) that are calculated using solver data, see Section 5.11.1.

MPCs update their initial states with sensor measurements, this planner sends position targets to actuators and relies on those actuators' lower level PID-based position controllers (see notes on hierarchical control in 4.2.3) to track the targets. I can be caught calling it a *feedback based* controller because the models that it uses to pick optimal targets are based on feedback from the system itself, so — feedback but in much larger loop times.

I discuss the relative value of models for control (at various levels of the control stack) in some more detail in Section 7.2, and the perceived value of *direct closed loop* control in 7.5.

4.6.3 Soft Functions in the Planner

Ensuring that all the planner's computations are *smooth / soft* is important because it has to compute a gradient of the whole cost function from $v(s)$ all the way to the final output of the cost function. There are a lot of steps in between, and if gradients are sharp or vanishing in any of these steps the planner will likely fail, or will optimize poorly. This is not uncommon practice in machine learning and other optimization fields, but it was an important *trick* that I learned while writing the planner, so I include two of these functions here for the curious reader (I also showed the use of tanh for smoothness in Section 4.3.3.2 and 4.3.2.2) - leaving out this detail would result in a planner that doesn't optimize. I found it useful to define the size of the smoothing boundaries as a `blend_width` to make the configuration of the *amount* of smoothness more sensible

Listing 4.6: A *soft* function to clamp the input value `x` within the `lo, hi` range.

```
def soft_clamp(x, lo=0.01, hi=0.1, blend_width=0.005):
    """
    Smoothly clamp x to [lo, hi]. Linear in the interior,
    soft-saturating within `blend_width` of each boundary.
    """

    k = 6.0 / blend_width
    x = lo + jax.nn.softplus((x - lo) * k) / k
    x = hi - jax.nn.softplus((hi - x) * k) / k
    return x
```

— see Section 4.8.3 for some more notes on configuring these widths (a tuning step!) and on normalization of errors.

First is the `soft_clamp` that I use to assert that time steps are in a fixed range:

I use `soft_err_over_under` (below) to calculate exceedances of constraints in a manner where the error begins to increase *slowly* as the constraint is approached, rather than suddenly. I also include a normalizing scalar in this step.

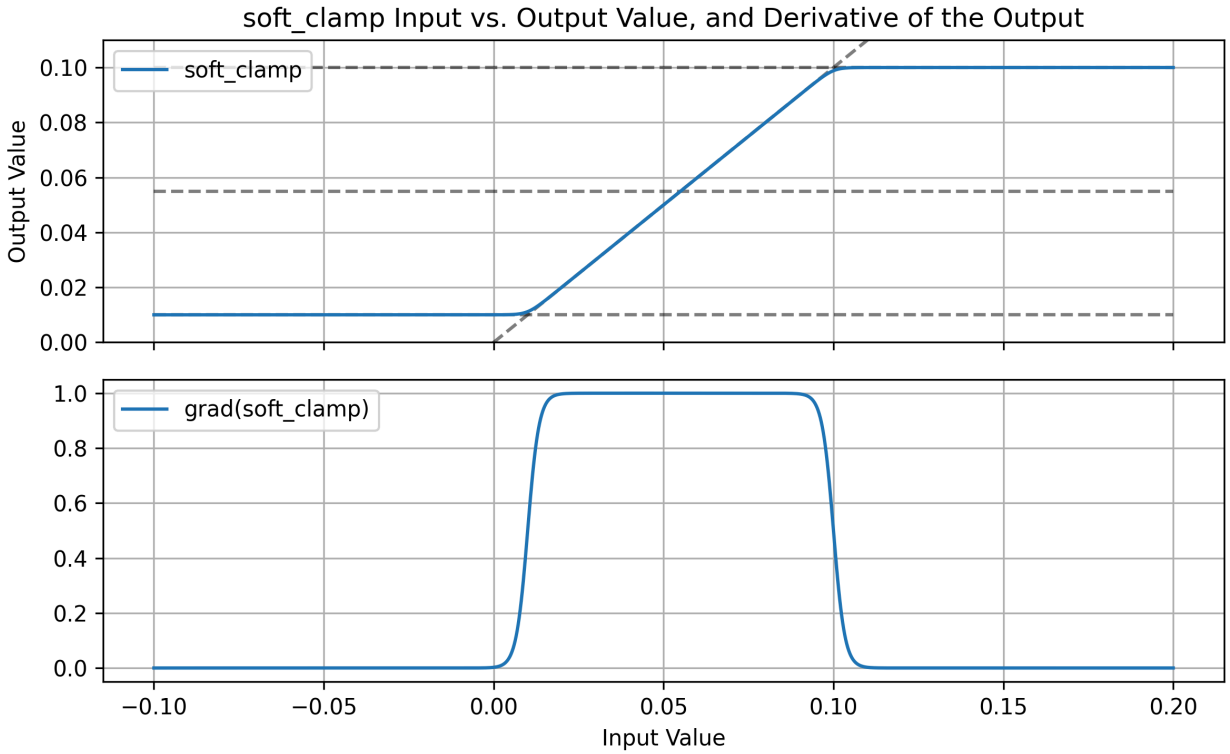


Figure 4.17: A plot of `soft_clamp` from Listing 4.6

Listing 4.7: A *soft* function for calculating exceedances of a value that has both minimum and maximum bounds, like the actuator currents in the planner.

```
def soft_err_over_under(val, min_val, max_val, blend_width=1.0, norm=1.0):
    """
    Error for value over/under exceedances of min/max,
    blend_width = size of softplus blend zone
    norm = expected max_val to norm errors such that 2x max = 1.0
    """

    hardness = 5.0 / blend_width
    scalar = 1.0 / norm

    umin = - (- min_val + val)
    omax = - max_val + val

    umin_soft = (jax.nn.softplus(umin * hardness) / hardness) * scalar
    omax_soft = (jax.nn.softplus(omax * hardness) / hardness) * scalar

    return umin_soft, omax_soft
```

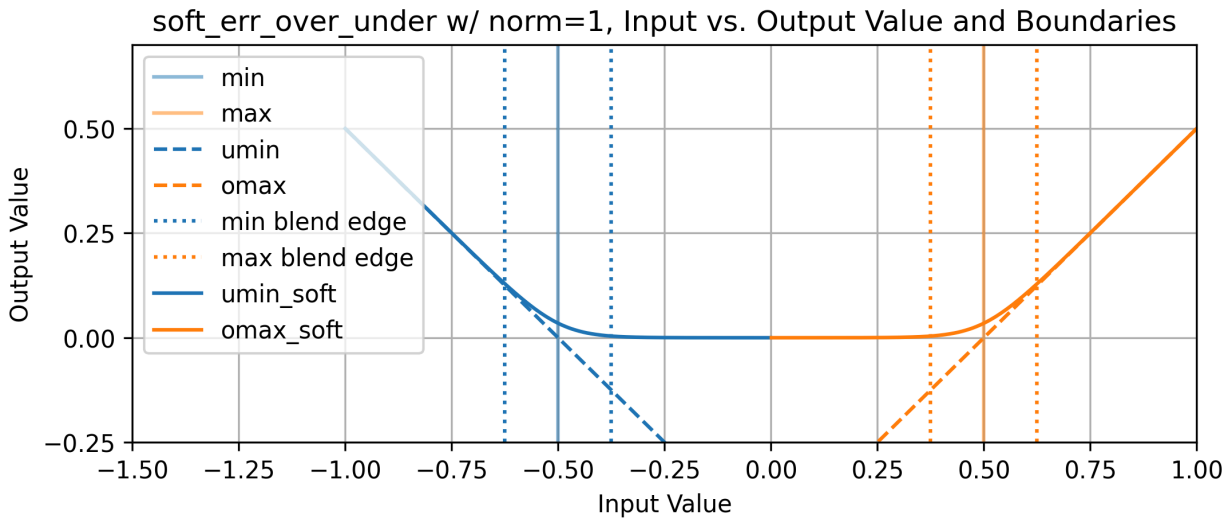


Figure 4.18: A plot of `soft_err_over_under` from Listing 4.7

4.7 Results

Here I share key results that are relevant to motion control on its own, but the methods that I develop here are extended in later chapters and form a basis for subsequent results.

4.7.1 Trapezoids vs. Model-Based Control

The planner allows us to use more of our motion systems' dynamic range than trapezoidal planners because they use models that describe motion limits in the full range of acceleration and velocity states. Trapezoidal planners used fixed values: a maximum acceleration and a maximum velocity (per axis of cartesian motion).

In Section 6.2.1, I show that we can use models to select these parameters for trapezoidal planners. An example of one such selection is below. The shaded area marks the space where the trapezoidal planner can operate. However, the real boundary (marked by $550N$, a force margin) encloses a larger area, especially in the left-hand side of the plot.

The system modelled in these images has a large damping offset. When we are accelerating *against* the current velocity, that damping can actually work in our favor; this dynamic is not exploited by trapezoidal planners. We can see this clearly if we look at motion states traversed during a test part by a trapezoid planner using these parameters.

The boundary is still violated somewhat by the trapezoidal planner. This is an artefact from the *junction deviation* step, which causes the machine to execute accelerations that are actually somewhat larger than the specified maximum. See 4.2.4 for more detail.

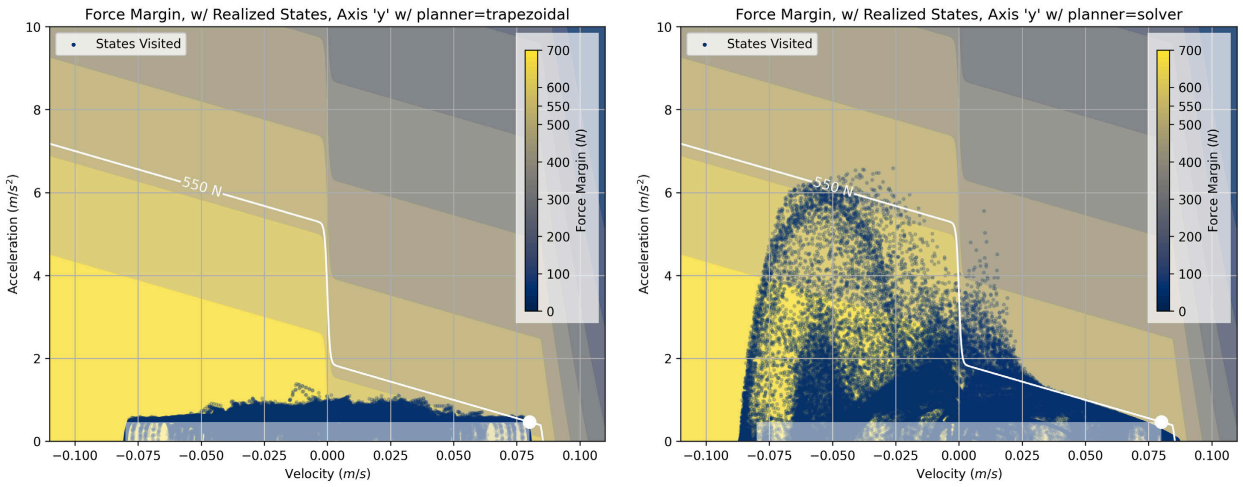


Figure 4.19: At left: states visited by a trapezoidal planner when it was configured to operate within the shaded rectangle (at a fixed maximum acceleration and velocity). At right: states visited by the planner as it optimizes velocities for the same path, where the force margin is the model-defined limit.

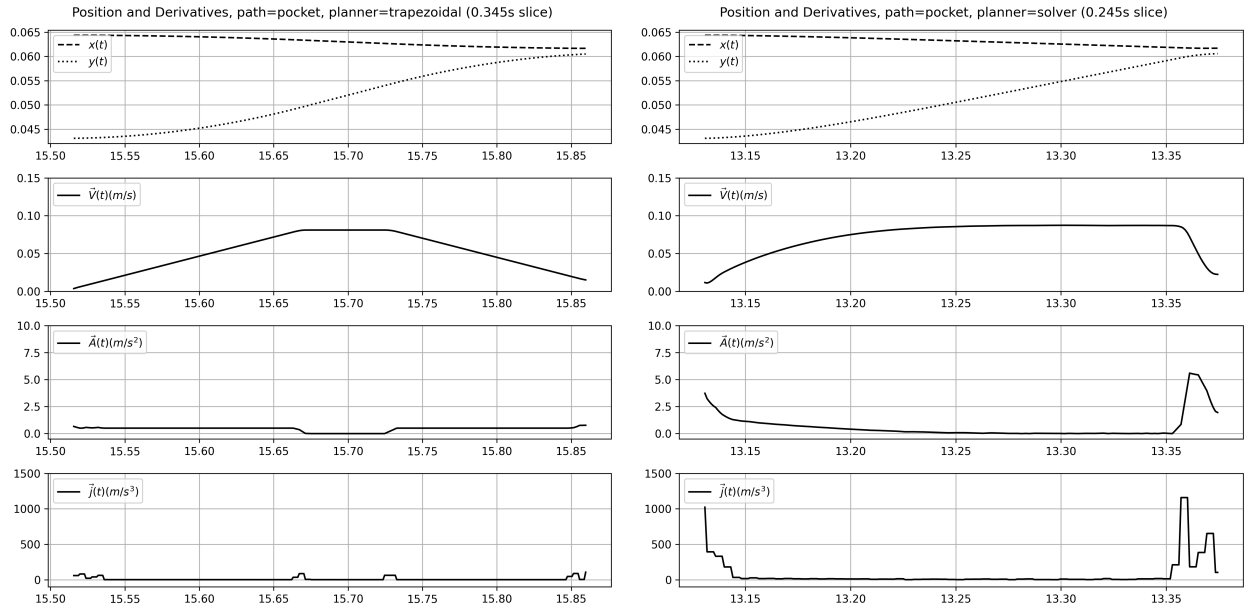


Figure 4.20: Position, velocity, acceleration, and jerk time-series data from a trapezoid planner (left) and ours (right).

If we run the same target trajectory with the planner, we can see that more states are used — again, especially in the left-hand side of the plot. There is some lossiness here, too — near the origin we can see a cloud of points that exceed the specified force margin. I believe that this is an artefact of the planner's own lossy cornering step (4.6.1.1). I discuss how a more direct use of splines in the planner could reduce these errors in Section 4.8.5.3. However, this shows that model-based motion planners have potential to greatly improve machine performance.

The difference between the planner and trapezoid planners is evident also when we look at the velocity-time plots that they generate, a comparison of which is below.

We can see that the planner exploits the system's damping dynamics: the front half of the speed curve (where it is working against friction) is much shallower than the tail, where it is working with friction. These plots are taken over the same trajectory segment: the planner also traverses it in less time ($0.245s$ *vs* $0.345s$).

We can also see that the planner uses more acceleration more regularly, and that acceleration is variable with respect to the system speed: a result of damping and motor dynamics together.

In CNC machining, maintaining feed rates such that they match user's cut parameters is of critical importance — when machine speeds decrease, chip sizes changes as well. When chips become too small, the cutting tool begins to *rub* instead of cut, which can cause chip welding. I discuss this in Section 6.1.1.

Because I developed a method to pick parameters for a trapezoidal planner using motion models (in Section 6.4.2), I was able to compare the model-based planner's performance against a trapezoidal planner where we know that the trapezoidal planners' motion parameters are not simply under-tuned. For this comparison, I use the trapezoid planner that I developed for MAXL 2.5.4.4.

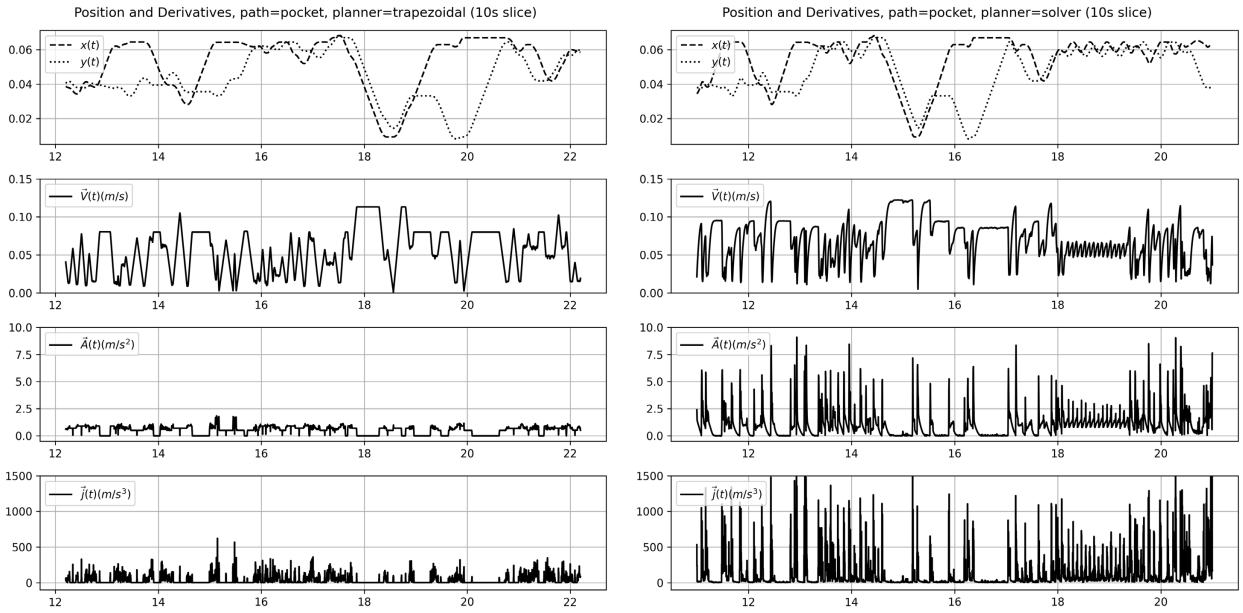


Figure 4.21: Position, velocity, acceleration, and jerk time-series data from a trapezoid solver (left) and our planner (right).

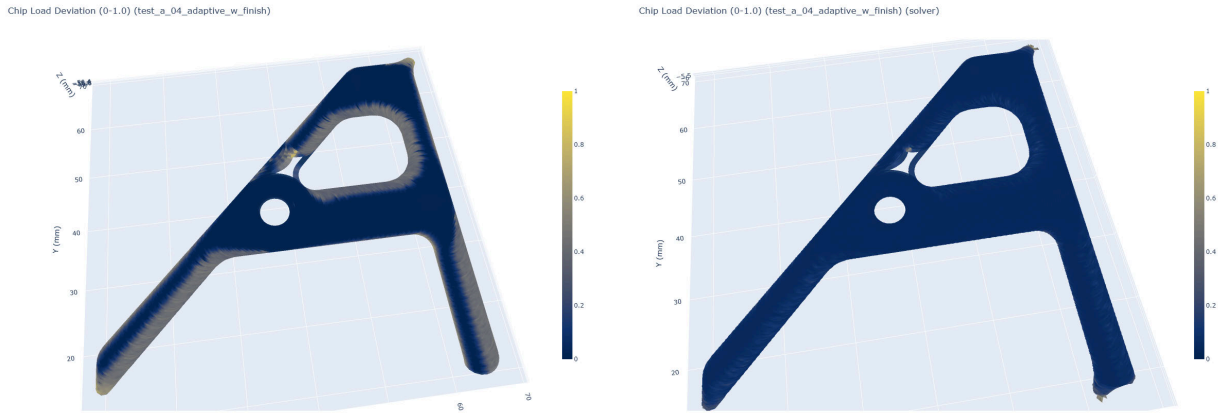


Figure 4.22: 3D heat map of deviations from target velocity from the same target trajectory, using a trapezoid planner (left) and the planner from this section (right).

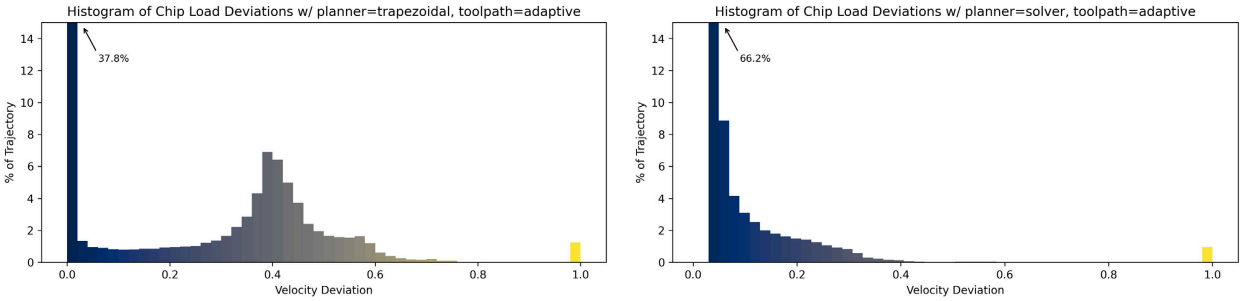


Figure 4.23: Distribution of deviations from target velocity from the same target trajectory, using a trapezoid planner (left) and the planner from this section (right).

MAXL / PIPES also lets us swap between the two planners without making any other changes to the system.

We can see that velocity deviations are greatly reduced when the planner is used. This is observable also in histograms that show the distribution of deviations over the course of the entire trajectory.

4.7.2 Model-Fit Quality

In Section 5.10.1, results from each part include an error metric on motion model fits. This metric measures the average deviation of the planner's motor current estimates from measured motor states. Those are calculated from time-series like those shown below, which renders a twelve-second slice of motion data from one of the prints. In it, we can see predicted motor currents (solid lines) vs. measured currents (dotted lines with some transparency).

On average, the printer models deviated from measurements by $0.175A$, which is considerable. These errors emerge partially due to noise emitted by the motor's PID controllers, which are tuned to be quite stiff. However, it is clear that the model fit is not excellent. The fits are initially performed on chirp data (per 4.5.2), but real-world machine control spans more states than those generated in the chirp test.

I also modified the printer between the chirp tests and operation, tightening its belts. This increases damping, which I forgot to account for by running new tests. However, noticing this leads us well into the next section here.

4.7.3 Continuous Improvement of Models

Because our hardware is also our testing equipment, we can *re-fit* motion models on data that we collect during regular operation, e.g., the 3D plot in Figure 4.24 shows current estimate errors in 3D from the FFF printer. In Section 4.5 I explained that friction values can be fit against any data where we know the systems' acceleration, velocity and actuator current measurements. These are all available in the datasets that these machines generate, and the models used in the planner to predict actuator currents are the same that we fit against. Because these are not time-varying physics (at this order), we can apply exactly the same process as described in Section 4.5.2 with new data to update models.

In Figure 4.25, I show motion states from a slice of time-series data generated on the FrankenPrusa (Section 5.4.2): solid lines are values collected from the model-based velocity planner in this section, dashed lines show current measurements sourced from the machine's motors as it followed this trajectory, and traces in opposing color show current estimates that would be made on the same data with *updated* friction parameters based on the new data. The update does not entirely capture the data, but does move estimates closer to measurements. This is perhaps because the motor's internal controllers add noise that the models don't capture. Here we are also *only* updating friction parameters, whereas other dynamics may have changed slightly or e.g. rotor inertia estimates may be off. In Section 8.2 I discuss a key limit to this work, which is that we cannot fit *through* multiple

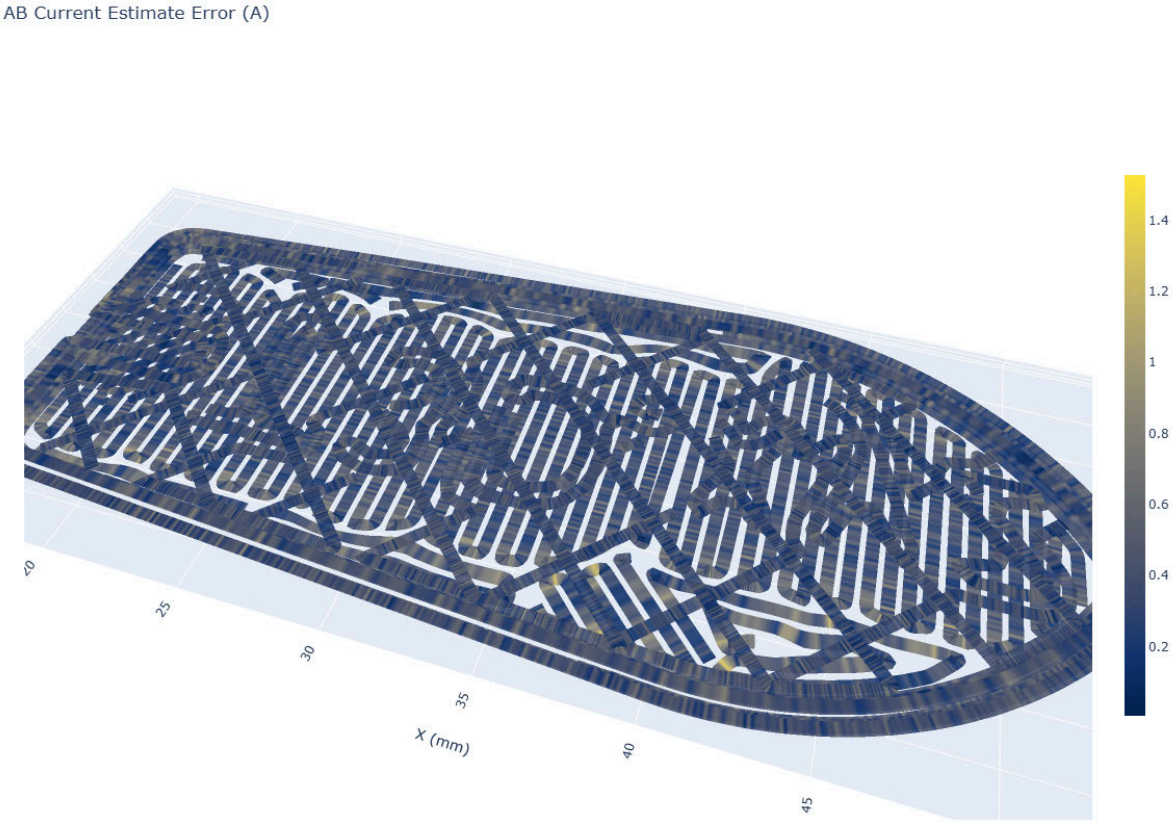


Figure 4.24: 3D trajectory data overlaid with current estimate errors, the difference between the planner's estimated actuator currents (for A and B motors on the 3D printer from Chapter 5) and measurements.

models, for example learning updated motor k_t values or estimating changes to machine mass or kinematic parameters using end-to-end datasets.

We also have the problem that real-world data tends to over represent certain states: machines spend most of their time at target feedrates rather than in highly dynamical states. To alleviate this to some degree, I weight data according to velocities at each data point so that varying velocities are equally represented in the updated fit. Really the distribution of states has many more dimensions of this, so more advanced weighting methods should be applied. This is especially true for continuous updates to process models, which I discuss as a possibility in Section 5.12.2.

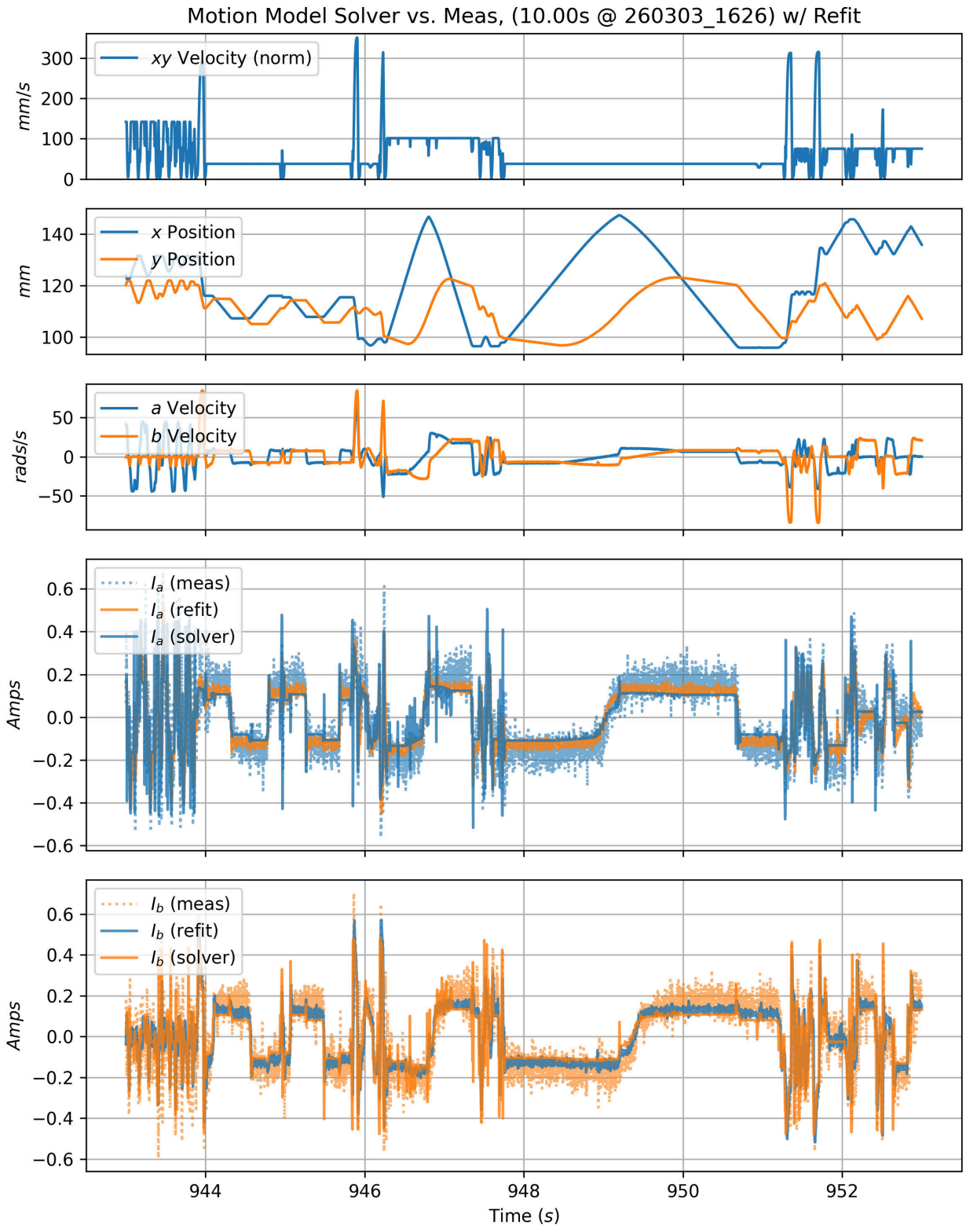


Figure 4.25: Time series comparison the system's motion measurements (dotted lines) with the planner's predictions (solid lines), for a subset of a print job. Lines of opposite color are including improved current estimates, calculated using model updates from a set of refit parameters.

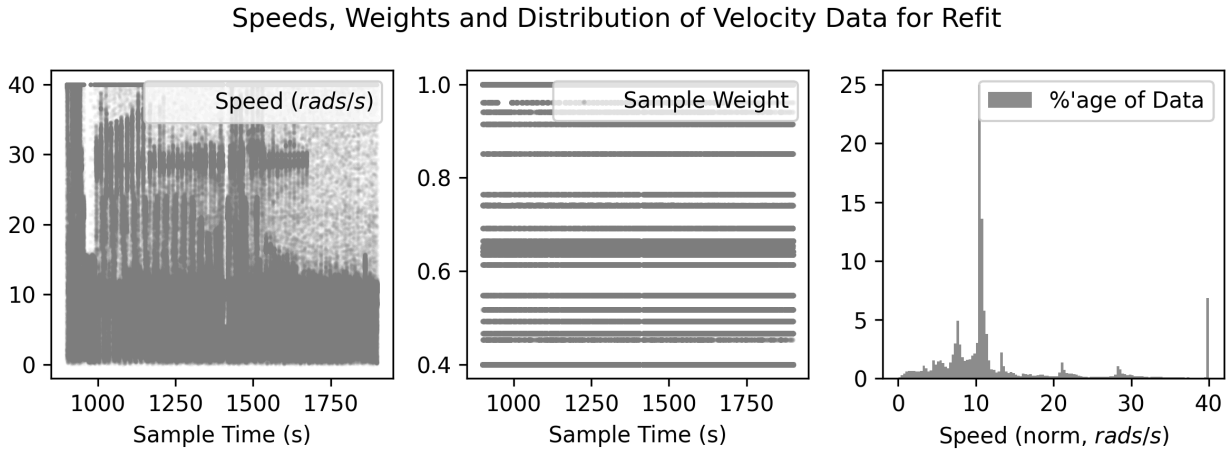


Figure 4.26: Speed-dependent weights for refitting: sample speeds (left) are weighted (middle) according to their distributions in a histogram of speeds (right) that defines the distribution.

Parameter	Units	Value Before Refit	Value After Refit
X Mass	kg	0.523	n/a
y Mass	kg	0.880	n/a
X Damping Offset	N	1.147	0.647
Y Damping Offset	N	0.053	0.079
X Damping	$N/m/s$	0.398	0.199
Y Damping	$N/m/s$	0.117	0.073
A Damping Offset	Nm	0.0144	0.0206
B Damping Offset	Nm	0.0143	0.0213
A Damping	$Nm/rads/s$	0.000370	0.000340
B Damping	$Nm/rads/s$	0.000370	0.000360

Table 4.5: Motion model parameters before and after refitting on new data from the FrankenPrusa

4.8 Discussion

This chapter answered one of the primary questions in the thesis, which was to learn how we can describe key machine control steps as using explicitly defined and numerically solved constrained optimization. To do so I developed a new type of velocity planner by combining relatively simple models for motion and actuators with a generic but powerful type of optimization framework. In contrast to state-of-the-art velocity planners it optimizes against models directly rather than being configured with simpler parameters for maximum jerk, acceleration, and velocities. This connects machine physics more concretely to machine control and allows us to e.g. update controllers by updating the models that represent the machine.

Many of the key results in later chapters are made possible by the work here, but I did show concretely that this planner out-performs an equivalent trapezoid planner; it can do so because it can leverage more of our machine's underlying dynamic range as expressed by motion models. Whereas I initially developed these models and model fitting methods for the purpose of integrating them with the velocity planner, I was surprised to find that their value in other aspects of machine design and control is perhaps more immediately applicable. For example motor and motion models are also an excellent machine design tool, and can be used alongside simpler velocity planners to more precisely pick their motion parameters (Section 6.4.2). They can also be combined with motor readings to make measurements like cutting force measurements (Section 6.4.4).

I was also surprised to find the limits of these models: I did not include machine stiffness here but vibrations that emerge when velocity planners excite machine structures are a key constraint to overall machine velocity. In stiff machines this is less of an issue but for the 3D printers that I develop in Chapter 5 it is a primary limit. I discuss those in Section 4.8.5 below and also how the application of heuristics on top of models was a critical step to ameliorate those issues. I also discuss here some more results that compare trapezoid-based solving to the planner that I developed here, some thoughts on the planner's formulation, and on motor models and machine component heterogeneity.

4.8.1 Machining with Trapezoidal vs. Model-Based Velocity Planning

I discussed a one-to-one comparison of a trapezoid planner to the optimization-based planner in Section 4.7.1. I was able to actually run these jobs on hardware. In these comparison runs, I saw two things of note.

First, it becomes clear that this planner is faster overall after the first few cuts. This is audible as well, there is a subtle difference to the sound that the machine makes as it enters a cut: it is more consistent (perhaps indicating uniform process parameters through the cut), whereas there is an audible *ramp* in tone in the trapezoid planner.

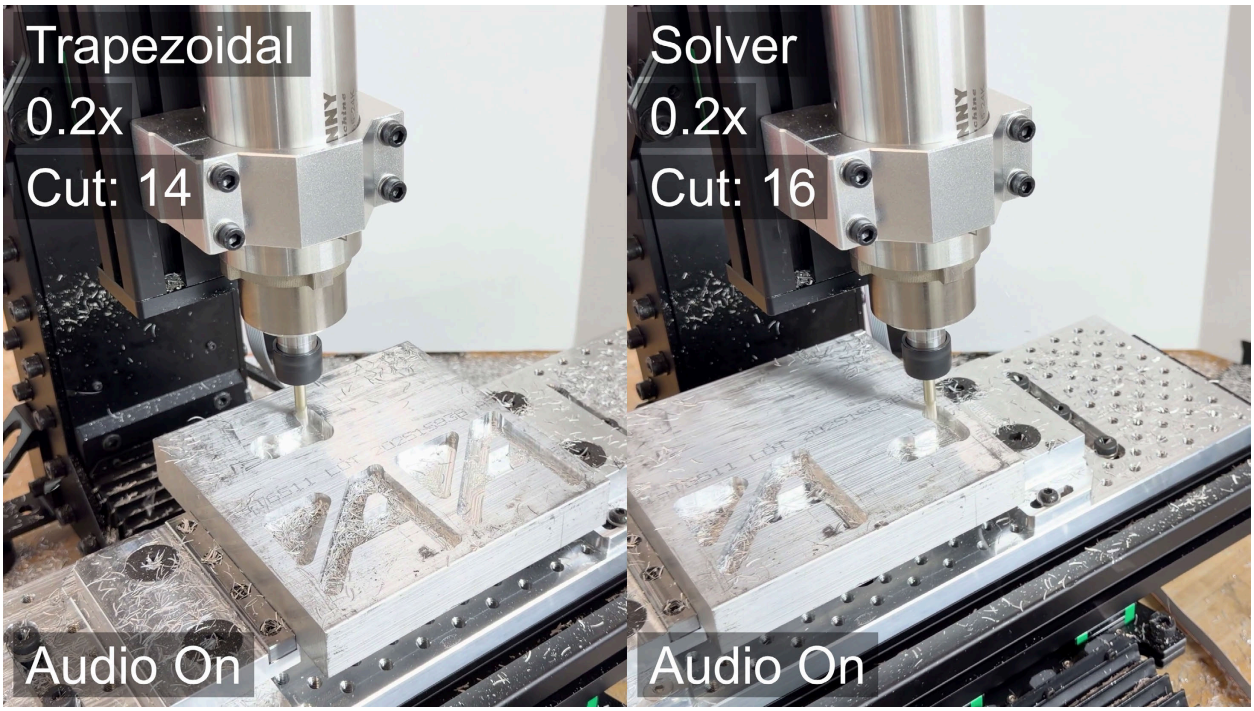


Figure 4.27: Comparing the trapezoid-based velocity planner (left) to the optimization-based planner (right).

The premise is that maintaining chip load reduces the likelihood of *chip welding,* and (while this doesn't constitute a rigorous evaluation), I could not get the trapezoidal planner to finish the cut without welding the endmill.

This shows how variable machining can be: the toolpath is the same as I had run a number of times previously, when I ran the experiments in Section 6.4.3. I tried the trapezoidal controller a total of three times, each with a fresh endmill. The updated controller run worked on the first attempt, and I tested it between one failed trapezoidal path and another. My guess is that in *this* case there was less time between jobs. In my prior runs of this part, I took a few minutes between each cut to adjust parameters in the toolpath. This time I was simply running paths one after another. The stock was warm to the touch during these experiments — perhaps the chip welding was more likely because of this increased temperature, however slight. This would also indicate that the toolpath parameters that I found in 6.4.3 were quite close to the limits.

4.8.2 Solving in *Phase Space*

Phase Space is a term I first saw while reading one of Ben Katz' blog posts on trajectory optimization [286], [287]. It sounds like something to do with the frequency domain but in generalized controls terminology a phase space is just "the set of all possible physical states in the system" [288], so this makes sense for our planner: for any position, which velocity? I want to explain why I think this is an important part of the formulation of the planner, and what it taught me about planners



Figure 4.28: Chip welding on the endmill used in this experiment.

like it, because I think it's an important component of why the planner works well, and it was somewhat counterintuitive to me at first.

The background section on MPC (4.2.7) shows the canonical state space representation in Equation (4.2). That is common in robotics because it generically describes how systems evolve in time (dynamics!) and cleanly separates a system's own evolution (the A matrix, subject to $x(t)$ system states) from changes that are driven by our controllers (in the B matrix, subject to $u(t)$ control inputs). This is perhaps the best way to *intuit* dynamical systems as well: if we are moving at some velocity (a state, x), and then apply some torque (a control knob, in u), we will have a new acceleration (goes in B), and also an acceleration due to damping on the existing velocity (goes in A). To integrate, we sum those changes to acceleration, update our new velocity, and we can get our next state.

In phase space *position* is the indexing variable, so we have variable time steps between each state, which can be a bit awkward for numerical integrators. However, it has advantage that the shape of the cost function doesn't change when we update our solutions. For example, the planner's main task is turning corners quickly: it needs to maximize speeds in between corners, but decelerate soon enough before the corner that it doesn't break things. If we increase velocities in the lead-up to the corner, the corner's *position in time* changes (i.e., it shows up faster), but its position *in space* stays the same. Thinking about the planner's perspective, this means that it needs to decide *where to begin slowing down* for the corner (an optimal point that doesn't move as its velocity selections change), rather than *when to slow down* (which does change).

If we formulated the planner such that the decision variables for the output trajectory were indexed in time, the shape of the problem that it is solving would change constantly as it updates its solution to that problem. The driving analogy works well here: imagine heading towards a corner

going very quickly and then, as you apply some braking, the corner is suddenly much farther away. This would make the braking point a very difficult optimum to find, and is effectively what happens when we use time indexing.

MPCs that use the *direct shooting* formulation are set up in this way (fixed time steps), and can work because their solvers are mathematically more precise than the one that I used here: they use *more* derivatives of the system (and tighter definitions for which problems are solvable and which are not) to see *through* time, i.e. to connect how decision variables in the past affect outcomes in the future. Since a design goal in this step was to be able to more easily articulate problems (eschewing this mathematical precision in favor of simpler and more flexible systems descriptions), formulating the problem in a manner where the position (in the system representation) of the relevant decision variables doesn't change over iterations was important.

The second reason that the phase space formulation works well in this planner is that *it is what allows us to parallelize the planner;* this is the same insight as in the *direct transcription* MPC formulation. Direct shooting formulations need to compute derivatives of their system through time through very many steps of the chain rule, that's a similar idea as the one above: they need to mathematically describe how control inputs in the beginning of the trajectory will affect changes in the cost function near the end of the trajectory. They also need to compute their simulations serially: take initial states, and for-each selected $u(t)$ iterate the solution one step in time, updating the next time step's states, and so on. Trapezoidal planners have the same serial compute requirement in their forwards-and-backwards planning step.

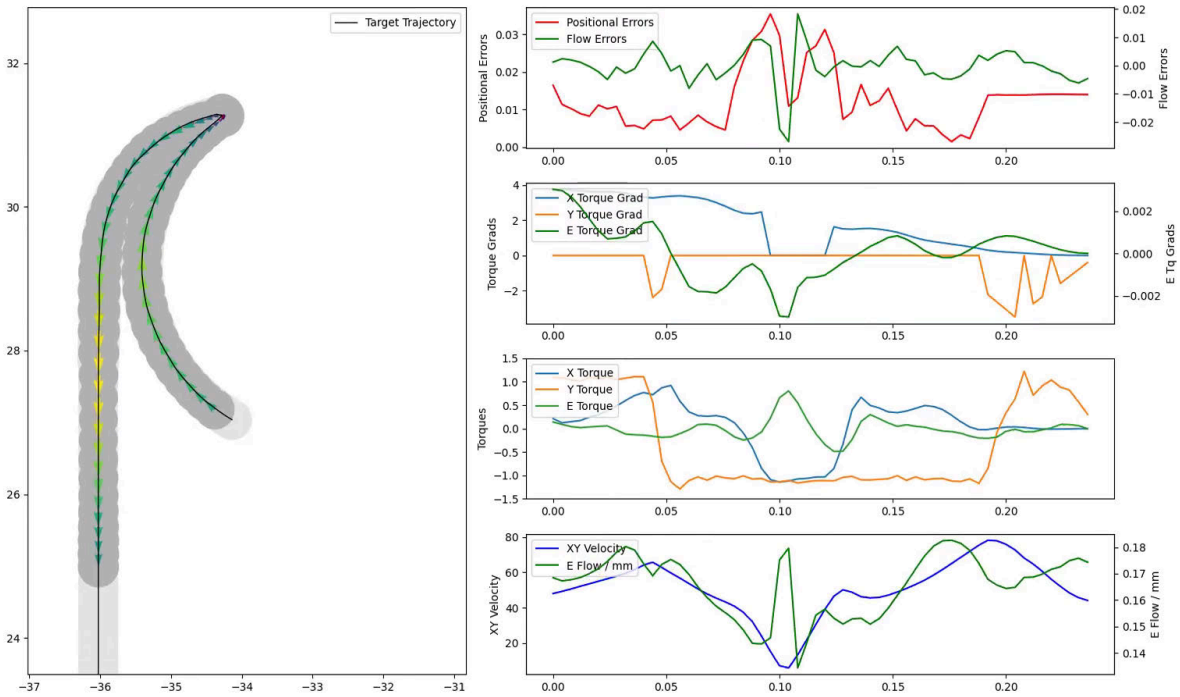


Figure 4.29: An earlier version of the planner, shown here, used the direct-shooting MPC formulation.

While this works well for motion, it introduces some challenges for other dynamic systems like the 3D printing extruder as I discuss in Section 5.11.7.4.

4.8.3 About Performance and Hyperparameters

In the introduction to this chapter (Section 4.1.2.1) I mentioned that optimization-based planning runs the risk of trading one set of parameters for another set in the planner's configuration. This turned out to be the case, as the planner has a few hyperparameters that sometimes need to be tuned in order to produce consistent results. There are three especially sensitive hyperparameters: the length of the trajectory that it solves *over,* the size of each segment that it will pick a velocity for, and its update rate (which defines how large of a step it will take in gradient descent). Together these set the size of the problem and the sensitivity that the planner will have to the system's convexity. For example a long trajectory with very fine spacing requires more computation to plan for, and a high learning rate can cause the planner to become unstable when machine models are sensitive to small changes.

For motion control alone, setting these parameters up was never an issue because motion and actuator models on their own are quite simple and are highly convex. However, once I added the 3D printer's flow system (see Section 5.9), the planner both required more compute cycles to generate optimal solutions and also more sensitive to subtle changes in the flow system — especially the time-dependence that is present there. As I added new physics to the planner in that work, t was common that I had to re-tune planner parameters (optimization rate, steps, size of the problem) to get it solving properly, and in time. For example with simpler systems I ran the planner for only 50 iterations per cycle at a relatively large update rate. Once the flow system was added in its entirety, I was running the planner for about 400 iterations per cycle at an update rate that was $1/8^{th}$ the size.

Getting the planner to work well also required that components of the cost function were well normalized. This is common practice in machine learning and optimization. I will not expand on it at length, but normalizing each new set of cost function components was another sensitive step that indicates that the planner does not come without its own set of internal configurations. However, I found that the best way to articulate these normalizations was to connect them to real physical values; constraint exceedances in the cost function are articulated in a way where they are traded against time (in seconds). This way we can say e.g. "for every amp of current used in excess of defined limits, a solution must be at least fifty seconds faster than it would have been if those limits were not exceeded." This connects the normalization back to the physical models that we are referencing.

4.8.4 Notes on Motor Models and Motor Heterogeneity

I wanted to briefly show that even two motors of the same make and model can have varying underlying parameters; no two parts are truly the same. For example Table 4.6 shows model-

fit parameters for two individual motors that are both from the same manufacturer, same part number, etc. I found a 7% difference in their k_t values, which is the key parameter that defines how much torque each produces for each Amp of current.

This is likely due to the fact that how current couples into a motor's magnetic system is a strong function of the quality of its windings, and winding stator coils is a somewhat imprecise process. This has real results for machine builders as well, who must normally select one set of motion control parameters for all the machines they will ship even though each has various underlying differences that are subtle but obviously present; model-based control deployed where models are fit against data generated on the *specific* hardware that will be controlled has clear benefits in light of these considerations.

Sourcing models from hardware also provides the benefit that it better informs machine designers choices as they trade off on other fronts e.g. between torque and speed. Controllers also factor in the tradeoff; low-end torque limits are as much a function of motor physics as they are of driver limits and high-end speed limits are a strong function of the voltage that we can supply to those drivers (and the speed of the controller's commutation). So again we can see why developing methods that fit models through each of these components could be beneficial.

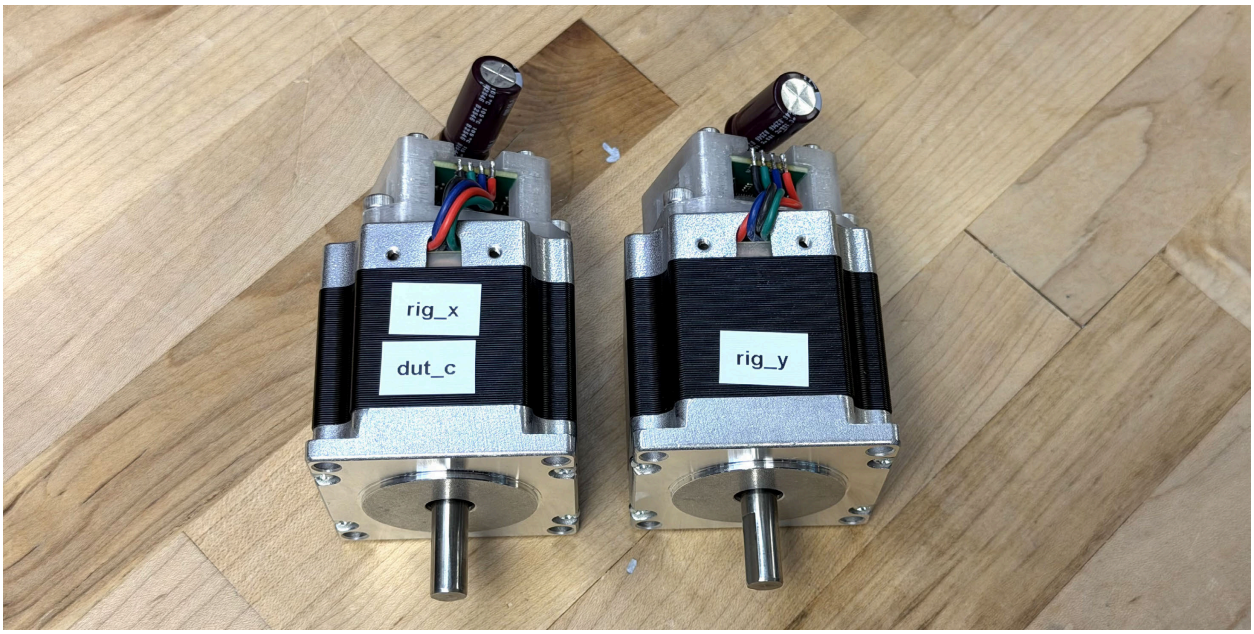


Figure 4.30: Same, but different. These two motors are the same manufacturer model (StepperOnline 23HS22-2804D), but have slightly varying `kt` values, as shown in Figure 4.31 and Table 4.6

Parameter for StepperOnline 23HS22-2804D Motor	`dut_c` Individual Motor	`rig_y` Individual Motor	% Difference
k_t	0.3234	0.3407	7.3
k_e	0.3312	0.3456	4.3
k_d	0.0009	0.0011	22.2

Table 4.6: Model parameters for two individual motors of the same manufacturer model.

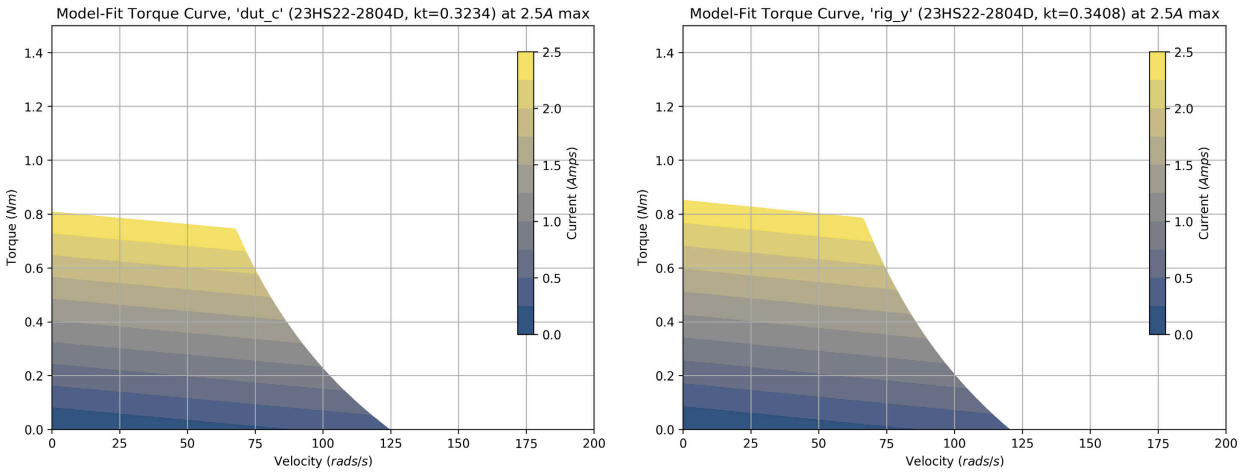


Figure 4.31: Motor `dut_c` (left) and `rig_y` (right), are the same manufacturer model, but have different model parameters, indicating that their internal construction is slightly different.

4.8.5 Key Limits

4.8.5.1 Stiffness and Vibration is Key

At the start of this project I had assumed actuators, friction and inertial physics would explain most of the limits to overall motion performance. In the past when I had designed machines, mostly I selected actuators based on this assumption and calculated machine stiffness for the purpose of modelling *deflections,* not vibrations. The coupling between vibration and motion control was not obvious to me, perhaps because the interaction between velocity planners, machine structures, and actuators and transmissions is much more difficult to model. Preliminary machine design is canonically a spreadsheet-based exercise [289] although obviously the practice is changing given new and more powerful modelling tools. At the design stage however it is difficult to tally all the aspects that will contribute to a machine's frequency response.

In this work I found that vibration and stiffness is a primary concern for velocity planning. In hindsight this is clear to other researchers, especially those who articulate this step entirely in terms of filter-based methods (see Section 4.2.5). Actuators, friction, and inertias are of course still critical because those define the basis for control — vibration is a response to their inputs.

4.8.5.2 Heuristics In Lieu of Models

I developed the application of additional heuristics for motion control (see Section 4.6.1) largely in response to this shortcoming; when the planner is allowed to use all the power available in e.g. the FrankenPrusa's actuators it's frame deflects tens of millimeters. To calm the planner down we can provide $I_{\text{lim}}, \dot{I}_{\text{lim}}$ values that scale the system's underlying limits in the cost function. Applying these heuristics makes a significant improvement but doesn't address the root of the problem. The application of these heuristics also make it difficult to tell what *really* limited machine performance in the plots that I produce in Section 5.11.1.

This points to another limit in the methods that I developed, which is that the planner does not model deviations from the toolpath that might arise due to these vibrations or due to steady-

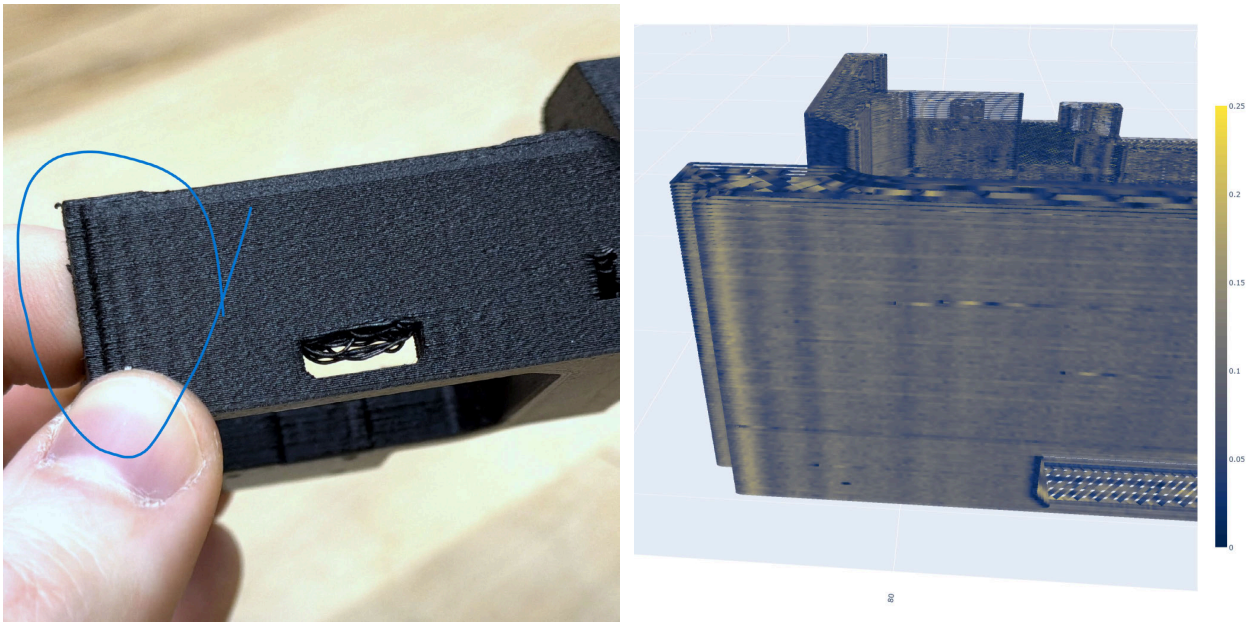


Figure 4.32: Our print of the Circuit Mount in PETG with Carbon Fill using a 0.4mm High Flow Nozzle. Left: the print, showing "ringing" errors (vertical bands). At right, deviations (yellow) from the planner's position target and the stepper motor's actual positions.

state structural loads. The application of any force to a machine will cause *some* deflection, and so the improvement over these methods would be articulate the speed-vs-precision trade-off against permissible geometric tolerances. I discuss this also in the 3D printing chapter, specifically in the section on *Pareto frontier printing,* 5.12.4.

That said, any attempt to model the *entirety* of a machine's physics will eventually come up short. There are also moments when we are bootstrapping models where the application of simple heuristics is just easier, as I discuss in Section 7.2.4.

4.8.5.3 Lossy Abstractions in the Planner

In several places in this thesis I have articulated how other machine controllers are rife with *lossy abstractions* where the models used to operate certain components of machine workflows poorly capture important behaviors (or fail to reveal them to higher level algorithms). However, I introduce three of these abstractions in the planner's formulation and in the way that the planner connects to hardware.

The first is in how cornering velocities are represented in the planner, which is described in Section 4.6.1.1. This step is central because it defines how machines move through curves and additionally calculates the time step for each spatial step; that is critical especially then the planner is connected to process physics later on. The loss emerges from the fact that the planner represents cornering using line segments, but the path traced by the machine is actually a curve. As I mentioned in Sections 4.2.4 and 4.2.5, any velocity carried through a sharp corner will in reality produce an infinite impulse in acceleration.

The second has to do with basis splines. The planner works by selecting velocities for each junction between a series of very small line segments, a spatial representation of the problem. But once solutions are generated, they are transmitted to motor controllers using basis splines. Although the lossiness of this handoff is minimal (see Section 7.4.1), it really *seems* as though we aught to be able to solve the spline outputs directly, rather than solving the trajectory in one representation and then transforming it into another. Because the splines are *also* a kind of filter, they would be an excellent basis for adding a geometric tolerance limit to the planner. At risk of stating the obvious, splines are also *actually* curves, not line segments that introduce the issues I just mentioned concerning cornering velocities, so using them directly in the planner could ameliorate that loss as well. Finally, they have other nice properties that emerge at a systems assembly level, which I will discuss at more length in Section 7.4. All of this said, representing the trajectory in phase space (with fixed spatial steps) was also extremely productive. In Section 4.9.1 I briefly discuss how we might resolve these conflicts.

The last concerns the way that the planner is connected to hardware through MAXL and PIPES. In those dataflow graphs, the machine's kinematic arrangement is essentially mirrored again in the graph configuration. The planner emits points in the machine's tool path space (i.e. XYZ axes) and those are then transformed again in the graph on their way to motors and other hardware. This is done even though the planner has its own internal representation of those kinematics, so misconfigurations can arise when the two representations do not match. Luckily both the planner and the MAXL graph are defined in software, so building a configuration step that aligns their representations would not be difficult whereas doing so across the GCode boundary is nigh impossible because there are no standard GCodes that let us query interpreters internal configurations.

4.8.5.4 Nonlinear Kinematics, End-to-End Fitting, and Models of Controllers

There are a few other clear limits to this system. I did not show how it might control systems with nonlinear kinematics even though doing so would have clear value because they are difficult to manage in other velocity planners. I also found that the bandwidths of lower level controllers is an important consideration in the velocity planning step, but did not include representations of those in the planner itself; I discuss these in the future work section here, 4.9.

4.9 Future Work

4.9.1 A Better Representation for Δt and $\vec{a}_{\text{req}}$

In Section 4.8.5.3 I explained how the planner's representation for cornering velocities is somewhat lossy and started to connect that issue to other abstractions in the planner. This largely revolves around the use of basis splines; Section 7.4 connects those also to systems integration more broadly. An overarching goal of the thesis is to use similar systems representations throughout control layers, so the particular representation that I use for cornering velocities has stuck out to me as

a failing in this regard. But it also *works,* mainly because it frames the planner's work in phase space, as I discussed in Section 4.8.2.

So, a key component of the future work in developing this type of planner should revolve around resolving this tension. It seems likely then that an improved planner would use splines in two steps: first with fixed spatial control points over which velocities could be planned, and then with a time scaling applied according to those velocities to transmit plans to actuators.

4.9.2 Exploiting Time-Varying Motor Dynamics

In Section 5.11.5, I show that the planner developed here can be extended to solve over time-varying dynamics. In that example the planner exploits thermal history in a 3D printer's melt flow system to briefly extrude at speeds beyond what steady-state limits alone would allow. There are similar dynamics at play in motion systems; actuator current is instantaneously limited by driver capability, but the motor's own current limit is thermodynamic: if we heat the stator's coils past the melting point of their insulating enamel, they short against one another and the motor fails.[87] It takes time for coils to heat up, so if we could model these thermodynamics we may be able to use much larger drive currents and torques than we would otherwise at particular instants, e.g., where our machines are turning tight corners. In fact motor drivers have these dynamics as well — MOSFETs have instantaneous current carrying capabilities that are much larger than their steady-state limits, which are also based on their maximum temperatures.

4.9.3 The Missing Models for Motion

Section 4.8.5 discusses the most important aspects of motion control that are missing from the current formulation here, but I want to expand on how I think we should develop these components in the future.

The most promising aspect of the planner is that it allows us to describe machine physics explicitly and then do control declaratively rather than via intermediate parameters and approximations of those physics. In the frequency domain this should be no different. State-of-the-art methods for vibration repression in machine control are formulated using filters that are designed to suppress excitations are particular frequencies (see Section 4.2.5). While this is often done using machine-generated data that plots which frequencies are most valuable to suppress, this amounts to another parameter-based configuration step. The underlying issue is that vibrations cause machines to deviate from target paths, so we should be able to *declare* geometric constraints to the planner and then provide it with models that describe how those deviations emerge. For example frequency response is phenomenological not physical, the underlying terms are stiffness and damping terms. This means that frequency response *tests* could be used not to design filters but to fit these values

[87] Motors also have *magnetic saturation* limits that reduce magnetic field strength even when very large currents are applied.

across a range of machine states, and those could be used to "show" the planner how much a machine will deflect according to candidate velocities that it selects.

Frequency response is history dependent; machines oscillate because energy that has been loaded into e.g. belts subsequently unloads at a later time (the loading is stiffness related and the unloading is damping related). I show in Section 5.9 that the planner can manage history-dependent physics, but spring oscillations have much higher bandwidths than the thermal systems that I integrate there. Because the planner is discretized, there will be a Nyquist limit under which solving vibrational dynamics in this way is prohibitively expensive, but it is worth experimenting with for less rigid machines; belt driven hardware is definitely the most relevant here but even some larger gantry-based milling machines may have oscillations that can be managed in this way.

Modelling vibration in this way may also make it easier to couple those physics into processes. For example high performance machining using belt-driven systems is an absurd proposition because the interaction between chatter and belts is especially chaotic, but perhaps this would become possible if cutting force loads were tightly integrated with models for belt stiffness?

4.9.3.1 Nonlinear Kinematics

I also mentioned nonlinear kinematics, which cause inertial dynamics to change alongside not just machine velocity but also machine *position.* Describing these physics is difficult in the state-of-the-art but machine designers often experiment with novel kinematics to improve performance. To simplify how these kinematics are articulated to planner, in Section 8.1 I propose developing a specification for a machine's 3D digital twin that could be derived from CAD models. I think that the planner could be extended to use this type of representation for varying inertial dynamics as well; it already tracks internal velocity and acceleration states for both motors and axes even where they are not directly coupled. Adding intermediate rigid bodies would require that kinematic functions like those in Section 4.3.2 also return positions and orientations for each of those bodies so that the forces required to move them around could be also reflected into actuator models.

4.9.3.2 Controller Models and Computational PIDs

I found that *other controllers* in the system should be modelled in the velocity planner because they have their own bandwidths. Motor physics alone do not describe how a motor will perform; to understand the whole picture we need also to understand things that normal hierarchical control considers external: encoder noise, phase lag from Kalman filters or other observers, and sample-and-hold delays introduced by discrete controllers.

Again, a benefit to the formulation here is that we can use the same computational representations of these systems across layers. It is already common for controls engineers to model these aspects of their hardware when they use optimization-based methods to write low level control gains. Here, modelling these finer grained details could have the extended benefits of connecting models of controller implementations to the planners that drive them.

Interdependence across layers is a theme in this thesis; try as we may to separate concerns, there are deep connections across layers that become hard to ignore at the edges of performance. In

Section 8.2 I discuss a question that this all raises, which is to see whether we can fit hierarchical models end-to-end.

Finally, the implied goal of Section 4.4 is that our controller should be deployable, somewhat automatically, on many unique stepper motors. Measurements of motor R, L and pole pair counts are possible to do using the controller itself, rather than (in the case of L), the somewhat expensive LCR probe that I use. In this case the measurements are simpler and can be done without decoupling the motor from the machine. More difficult is estimating also the rotor inertia, as error in its estimation can easily slip into the motor's k_t value (less k_t has a similar effect on acceleration as having larger J does); although it *should not* slip into the value for k_e and the two values are the same according to the motor model, so the difference can perhaps be distinguished form data. Overall on topics like this, I think that extending the modelling techniques used to quantify uncertainty about parameter fits is a clear next step for the work.

5 Model-Based Control of FFF 3D printers

5.1 Introduction

I discussed in Chapter 1 how machine control is fundamentally a *constrained optimization* problem. We want machines to work quickly and precisely, but their operation is constrained by machine and material physics. In the state-of-the-art we must intuit these constraints and then estimate which process and control parameters will prevent the machine from violating them; see the CAM primer 1.2, these parameters are exposed in 3D printing within softwares called *slicers* that turn 3D models into machine instructions (GCodes). The proposal in this thesis is that we can instead articulate constraints using mathematical models of machine operation and then use those models either directly (making the optimization explicit) or indirectly (to better inform our intuition). In addition, we can make models *using the machines themselves* rather than going through longer metrology loops.

In the last chapter I explained how models for motion can be developed using a machine's own actuators and then showed how to deploy those models directly in an optimization based motion controller; doing so makes for faster machines and provides feedback that we didn't previously have access to. This improvement to motion control is valuable primarily to machine builders (who *must* tune their machines), but also to users who would like to run their hardware closer to its real physical limits. I also showed how exposing machine controllers' hidden motion optimization (1.3.3) can provide more direct feedback to machine users about how their chosen CAM settings will affect real-world outcomes. Section 2.7.6 describes why this is valuable in a broader systems integration and modelling context and Section 6.4.4 makes its value clear in a user-facing context to help CNC machine operators understand what their hardware will do before they run a job.

However, motion control is only half of the story for digital fabrication machines. The other half is the reason we are here: fabrication, which requires *process control.* This side of the problem is often directly under the purview of machine users who must tune their machines across many materials and configurations, not to mention the infinity of objects they may be trying to create. In this chapter, I will show that we can apply smarter control to one of the most ubiquitous digital fabrication processes: Fused Filament Fabrication (FFF[88]) 3D printing where molten polymers are extruded line-by-line (in "tracks") that sequentially build parts (see Section 5.2.1 for a more complete explanation of the process). Applying model based methods here can make our printers

[88]**FFF:** *Fused Filament Fabrication* is the nomenclature preferred by academics for the 3D printing process of extruding plastic filaments through a hot nozzle which *typically* goes by another three-letter acronym. The other acronym is actually trademarked by a 3D printing firm who shall not be named.

faster, more flexible across materials and nozzles, and re-frames the process tuning step against machine-sourced physical constraints directly rather than as a purely guess-and-check exercise.

5.1.1 Feedforward FFF Configuration and Operation is Lossy and Limits Machines and Operators

5.1.1.1 Separating Low- and High-Level Control Configurations

In printing and in CNC machining, low-level velocity planning and control is done within GCode interpreters but process planning is done entirely within CAM configurations. However, some process physics must also be handled online. In FFF this primarily means operation of the extruder motor to manage filament compressibility as I will explain in Sections 5.2.1.1 and 5.2.3.2. Although these dynamics vary greatly depending on material and melt flow temperatures, the parameters that control them are normally set on a *per machine / per nozzle* basis.

The velocity planners that are used in state-of-the-art FFF printers cannot solve for nonlinear or history-dependent dynamics, the likes of which we will see are present in FFF filament flow. Nor can they integrate directly with models: we write *configurations* for them, which can be *extracted* from models as I showed in Section 6.2.2, but in common practice are often guess-and-check.

This all represents a limit for systems configuration: rather than reflecting on physical models to control hardware, we must develop piecewise heuristics, parameters and lossy algorithms that enable control of those physics in simpler systems but then require subsequent tuning across multiple internal representations i.e. some factors are exposed in slicers, others are expressed as firmware-level configurations, and others (like variable nozzle temperatures and time-dependent melt flow dynamics) are simply not possible to express given the current limitations.

5.1.1.2 Limits to Offline Preset Development

To describe how machines should use a certain material 3D printing machine and material, companies carefully develop *parameter presets* for use with their products that can be used to configure slicers. However, these can take lots of time to develop and limit the adoption of new materials and machines. For example there are thermoplastics that are common in injection moulding that have not been adopted in FFF 3D printing; because learning which materials are suitable for 3D printing must be done using trial and error, filament manufacturers must carefully select the materials that they will introduce to the 3D printing market. Even when they do so, machine companies must then develop the preset that describes how that material can be used with *their particular* hardware and so there are many more filaments available in the market than there are presets; material development tends to lead adoption and even when excellent new materials emerge it can take time for them to arrive in users hands. For example Prusa (a 3D printing company) brought their PCBlend filament to market in October 2022 [290] but their PCBlend *with carbon fiber fill* to market in April 2023 [291], a six-month lag. We cannot know for sure if this lag was due to preset development or some other factors, but this many months between a material's availability as a filament and the arrival of machine-specific presets is not uncommon. Add this to

the fact that machine companies can go out of business, users may modify their machines making factory-developed presets irrelevant, and that people have varying uses for their printers. While some may primarily print i.e., small minifigures for their dungeons and dragons sessions, others may be printing engineering prototypes or even production-ready parts. Presets that suit small, detailed objects are not necessarily optimal for large parts (as we will see).

So, where presets are not available, not suited for the particular use-case, or a machine has been modified from its factory configuration, users must modify existing presets or develop a new one. Doing so requires that they intuit not only how the underlying physics works, but how their parameter selections are modified by the slicer and by the machine controller itself; both the slicer and machine controller contain some *lossy representations* that change those process values in non-intuitive ways.

5.1.1.3 Presets are Tuned Through Lossy Representations

In Section 5.2.2 I describe the tune-able parameters in a state-of-the-art slicer and in Section 5.2.3 I explain how these parameters relate to process physics. While updates to certain parameters can have massive effects on print physics, those relationships are not made clear to the users of 3D printers who are ultimately contending with those physics.

For example if we double the layer height but retain the same translational speeds we double the flowrate of the polymer melt: this has huge ramifications for the physics of the process, but that is not reflected anywhere else in the parameter set. Manually changing print speeds to reflect the new layer height (or otherwise slowing down polymer flows globally in the print job) would mean updating *nine* other parameters (excluding those that can be set by a percentage). Nor are those parameters related to longer-term outcomes like the resulting thermal history of the print's interlayer joints, which are a main indicator of part strength [292]. Inversely, if we increase the nozzle temperature we often unlock more flowrate, but there is no way to update print speeds as a function of temperature, or increase layer cooling time. Instead of having any knowledge of print physics built into the slicer, users need to intuit how these many parameters will relate to the machine's operation.

Part geometry can also change parameters: in Section 5.2.3.4 I explain how printer speeds change based on layer size ensure that parts do not slump (printing at high temperature but without any cooling time leads to molten parts). This means that print parameters developed by hand for small geometries may suddenly fail when we try to print larger parts: the translational speeds that were selected in the slicer were not actually being sent to the machine because this cooling-related slowdown was applied in the small part but not the large part. I mentioned similar effects that emerge from the hidden velocity planning optimization in Section 1.3.3, where GCode interpreters scale target velocities (which are set in slicers) according to acceleration limits (which are set in interpreter firmwares) — again, in some cases small parts may work well with large translational speed selections (even though those speeds are not ever actually reached) but large parts subsequently fail.

Parameter presets represent a wealth of tacit knowledge and hidden labour, but their use also constrains our operation of machine systems when we compare them to real underlying limits. For example in Section 5.8.4 I show that nozzle temperature can be selected based on flow and cooling models, and that this method often selects temperatures that are well above recommended values (and that lead to large increases in print time). This is because the model-based method can easily pick process parameters across a range of temperatures and large parts are *not* constrained by part cooling. Presets that are tuned by hand tend towards colder nozzle temperatures because they are normally developed to print a small test part (the 3DBenchy), which *is* dominated by part cooling; process tuners tend to select nozzle temperatures that work well for this part and then tune all the other parameters at that temperature.

So in the state-of-the-art, feedforward tuning of these parameters is basically guess-and-check: we tune parameters and print parts, watching to see if and where those parts fail. No data is produced during the print and operation is not based on physical models so we have limited recourse to understand *why* a failure occurred — i.e., which physical constraint our parameter selections cause the hardware to exceed.

5.1.2 Chapter Overview

In this chapter I try to re-frame this preset-based workflow into one that is informed by feedback from our machines. In this feedback-based workflow process tuning is done with respect to underlying constraints: rather than picking parameters directly that may exceed those constraints we can tune heuristics *against* those constraints, for example declaring that "for infill tracks (that are not visible in the final part) use as much flowrate as is physically possible and for perimeters (which are) use only 40% of the maximum." This still exposes human-tuneable parameters so that the space of optimal print outcomes can be explored (trading off between speed, precision and strength) but the tuning exercise happens within physics-based bounds that are sourced from the real world.

This chapter first covers basic physics of the FFF process in Section 5.2.1 (for melt flow physics and layer cooling), and then how these physics are integrated into state-of-the-art 3D printing slicers and controllers in 5.2.3. Section 5.2.2 describes the state-of-the-art workflow most often used to produce parts, discussing how feed-forward parameters in those workflows are related (but not directly explained with regards to) process physics across motion, rheology and cooling. I then look at related research in FFF in Section 5.2.4, covering other efforts to model polymer flows (5.2.4.1), to control them (5.2.4.2) and to improve FFF workflows overall (5.2.4.4). As in other cases in this thesis, those efforts all make important contributions in *parts* of the system but struggle to combine models with model-based control or with higher-level optimizations and workflow formulations.

In the rest of the chapter I explain how we can use models for FFF *throughout* the printing process not only to more directly operating machines against their physical limits but also to pick higher level parameters using optimization and to apply model-mediated heuristics to the process.

This means that the tuning steps that remain in the workflow are articulated also against real constraints.

Developing this system happens in a few steps. In Section 5.4 I build two instrumented machines that modify state-of-the-art FFF hardware, enabling them to build as well as use models. The first machine was a development mule, the second is a modified off-the-shelf machine that also demonstrates how the work in this thesis can be implemented without requiring that we rebuild our machines themselves. It also serves as a 1:1 comparison against state-of-the-art workflows because its underlying hardware is the same.

Models for FFF (and methods for fitting them without any external equipment) are outlined in three sections. Section 5.5 develops models that can be automatically fit to data from our hardware, as well as methods for generating that data that can probe boundaries of operation without breaking the system. These models advance the state-of-the-art in a few ways: they can fit flowrates *at the nozzle tip* using a measurement of the internal pressure state; this removes a difficult flow modelling step that requires observing printed tracks directly (Section 5.2.3.3). They also combine isothermal models with time-varying estimates of melt flow temperatures as they deviate from nozzle temperatures; a key limit to other modelling efforts is that they do not connect these two components of melt flow physics. These flow models couple to motor models (see 4.3.1) for the printer's extruder in Section 5.6. Finally, FFF also requires the consideration of longer dynamics, namely part cooling. In Section 5.7, I explain a simple model that we can use for these purposes when combined with some machine-estimated material parameters.

Bringing these models to bear on printer control starts by extending the work from the last chapter. One of the claims that I made in Section 4.1.1 is that state-of-the-art velocity planners cannot be extended to simultaneously optimize over motion *and* process physics without major structural changes, but that the velocity planner formulated there (in Section 4.6) can do so by adding new simulations and extending the cost function. Section 5.9 realizes that by including models for FFF rheology, extruder dynamics, filament compression, and time-varying internal states like *melt flow temperature* in the optimization loop.

Doing so means that we can build a *velocity planner* for FFF that optimizes directly against models of process physics rather than direct parameters that are set by hand by machine developers (for motion) and machine users (for processes and materials). As in the earlier work on motion alone, this allows us to use more of the machine's underyling dynamic range and more intimately connects machine operation with machine physics.

But it is not enough to make the process work *well* because even these more advanced models do not capture all the relevant physics: layer cooling happens on longer time-scales than the velocity planner can consider, our motion models are missing key frequency domain components (machine stiffness), and extrusion at exceptionally high pressures and cold melt flows (which our planner would tend towards if it were simply told to maximize speeds) can cause plastics to degrade and /

or to exhibit *die swell.*[89]

To encode these higher-level heuristics and optima, Section 5.3 develops a new *feedback-based* FFF system that inverts the state-of-the-art; instead of feedforward configurations using presets, the machine itself develops models of flow and motion and those are combined with part geometry (and some heuristics) to produce parts in an end-to-end workflow. This can automatically generate parameters that respect long-term process constraints (Section 5.8) and selects nozzle temperatures that optimize total print time by trading off between hot nozzles (which allow for faster flows) and cold (which reduce layer cooling time), striking a geometry-dependent balance. These plans are then sent to the velocity planner so that dynamical constraints are also respected.

To evaluate all of this, I deploy it all to print seventeen different parts in nine different materials across three types of nozzle (Section 5.10.1). I characterize the reduction in the size of the human-tuneable parameter space and discuss other advantages (like printing in new and renewable materials) of the feedback-based workflow in Section 5.10.3. In Section 5.11.6 I show model-based printing's resilience to human errors and in Section 5.10.4 I show the speed increases that arise from model-based printing's ability to more fluidly move around in process parameter-space. Section 5.10.5 qualitatively compares print quality and shows that visual errors correlate well with model-fit errors.

At runtime, these machines generate more data than they consume (about one gigabyte per hour). I use these data to evaluate the quality of key mathematical models and fits by comparing the controller's predictions vs. real world measurements (Section 5.10.6). I also discuss how these data and models could be used to inform machine designers and users in Sections 5.11.1 and 5.11.3.

Finally, a section on future work in FFF 5.12 discusses the frontier of new projects that are enabled via model-based control, the most exciting of which considers process tuning with pareto optimality (5.12.4).

5.2 Background in FFF 3D Printing

I describe Fused Filament Fabrication (FFF) physics in increasing detail as this section progresses: first I set us up with a good baseline in the fundamentals (below), and then Section 5.2.2 explains how these physics are articulated in state-of-the-art workflows using direct parameters. In Section 5.2.3 I expand on that to explain how those parameters interact in the real world, how they are interpreted by GCode-based controllers, and how some newer controllers have integrated online sensing to improve the state-of-the-art - comparing that strategy to an improvement that this work makes. Finally Section 5.2.4 covers other research on the topic: advanced modelling methods and physical insights, other examples of model-based control for FFF extrusion (but none

[89]Die swell is a phenomenon where plastics extruded at high pressure will return to their pre-extrusion geometries - in FFF 3D printing this means that tracks extruded at, say, 0.6mm width and 0.3mm height will tend back towards the shape of the filament - a 1.75mm cylinder - as they cool. This happens because the filament has not completely melted during extrusion, and retains some *memory* of its original shape.

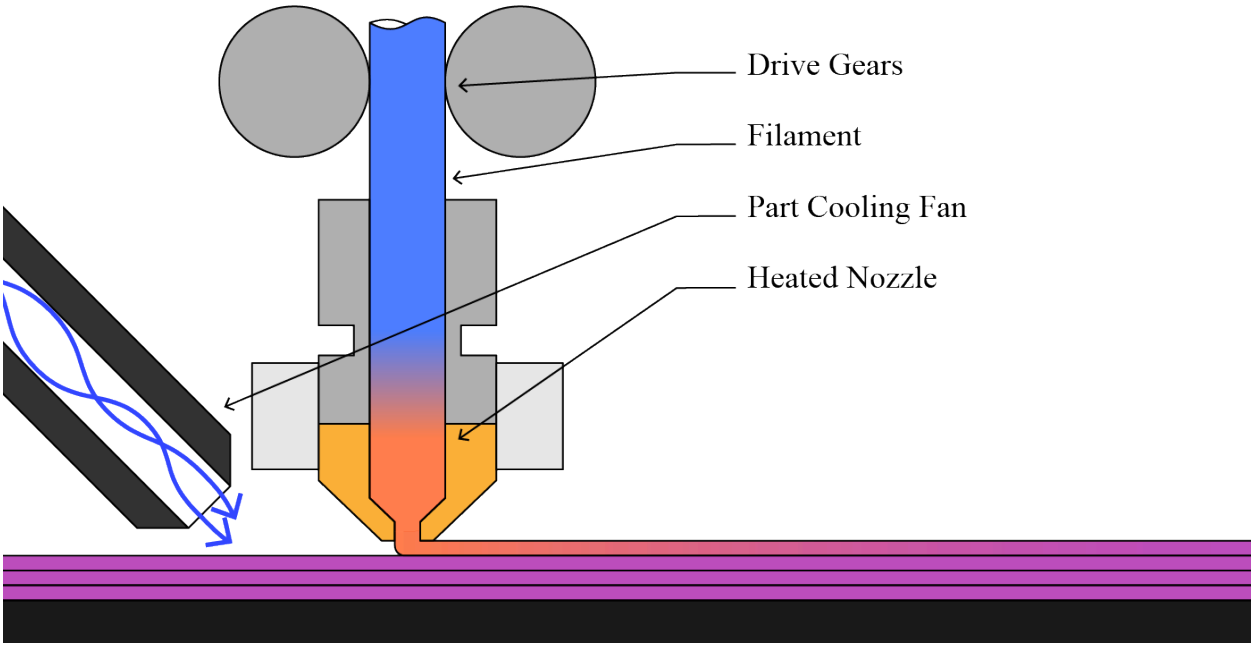


Figure 5.1: A diagram of the FFF process. Gears drive filament into a hot zone, where pressure is generated and filament is melted before it is forced out of a nozzle. Some nomenclature: the *melt zone* is the heated part of the nozzle, *inflow* (or *inlet flow*) is the volumetric velocity into the top of the melt zone and *outflow* is the volumetric velocity at the tip of the nozzle, and we call the melting-and-then-flowing mass of plastic the *melt flow* — this is where the most devious physics happens.

that couple extrusion models into velocity planners), and work that lives in the outer loop which relates process parameter selection to printed component outcomes.

5.2.1 Fundamentals of FFF 3D printing

5.2.1.1 Rheology, Thermodynamics, and Motion

The FFF process is akin to the use of a hot glue gun: we drive a thermoplastic filament into a hot nozzle and extrude *tracks* of molten material. By translating the machine in 3D while we do this, we can additively construct parts.

Extrusion of these tracks must be tightly coupled with our motion controller: to extrude a target *track width* we need to carefully match the flowrate from the *tip* of the nozzle (mm^3/s) with the machine's translational rate (mm/s). The quality of our machine's motion obviously has a lot to do with the machine's ability to produce precise parts, but so does its ability to control the filament flow.

The fundamental physics for flow are *rheological* — i.e., having to do with material flow and deformation under various loading and thermal conditions [293]. The key relationship is between pressure in the nozzle and the flowrate from the nozzle tip: this is why our most valuable tool for FFF model building is a *load cell* positioned in the printer such that we can measure extrusion forces alongside the nozzle temperature measurement (see hardware used in Section 5.4). This

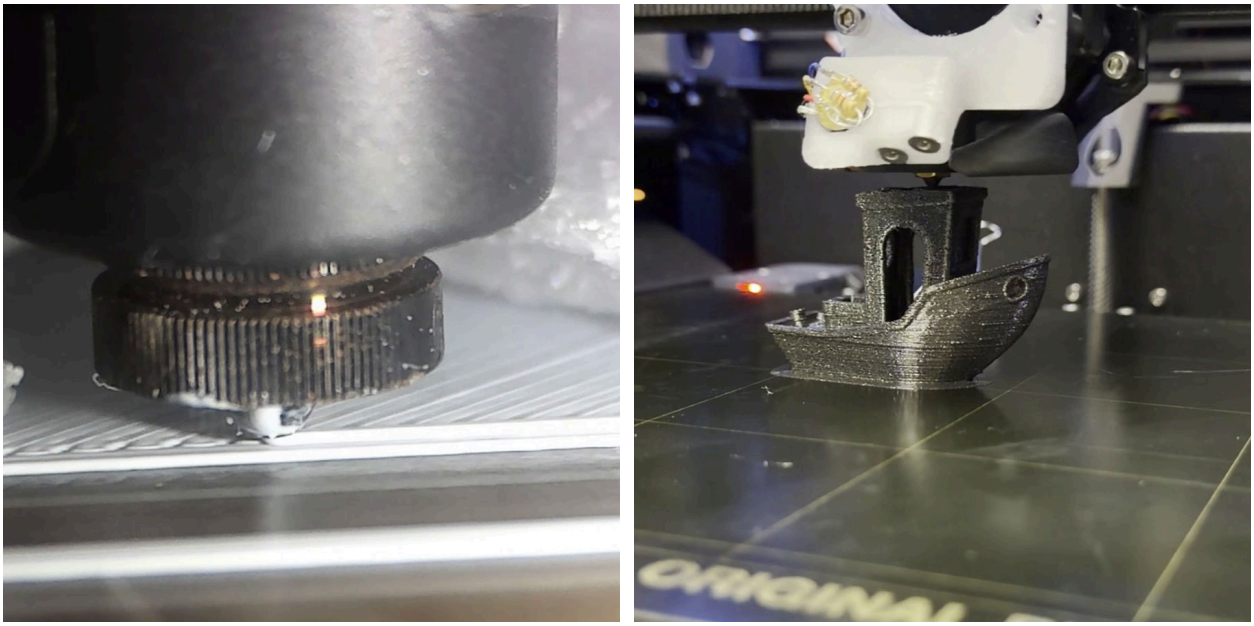


Figure 5.2: A nozzle-camera view of FFF 3D printing (left), showing a track of molten filament being extruded on top of the prior layer. At right, a printed part near completion, with the nozzle printing the final layers.

pressure-to-flow relationship depends on the material's own properties, its temperature, and the nozzle geometry.

The filament also *compresses* before it leaves the nozzle, meaning that changes to flowrate at the tip must be anticipated by our motion controller, preloading the melt zone. These physics are coupled into the extruder motor's own physics, as I explain in Section 5.6.

Thermodynamics are also important: the filament is cold when it enters the nozzle and approaches the nozzle temperature as it transits towards the exit. At high printing speeds, filament heating is the fundamental limit [294]. This phenomenology lead me to develop two models for flow: one characterizing steady-state system behaviour (5.5.2) where thermodynamics dominates and one for *isothermal* behaviour (5.5.3) where the material's actual underlying viscosity dominates.

Depending on who you ask, the FFF process is either incredibly simple (heat, squish, and extrude material) or complex — involving nonlinear rheological models, history-dependent thermodynamics, and interlayer polymer diffusion and cooling models. One of the primary challenges faced in this chapter is to describe these physics using models that can be easily fit to data available *from our own machines,* and that can be useful across high- and low-level planning tasks. This requires that we strike a balance between model fidelity and simplicity.

5.2.1.2 Layer Cooling

The temperature of the melt flow affects its viscosity as we extrude and *also* affects the part geometry: if plastics are *too* molten as we print, our parts *slump* — previous layers not being solidified enough to fully support succeeding layers. This is why FFF printers all include *part cooling fans* (PCF); one of which is visible in Figure 5.13. They can accelerate layer cooling, but

we need then to know when we should try to cool layers and when we should deactivate the PCF in order to promote better interlayer adhesion.

As I learned over the course of this project (and am able to show quantitatively in Section 5.11.1) *cooling* of printed parts is often the *primary* limit to overall print speed. I discuss how state-of-the-art systems articulate and choose cooling parameters and how this work improves upon those methods and combines them with other models to optimally select nozzle temperature *against* cooling models in Section 5.2.3.4.

5.2.2 Parameters in the State-of-the-Art FFF Workflow

So, we can begin to see what the key *physical constraints* to our FFF machines are. How then are machines configured in the state-of-the-art such that users can have success while printing parts?

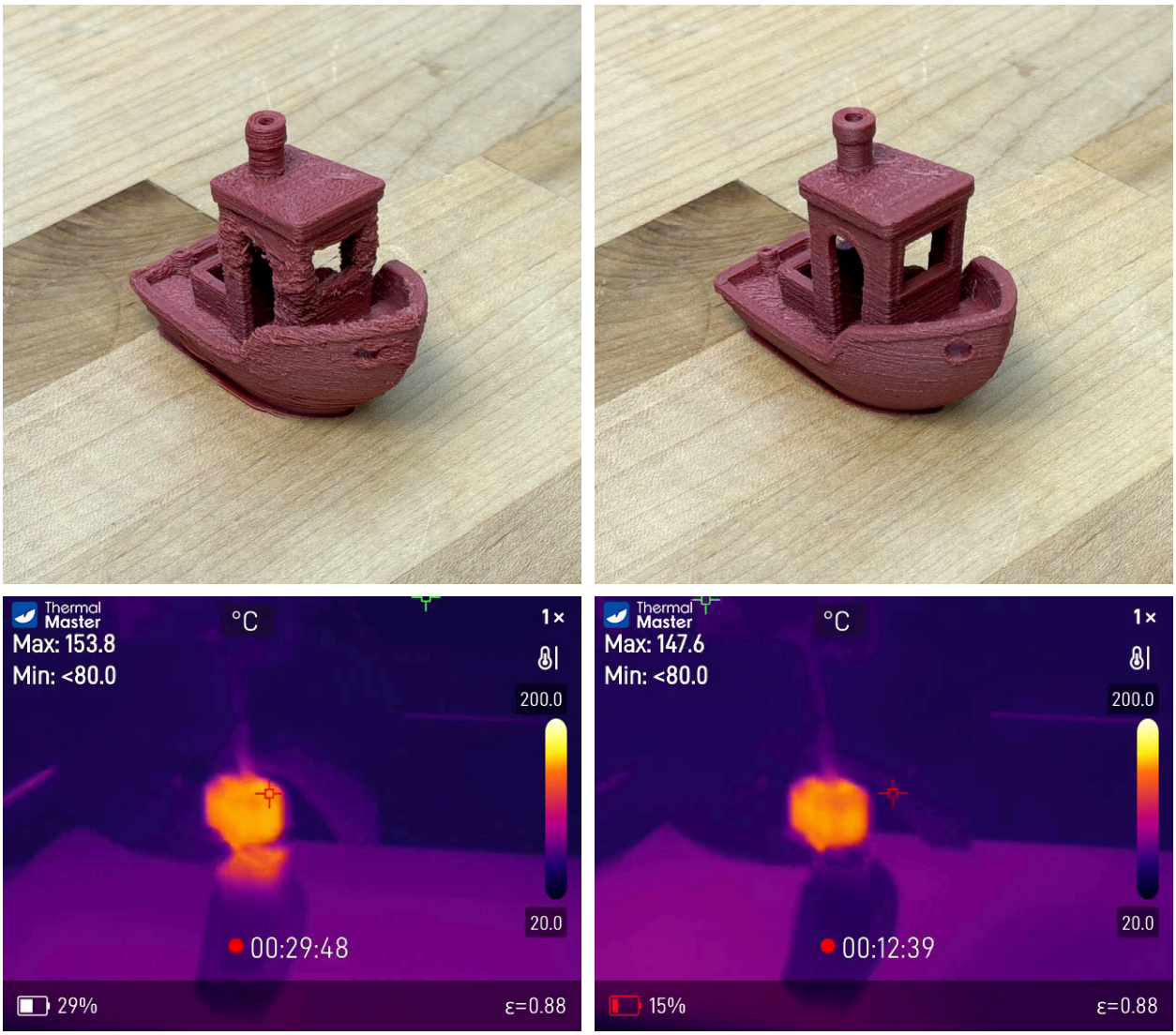


Figure 5.3: A 3DBenchy printed with no cooling effects applied (left), and with model-selected cooling (right). Bottom row shows frames from thermal videos of the prints.

In this section I will explain how these physics are exposed to 3D printer users, and then describe how these parameters relate to physics and how those physics are then controlled (sometimes) in lower level planners.

In the CAM tools for 3D printing (known as *slicers*) users import 3D objects and set parameters to tune the process. Some parameters are geometric: layer height, track widths, external perimeter counts, and infill pattern and density. Other parameters set the machine's speed (how fast the nozzle moves in cartesian space, also called *translational* rates) for different features: outside perimeters, interior perimeters, infill, etc. They select temperatures: for the nozzle, for the print bed, and (sometimes) for the print chamber. Finally, they adjust *cooling* parameters: slicers maintain a *minimum layer time* with the goal that when the next layer is printed, the previously deposited material has nearly solidified.

All told, there are about one hundred parameters to tune in a slicer. Historically, tuning these parameters was a major barrier to the adoption of 3D printing, but *sets* of parameters that are matched to a particular machine, material, and nozzle are commonly available for users now as *slicer presets*. These can be imported, modified, and exported from slicers and shared online in user communities [295]. Presets have been developed manually over the years by the 3D printing community, and are distributed by machine manufacturers (for example Prusa bundles presets for its machines and materials within PrusaSlicer [296]), by material manufacturers, and by user groups.

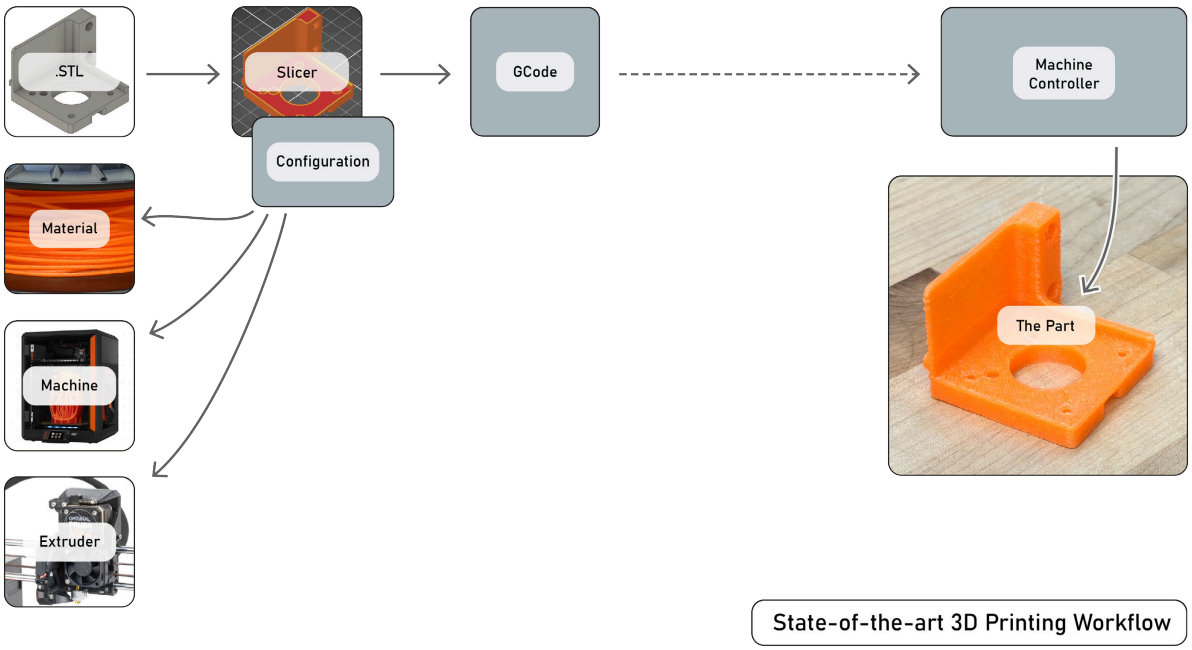


Figure 5.4: A diagram of the state-of-the-art FFF workflow. Part geometries are imported (normally as `.stl` mesh files, which are exported from design software), and slicers are set up with parameters to match the machine, material, and extruder configuration. The slicer writes a GCode, which is transferred to the machine controller to produce the part.

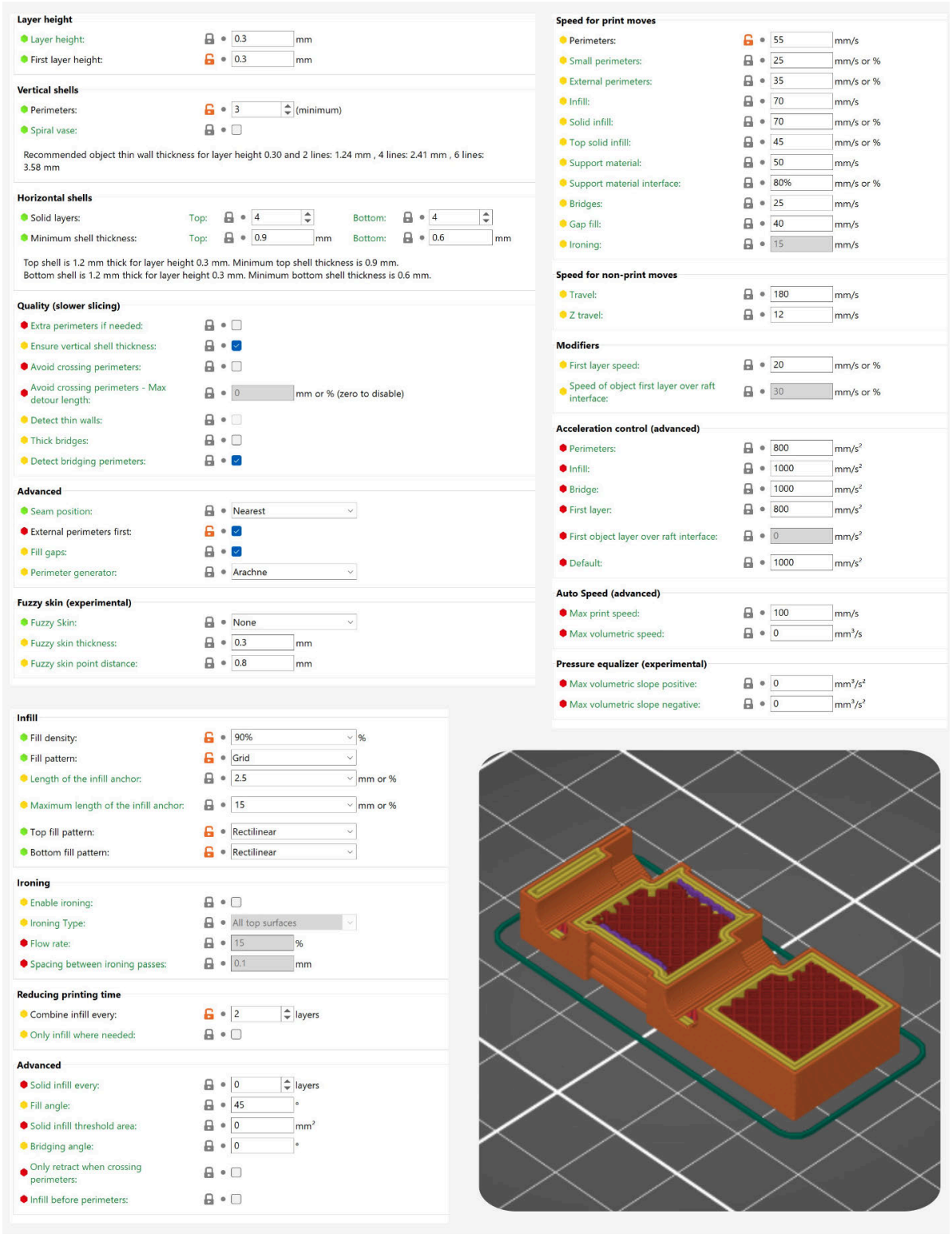


Figure 5.5: A screenshot of some parameters available to configure an FFF print job, from PrusaSlicer. The set which is displayed is incomplete, see proceeding tables in this section. Inset image in the lower right shows the "sliced" part as decomposed into line segments with infill (red), perimeter (yellow), and external perimeter tracks (orange).

Geometric Parameters	Typical Values	Units / Type	Physical Relation
Layer Height	$0.1 - 0.4$	mm	Geometry, Flow
Vertical Shells	$2 - 6$	count	Geometry
Horizontal Shells	$3 - 6$	count	Geometry
Infill	$10 - 100$	%	Geometry
Infill Pattern	i.e 'grid', 'gyroid'	selection	Geometry
Shell Pattern	i.e. 'rectilinear'	selection	Geometry
Support Material	yes / no	selection	Geometry
Support Material Type	i.e. 'grid', 'organic'	selection	Geometry

Table 5.1: Slicer parameters that relate directly to toolpath geometry.

However, we need a new preset any time we change *either* the machine, the nozzle, or the material. It is becoming common also to develop a series of presets for each combination: one for "speed," one for "quality," and one for "strength". This leads to a combinatorial explosion where the space of effective presets we would want explodes. For example with a small set of only four machines, five materials, four nozzles, and three (speed, quality or strength) options we quickly arrive at **240** individual *presets* to develop, each of which contains tens of parameters.

Motion Parameters	Typical Values	Units / Type	Physical Relation
Speeds	—	—	—
Perimeters	$25 - 250$	mm/s	Motion, Flow
Small Perimeters	$20 - 150$	mm/s	Motion, Flow
External Perimeters	$20 - 200$	mm/s	Motion, Flow
Infill	$50 - 350$	mm/s	Motion, Flow
Solid Infill	$20 - 250$	mm/s	Motion, Flow
Gap Fill	$20 - 100$	mm/s	Motion, Flow
Bridges	$10 - 50$	mm/s	Motion, Flow
Support Material	$10 - 100$	mm/s	Motion, Flow
Support Interface	$5 - 50$	mm/s	Motion, Flow
First Layer	$25 - 50$ (or as %)	mm/s	Motion, Flow
Travel	$50 - 500$	mm/s	Motion
Z-Travel	$10 - 50$	mm/s	Motion
Accelerations	—	—	—
Perimeters	$1500 - 5000$	mm^2/s	Motion, Flow
External Perimeters	$1000 - 3000$	mm^2/s	Motion, Flow
Infill	$2000 - 10000$	mm^2/s	Motion, Flow
Bridges	$500 - 2000$	mm^2/s	Motion, Flow
First Layer	$100 - 1000$	mm^2/s	Motion, Flow
Travel	$2000 - 10000$	mm^2/s	Motion
Travel (short)	$100 - 1000$	mm^2/s	Motion

Table 5.2: Slicer parameters that relate most directly to printers' motion system.

Extrusion Parameters	Typical Values	Units / Type	Physical Relation
Extrusion Widths[a]	—	—	—
Default	$d_{nozzle} \cdot 1.25$	mm	Geometry, Flow
First Layers	$d_{nozzle} \cdot 1.25$	mm	Geometry, Flow
Perimeters	$d_{nozzle} \cdot 1.25$	mm	Geometry, Flow
External Perimeters	$d_{nozzle} \cdot 1.15$	mm	Geometry, Flow
Infill	$d_{nozzle} \cdot 1.1$	mm	Geometry, Flow
Solid Infill	$d_{nozzle} \cdot 1.0$	mm	Geometry, Flow
Top Solid Infill	$d_{nozzle} \cdot 0.9$	mm	Geometry, Flow
Support Material	$d_{nozzle} \cdot 0.8$	mm	Geometry, Flow
Temperatures	—	—	—
Nozzle Temperature	$160 - 300$	$^\circ$ C	Flow
Bed Temperature	$40 - 120$	$^\circ$ C	Bed Adhesion
Chamber Temperature	$40 - 80$	$^\circ$ C	Part Warping
Cooling Settings	—	—	—
Fan Enable Layer Time	$5 - 50$	s	Part Cooling
Minimum Layer Time	$8 - 20$	s	Part Cooling
Filament Settings	—	—	—
Maximum Flowrate	$10 - 50$	mm^3/s	Motion, Flow
Extruder Settings[b]	—	—	—
Retraction Length	$0 - 5.0$	mm	Motion, Flow
Retraction Speed	$10 - 100$	mm/s	Motion, Flow
Deretraction Extra Length	$0 - 1.0$	mm	Motion, Flow
Deretraction Speed	$5 - 50$	mm/s	Motion, Flow

Table 5.3: Slicer parameters that relate most directly to polymer flow physics, cooling, and extruder dynamics.

[a] I've included extrusion widths here as a function of nozzle diameter (d_{noz}) but they are often also calculated with respect to layer height; if we try to extrude i.e. a 0.2mm track width at a 0.4mm layer height, the track does not connect to the previous layer. These are not a commonly re-tuned value, as defaults are easy to arrive at using those known variables.

[b] Settings for retracts can be tuned in two places: connected to the *machine* (which may have an amount of baseline extrusion slack) and as an "override" in the *filament* related settings, for i.e. very flexible filaments that may require extra retraction.

The values we set in slicers all relate explicitly to the machine instructions (how fast to travel, how thick to make the layer, etc). They relate *implicitly* to the physics of 3D printing, which has mostly to do with flowrates and temperature of the polymer melt. To explain in more detail, I list the most important slicer parameters and how each relates to our control task in the tables here. These are separated into three subsets: parameters relating to path geometry (5.1), to the

Motion Parameters	Typical Values	Units / Type	Physical Relation
Fan Speed Maximum	$60 - 100$	%	Fan Power, Matl. Heat Capacity
Fan Speed Minimum	$0 - 40$	%	Fan Power, Matl. Heat Capacity
Enable Fan When <	$30 - 60$	Layer Time s	Print Temperature, Heat Capacity
Slow Part When <	$10 - 30$	Layer Time s	Print Temperature, Heat Capacity
Minimum Print Speed	$10 - 30$	mm/s	Maximum Slowdown

Table 5.4: Slicer parameters that relate to layer cooling.

extrusion process (5.3), and to the printers' motion system (5.2). While it is possible to separate these into three main categories, the physics that each parameter affects are coupled.

5.2.3 Physics Management in the State-of-the-Art FFF Workflow

5.2.3.1 Flowrate Limits Combine Translational Rates and Track Geometry

We can see that slicers allow users to tune printer motion physics via direct application of maximum acceleration and velocity. Chapter 4 showed that these alone do not completely describe motion control constraints: maximum acceleration is a function of speed not constant throughout speeds, and machines can decelerate much faster than they can accelerate because friction helps to stop the hardware.

In purely kinematic systems our top speed limit is based only on actuator power / voltage and friction. When we introduce melt flow physics, we add a velocity limit based on maximal flowrates: the tracks we extrude have a certain width and height and so to increase translational velocity we must also increase flowrate. If we change the nozzle temperature we increase this flowrate limit, and if we update the path geometry to print e.g., increasing layer thickness increases the flowrate *required* per unit of translational velocity by the same proportion. However, slicers expose tuning of flowrates in terms of *translational rates* alone; changes to either of these values are not reflected into those rate limits even though the physics are deeply coupled. They do allow that we set a global *maxima* for flowrate per nozzle, but that is itself not updated if we increase the nozzle temperature *or* even if we change the material definition. The flow models in this chapter show that these two factors change maximal flowrates significantly.

So, motion constraints can emerge from either motion or process constraints an in the state-of-the-art this coupling is not exposed. It is not either the case that we can simply pick one of these constraints to frame the problem because realtime operation is dynamic and includes both purely motion moves (jogging, where just motion is important) and extruding moves (where both factor). Track geometry also varies; if we print a track whose width goes from thin to thick suddenly, we need to decelerate the machine in the interval where the primary constraint changes from the motion system's physics (for the thin component) to the flow system's physics (in the thick portion).

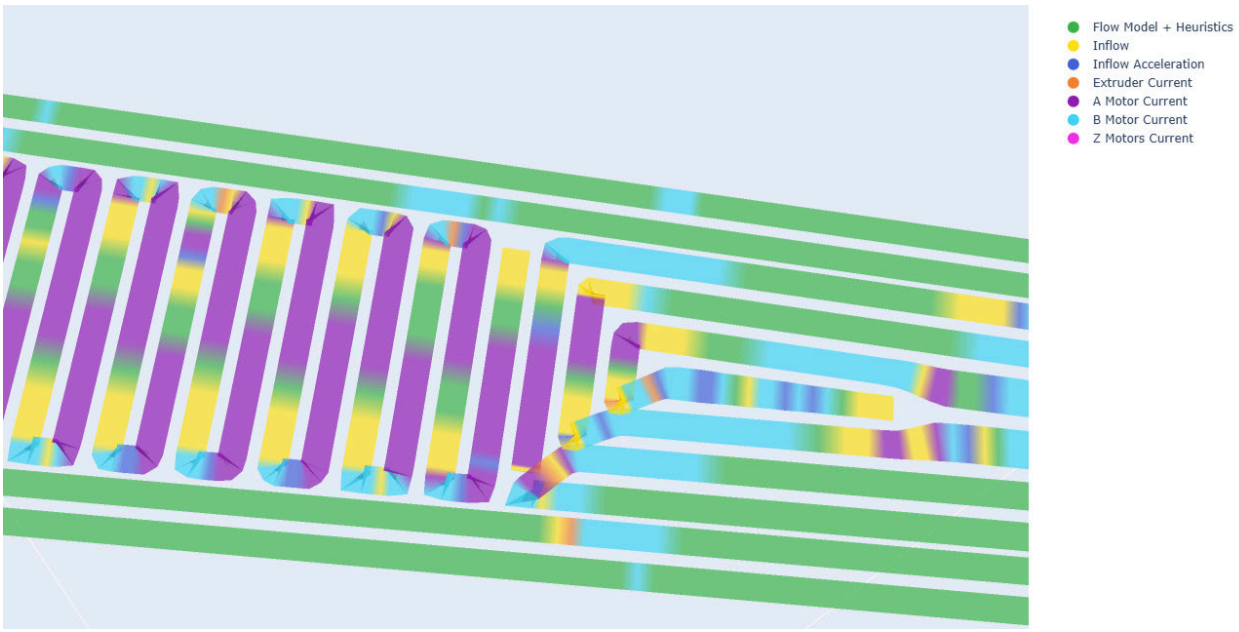


Figure 5.6: Colours here map each trajectory segment to the machine physics that limited overall velocity at that point: green tracks are flowrate limited, yellow, deep blue and orange tracks are limited by extruder system dynamics and purple and light blue tracks are limited by the motion system. I describe how it is generated using the methods from this chapter in Section 5.11.1. At any time, the physics that limits overall speed vary depending on machine performance and track geometry, for example here we see two tracks that are straight *but* have varying thickness. In these cases the machine must decelerate *while* operating the extruder system such that the width change is controlled precisely; motion and extrusion systems are deeply coupled.

5.2.3.2 Extrusion Dynamics via Linear Advance and Retraction

Machines are constantly accelerating and decelerating and do so in relation to motion system dynamics, but in FFF 3D printing we have an additional order in the dynamical system to consider which is present due to filament *compressibility;* filament is squishy (and is squishier when it is hotter). This makes *cornering* in 3D printers even more complicated than it is in purely kinematic systems: when we decelerate into the corner the extruder motor needs to rapidly *unload* pressure in the nozzle and when we accelerate it needs to rapidly *load* pressure in advance of the motion system's own acceleration.

This means that the maximal overall printer velocity through a corner can be constrained by a number of factors. First, curvature defines how fast the *motion* system can traverse the corner. Flowrate limits and track width define how fast it enters the corner and what speed it will try to reach on corner exit. Then, as it decelerates it must unload the nozzle pressure accordingly and so the *extruder motor's* own dynamic limits can constrain the system: if it cannot unload pressure as fast as the motion system can slow down to meet the curvature, the motion system must decelerate *slower* than it may otherwise, the same is true as the machine accelerates out of the corner. There is an asymmetry here as well because it is easier to release pressure in the extruder than it is to generate pressure — similar to the earlier notion from kinematic control alone that stopping with friction is easier than accelerating against friction.

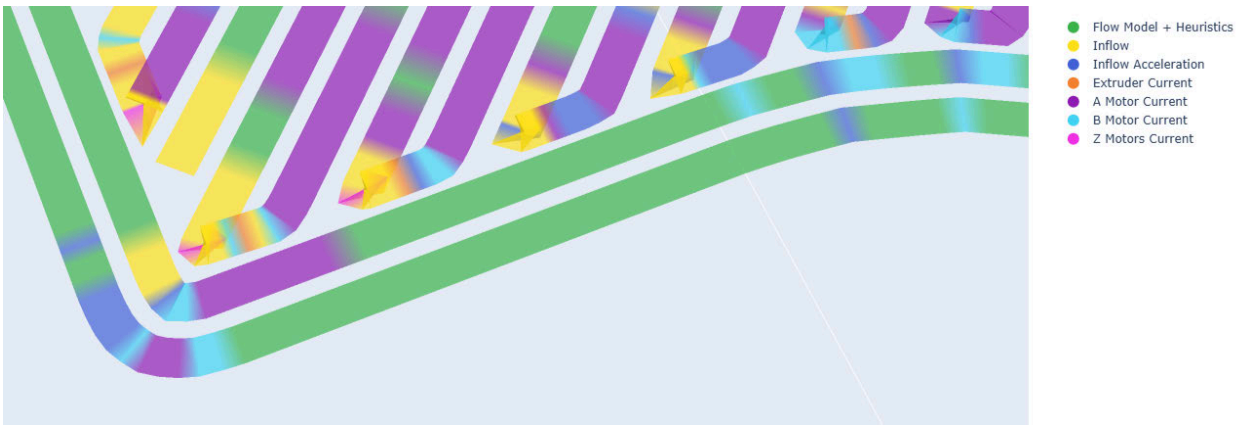


Figure 5.7: Like Figure 5.6, this plot maps physics limits to trajectory segments. In this case we see two corners, as the machine approaches each it must slow down to satisfy motion control constraints (motion limits are in light blue and purple) while also dynamically controlling the extrusion system in sync. Both plots also show infill sections where the machine repeatedly traverses small corners, and show how overall limits routinely change between motion and extrusions systems.

Filament compressibility is managed approximately in state-of-the-art printer firmwares using an approach called *linear advance* [297] or *pressure advance* [298]. This is a firmware setting that adds velocity to the extruder motor for every unit of *acceleration* of the motion system in order to pre-load and un-load nozzle pressure. This is tuned as a single spring rate that is tuned *per nozzle diameter,* even though the flow models in this chapter show that filament spring rate can vary greatly across materials and melt flow temperatures.

Because pressure advance modifies extruder *velocity* based on machine acceleration and acceleration can change instantaneously in these controllers, this injects instantaneous changes to extruder velocities in the control of 3D printers. The motor models from Section 4.3.1 shows that even instantaneous changes to a motor's *acceleration* violate important motor dynamics. In Section 5.2.4.2 I will discuss some research efforts that improve control of FFF extrusion dynamics using models; these make significant improvements on the simpler pressure advance scheme, but they too couple control to an existing velocity planner's trajectories (using their outputs as a reference signal for a feedback-based extrusion controller) even though the extruder's own dynamics should limit operation: neither pressure advance or these new methods completely couple motion control and process control, or can show us which constraint limited overall performance of our systems at any given point.

So, these methods manage *cornering* with respect to extruder dynamics, but we must also *stop* flow during purely motion moves, i.e. when we finish one track and begin another. Done properly, this step would also involve computing how the extruder system should be driven such that pressure in the nozzle goes to zero when we make a purely-motion move, based on flow and extruder physics. In the state-of-the-art, each filament can define simpler *retraction* parameters, these define the length of filament that should be pulled back from the nozzle tip in order to come to a full stop. As in other cases, the underlying physics depends on nozzle temperature and nozzle size as well, but here slicers expose just one set of parameters per material.

5.2.3.3 Direct Measurements of Output for Flow Calibrations

Understanding flow dynamics is challenging because it is difficult to measure how much filament is flowing from the *tip* of the nozzle at any moment, known as "outflow" or Q_{out} in the modelling in this chapter. Measuring the flow that goes *into* the nozzle is easier because it is basically equivalent to our extruder motor velocity (times some scalar), but between that value which we can control and the outflow are the compression dynamics and flow rheology.

To establish ground-truth measurements for these systems, state-of-the-art controllers make direct measurements of outflow by looking at extruded tracks of polymer. I implemented one example of this type of instrumentation in Figure 5.8: a camera points at extruded tracks and we use computer vision to measure track widths, then estimating flowrates using a known layer height and known translational velocity. Bambu Labs 3D printers make similar measurements using small line lasers, and the Rubedo system [299] developed in the open source does the same.

These methods are used to tune pressure advance parameters on a per-setup basis, but require that measurements be made under specific process parameters: measurements must be taken on extruded tracks on the build plate itself because differentiating layers can be difficult, instrumentation must be aligned with the tracks themselves (we can only make measurements of tracks that were extruded along the x-axis, for example), and the linear translation rate of the machine must be well known: normally these tests are done at a fixed translation rate so that machine acceleration does not couple into changes in track width.

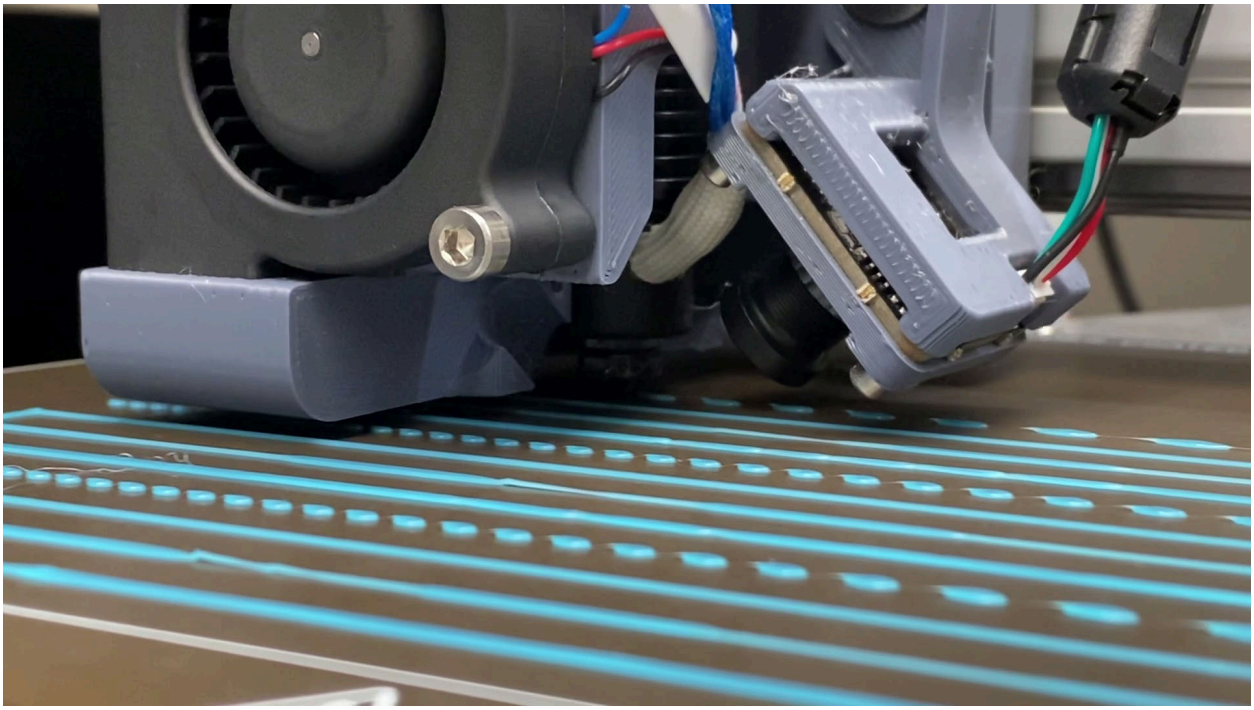


Figure 5.8: This system is measuring polymer flow from the nozzle tip using a camera (to the right of the nozzle) that measures track widths. The modelling in this chapter uses computational metrology and the flow system's *internal* pressure state to make the same measurements, eliminating the need for these types of out-of-band measurements.

In this work I use the idea of computational metrology to perform the same measurements: in developing a model that connects the internal pressure state to both outflow and inflow using rheology and compressibility, we can fit models against this internal state that extrapolate to predict outflow (Section 5.5.3). This advances the state-of-the-art because it removes measurement complexity (no cameras or line lasers required), enables higher frequency measurements (at hundreds of Hz, rather than camera frame rate limits), and means that we can measure flow dynamics *all the time* rather than only on the first layer and in certain directions.

5.2.3.4 Minimum Layer Times for Layer Cooling

OK, the aforementioned physics concern polymer behaviour *during* extrusion, but in Section 5.2.1.2 I explained the importance of polymer cooling *after* extrusion. In particular, we need to ensure that previous layers are solidified enough to accept subsequent layers without slumping before we print on top of them.

This is one of the more complicated phenomena to model properly in 3D printing because it depends on the material's heat capacity, the selected nozzle temperature, the part cooling fan's performance, and the part *geometry* because cooling is slower in i.e. dense parts than it is in sparse prints etc.

In state-of-the-art slicers, we tune a few values for *minimum times* per material, and for cooling fan utilization. Slicers will turn fans on if a layer takes less than a minimal time and will slow the layer down (changing translational rates) if it falls under a second minimal time.

In Section 5.7 I extend this method to calculate minimum cooling times based on volumetric heat capacity estimates made from the printer itself alongside estimates for part cooling fan performance and a simple thermodynamic model for diffusion of heat into previous layers an into air. Then, because we can select process parameters across a range of nozzle temperatures, we can pick nozzle temperature on a per-part basis for time optimality by trading off between hotter flows (which are faster) and layer cooling time (which increases with hotter flows). A clear optimum emerges where smaller parts should be printed close to minimal flow temperatures whereas larger parts can easily exploit high flowrates with larger nozzle temperatures because layer times are long *anyway* since the parts are big.

This leads to significant time improvements in large parts and suggests that printers have been mostly tuned by hand to produce small parts — not necessarily because most 3D printing users want to print only small parts, but because this is the faster way to tune parameters: a loop around parameter tuning for a large print being obviously slower (by hours) than the same loop around printing a small part.

5.2.4 Related Research in FFF

There is a lot of interest in FFF printing in the literature, and so the applied physics are quite thoroughly understood [293], [300]. In this section I will cover existing models for polymer flows,

related work that applies those models to realtime control, and then research that takes place at higher levels to develop improved workflows overall.

As in the background section on related research for motion control with models (Section 4.2.8) the through line here is similar: there is excellent research at lower levels that model advanced machine physics and also research at the workflow level that shows how slicer and CAM parameters relate to print outcomes, but work that connects models directly to controllers is limited. For example most model-based controller for CNC machines integrate *on top* of existing velocity planners (Section 4.2.8.1), and where they do integrate directly they still use direct parameters (i.e. maximal jerk, acceleration, and velocity values) for planning (4.2.8.2).

In the examples here there is extensive modelling research on polymer flow physics both within the nozzle *and* outside the nozzle as tracks and parts cool. There is one example where researchers go to great length to make online model-based measurements of these physics, but they cannot connect those measurements back into a predictive controller, (they are also using an industrial robot arm with a proprietary controller, which may have been the limit). One group of researchers has applied some modelling to realtime control of FFF extrusion, but they too connect their controller *below* velocity planners, taking their outputs as a reference / target signal (their models only describe shorter scale compressibility dynamics).

In this chapter we try to connect these models more directly to velocity control, allowing constraints from the flow system to reflect *back* into the velocity plans. We do this by coupling the two systems in the same controller rather than splitting them hierarchically, and also include those time-varying dynamics. This is enabled both by developing *simpler* models (rather than e.g. complete CFD-based solutions) that can be fit to data from the machine and in leveraging the flexibility and computational power of the velocity planner that I developed in Chapter 4. Computing both of these systems together at the fine-grained resolution required for velocity planning is possible because the work from Chapter 2 allows us to relocate the velocity planner onto the GPU. This gives us many orders of magnitude more compute power than we would have in a controller deployed in firmware, while maintaining a connection between that GPU and the embedded devices that execute control plans and also generate data.

5.2.4.1 Related Work on Modelling of FFF Flows

I gave a brief overview of FFF rheology in Section 5.2.1.1; polymers are driven into a melt zone where thermodynamics, compressibility, and rheology all interact to deposit tracks of molten material that then subsequently cool and become printed components. There are many factors that come into play during this process and the underlying physics turns out to be quite involved. That is reflected in a very active group of researchers who are all trying to understand different components of the process.

Of primary relevance to this work is a line of research begun by TJ Coogan and David Kazmer, who developed an instrumented extruder similar to ours in [301] — it also adds a load cell between the extruder motor and the nozzle to estimate melt flow pressures and turns the hotend into a rheometer. But melt flow viscosity alone is not sufficient to describe the operation of a hotend,

Kazmer continued this work to characterize the *dynamic* behaviour of filament during extrusion his paper on concurrent characterization of filament compressibility and flow in [302], [303] and [304], which helped me to develop the dynamics model I use in the current iteration of the FFF flow model that also model compressibility alongside rheology. These papers were also fundamental to my earlier work in [305] (see Figure 5.16), where we combined online model-building with parameter selection in a reduced space to print with unknown materials but did not integrate dynamical models for flow. For a closer look at melt flow rheology on its own, [306] analyzes some common 3D printer thermopolymers using capillary and parallel plate rheometers and shows that power law equations (which I use here) are a good basis for melt flow models.

These efforts cover compressibility and *isothermal* rheology, i.e. melt flows whose temperatures are not time-varying and can be assumed to be basically equivalent to the nozzle's temperature. However, nozzle thermodynamics turn out to be a main limit to overall melt flow dynamics. Jamison Go's [294] discusses absolute limits to print speeds based on nozzle thermodynamics (providing insight into how printers should be improved accordingly) and Mackay's [307] models flowrates through a hotend as a function of nozzle temperature, mapping flowrate failures across nozzle temperatures.

Kazmer's [302] notes that the connection between isothermal and thermodynamic aspects of a polymer melt flow are of key importance to understanding real printer operation; after only a few seconds of flow, the thermodynamic limits of the extruder begin to dominate the system's phenomenology. In practice printers operate under non-isothermal conditions that are difficult to understand because it is nigh impossible to measure the internal temperature of the melt flow directly. The measurements that we *can* make are a few orders away from that internal state: thermocouples in most printers are located in the heater block which is much more thermally massive than the melt flow and are normally tens of millimeters away from it. In [308] the authors develop a hotend that locates a thermocouple and pressure transducer much closer to the melt flow, but still only at the melt zone's *wall* rather than at it's core, and located 6mm above the nozzle tip. They still report 6.5° C of temperature deviation during extrusion at only $5.5mm^3/s$ (state-of-the-art printers can operate closer to $50mm^3/s$).

Zhang et al's [309] uses a finite-element (FEA) multiphysics simulation of an FFF printer's hotend to model melt flow thermodynamics within the nozzle and Nzebuka et al [310] uses a numerical model for a similar simulation, which they verify against real-world data. Pigeonneau's work in [311] develops the complete mass, momentum and energy equations for non-isothermal flows in the melt zone of an FFF 3D printer and also uses FEA to solve the equations. They show that their model can provide the correct working conditions for the extruder in steady-state[90] inlet flow conditions; based on nozzle temperature and some material parameters they can predict the maximum flowrate.

[90]Technically in rheology "steady-state" flow means that a flow is "fully developed," and so temperature in the flow is unchanging. In the modelling here I call "steady state" the flow conditions that we reach after extruding at the same *nozzle temperature* and inlet flow i.e. this is steady state of the flow dynamics even though the flow's internal temperature changes significantly from one end (the inlet) and the other (the nozzle tip).

These efforts all represent more advanced modelling techniques that have been applied to the same system that I am modelling in this chapter, but there is a split between those who model isothermal flow and compressibility and those who model non-isothermal steady state flows. In control, we need to connect these to one another to simulate how the system evolves over time based on time-varying inputs from the past.

The authors of [274] explicitly set up their modelling workflow for the purposes of integration within a realtime controller. They use a thermal camera and thermocouple to model non-isothermal (steady-state) flows and build a "response map" for a range of steady-state flow conditions that is similar to the one that I develop in Section 5.5.4, which predicts melt flow temperature decrease as a function of nozzle temperature and inlet flowrate. While they develop their controller for realtime performance (focussing on high performance computer vision to process data), they do not ultimately connect this controller to a machine system and their model does not include shorter-order dynamics like compressibility.

5.2.4.2 Related Work on Real-Time FFF Control

I covered other work that combines process and motion control in Section 4.2.6.3 (in industrial systems) and in 4.2.8 (in other research). Many of those examples use *feed override* outputs from process-aware controllers to modify velocity planners rather than solving the problems together. This works well but requires that the velocity planners themselves are modelled so that their internal behaviour can be anticipated. There is one example there (in 4.2.8.2, for process control in CNC machining) that seems to completely combine the optimizations into one solver, although theirs is still formulated directly against direct parameters rather than models for motion control. I discuss in more detail how the planner used in this work is different from other motion control work in Section 4.2.9.

There is more research overall that connects process modelling to motion control in CNC milling (again see Sections 4.2.6.3 and 4.2.8), but here we are concerned with FFF extrusion. I mentioned [274]'s efforts to formulate steady-state polymer flow models such that the *could* be integrated with control, and each of the papers in Section 5.2.4.1 above mention integration of their modelling efforts with control as a primary motivation, but there are only a few efforts to actually do so.

Pinyi Wu's work is one example, he develops a simple nonlinear model for filament flow that is based not on physical properties of the material but on the time-constant τ of the delay between the extruder motor's velocity and the flow velocity, which is a useful approximation that re-frames the problem in terms of delay. This is fit to data from a machine using the track-width measurement methods that I mentioned in 5.2.3.3. In [272] he applies this in two ways: the first is to modify that machine's *velocity planner* so that the extruder controller (which is left as-is, i.e. it uses linear advance) is corrected according to the measured τ. The second method modifies the velocity control *only* of the extruder motor to effectively improve the linear advance algorithm. In his PhD thesis [273] (and separately in [312]) he develops a hybrid extruder that acknowledges that FFF control has fundamentally two components: steady-state pressure generation and high-bandwidth pressure updates that must be applied while the system is cornering - he splits this

into a slow extruder mounted via a bowden tube[91] and a fast direct-drive extruder that provides higher bandwidth control.

These methods benefit the extruder system based on model insights, but they do not connect directly to process physics - instead using controls-oriented / delay-based compensations that are fit to physics and only apply on short time-scales (i.e. they miss the non-isothermal component). They also work in one of two ways: to shape velocity planner outputs to suit extrusion dynamics, or update extrusion dynamics to better respond to velocity planner outputs, using those as a reference signal. That is, neither approaches combines velocity control *with* extrusion control. To compare with work in this thesis, my approach combines physics-based models for extrusion with more advanced models of machine motion and kinematics, which should yield overall improvements to print speed - taking the machine system *as a whole,* whereas Wu et al's work focuses mainly on the optimization of extruder commands. Ours also integrates the additional load cell sensor which allows us to bootstrap models on previously unseen filaments and develops an end-to-end workflow that extends the controls method up into higher level parameter selections by developing a model that captures both isothermal and non-isothermal flows.

In the background section on systems design, I also covered the integration of rheological control with motion control using "TCodes" (Section 2.2.4.8) where researchers work around interpreters entirely to produce control outputs that are synchronized against the *expected* time alignment of their machine's GCodes to their custom control for gel extruders [162], [163]. This method allows those researchers to combine their own models for extrusion control with off-the-shelf motion systems but also misses conjoined optimization of the two problems: flow control is overlaid on motion plans already generated by off-the-shelf velocity controllers.

Finally I would like to cover Douglas Brion's excellent work on FFF printer control using deep learning and machine vision [313], [314]. This takes a higher level approach, fitting cameras to a fleet of 3D printers and running them for weeks while collecting image data using GCodes that had segments with known good parameters (which were labelled as such) and segments that had known bad deviations from those parameters across lateral feed rate, z-offset, flowrate commanded and hotend temperature. That then *generalizes* into a framework where the vision model is used to estimate the current offset from "good" parameters, which can be used to update the relevant parameters on the fly (and subsequently modifies input GCodes to the machines). This is a completely different approach to the one we are taking here which seeks to model flows on a first-principles basis and can be bootstrapped without known good parameters or a large image dataset like the one used in Brion's work. The work here also provides much higher bandwidth data and controls responses, again integrating with motion controllers themselves to find *other* parameter spaces that are viable rather than generalizing on a set that has been developed heuristically. However, their results are excellent and show the value of big data in 3D printing: while more generalizeable approaches with hidden internals can be difficult to interpret and don't (for example)

[91] A *bowden tube* is a design used in many FFF printers where a drive motor pushes filament into a (normally PTFE / Teflon) tube that is only marginally larger than the filament, which is connected to the hotend. This moves the extruder mass off of the moving gantry, but introduces more delay between the extruder motor and extrusion.

tell us anything new about our systems, they can perform well for complex tasks. In Section 7.7 I make a broader connection between the approaches that I take in this thesis and "AI-Powered" approaches like Brion's contributions.

5.2.4.3 Related Research in FFF Cooling Physics

Most of my work in this chapter is concerned with the *extrusion* dynamics of polymer flows, but I have also mentioned that there are important physics on the other side of the nozzle tip — i.e. in how the part cools and solidifies after extrusion.

For example, we would like to be able to optimize prints to be maximally *strong* as well as precise and fast. Part strength also relates to nozzle temperature and part cooling because interlayer weld strength is a strong function of the layer interface's thermal history [292], [315]; more time above the plastic's glass transition temperature equates to more diffusion of polymer chains across the weld. The inter-layer pressure exerted by the melt flow during extrusion also factors. High shear rate extrusion also leads to increased die swell — which is shown in [316] — and die swell can leave unwanted stresses in finished parts that change geometry and further reduce strength.

These physics are complex because they involve thermodynamic models of polymers alongside models for e.g. how molten polymers crystalize or set[92] as they cool. [317] uses multiphysics simulation to understand this behaviour, modelling both die swell and diffusion between recently extruded tracks and the rest of the part. Zhang et al's [318] extends work from the group that I mentioned earlier on non-isothermal flows *within* the nozzle [309] to model non-isothermal cooling behaviour *after* the nozzle, now estimating the track's internal thermodynamics using thermal image data of the track. In one of their tests, the exit surface temperature of the track was 194° C whereas the core temperature was only 164° C, a significant delta.

These works also note that the short-term dynamics of extrusion also matter when considering longer-term cooling physics because the thermodynamics *in* the nozzle affect how much melt flows are reconstituted there (or not), i.e. flows that become more molten during extrusion also weld to one another more completely post extrusion. McIlroy's earlier 2017 paper [319] reflects this: parts that are extruded at higher pressures do not weld as well because their polymers become "disentangled" at high pressures.

This is an important set of research for this workflow because these factors are missing from the models that I use to select print parameters. For example in Section 5.8 I pick nozzle temperatures based on a print speed optimization and some applied flowrate heuristics and in Section 5.9 my velocity planner selects maximal flowrates against those heuristics and dynamical models of the system. Maximal pressures in each of those cases are based on flow systems and extruder motors alone (and motion systems), but according to this research we may also want to avoid extremely high extrudate pressures to promote layer cooling and reduce die swell.

So, yet again everything is coupled. In Section 5.12.4 I discuss the possibility of eventually formulating the FFF workflow as a physics-based trade-off on three fronts: print speed, precision,

[92] Some polymers are amorphous whereas others are crystalline, so behaviour at this stage (and during extrusion) can vary based on this property as well.

and strength. Each of these relates in different ways to melt flow physics *in* the nozzle, part cooling physics *outside* of the nozzle, and motion dynamics of the machine.

5.2.4.4 Research on FFF Workflow Optimization

On the note of combinatorial optimization through the trade-offs between speed, strength and precision, I want to point out another selection studies on the relationship between slicer settings and print performance [320], [321], [322], [323], [324]. Rather than using physics-based analysis, these studies all operate in an outer loop around the state-of-the-art workflow - i.e. they optimize *slicer* parameters directly and observe the results in printed parts. As I've noted there is a great deal of loss between the parameters that slicers expose, the GCodes that machines consume and the control outputs that GCode interpreters ultimately use to operate machines, so the results from each of these studies is limited. However, they do show that optimizing print parameters alone and eschewing all of these complex physical models is possible, it is merely more difficult to trace and adapt across varying machines, materials, nozzles and geometries.

5.3 A Feedback-Based FFF Workflow

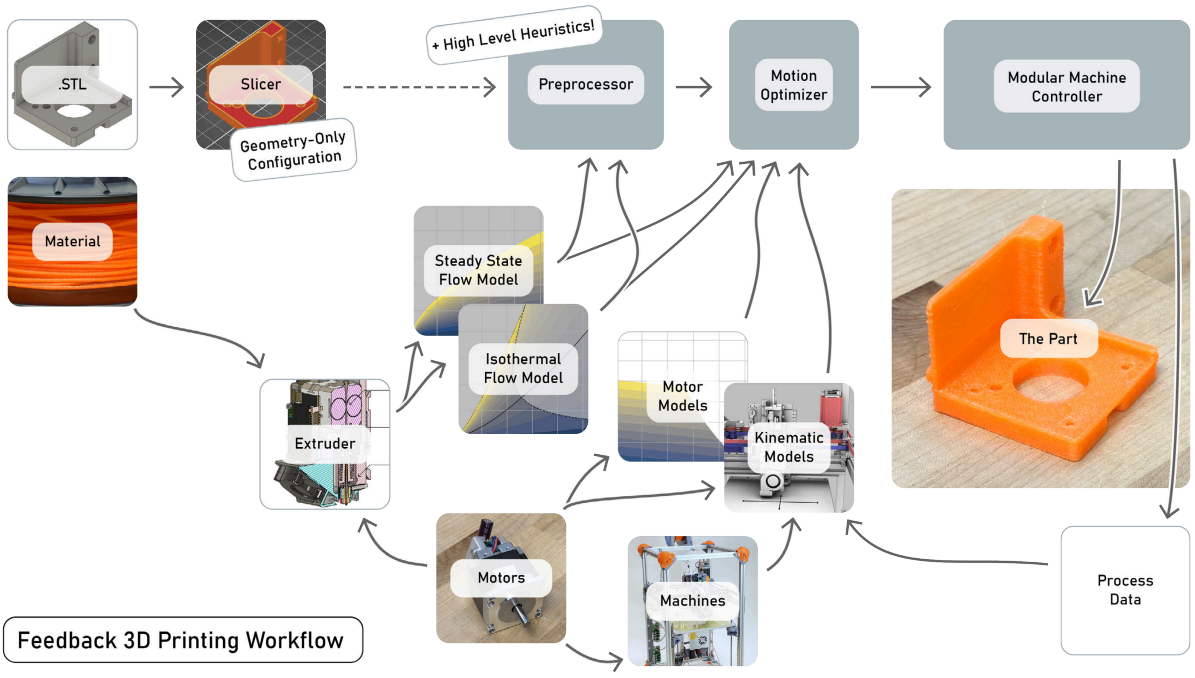


Figure 5.9: The workflow that I develop and evaluate in this chapter, a description of which is in the surrounding text. Compare to the state-of-the-art workflow in Figure 5.4.

Combining the methods in this chapter, I developed a complete workflow for FFF that replaces feedforward machine configuration with feedback configurations using models. In overview, the workflow operates in two steps: machines make models using the materials and nozzles that they will use to print, and we then combine those with heuristics to operate the machine. Figure 5.9 below shows this process in more detail.

I do not take on the task of generating path *geometry,* instead focussing on the parts of the problem that combine motion and flow optimizations, i.e. speed and temperature selections.

Like the state-of-the-art workflow, this one takes in a part geometry as an `.stl`, and produces a fabricated part. It also uses an off-the-shelf slicer to generate *path geometry* from the part, i.e. to decompose the `.stl` into a series of tracks. In addition to the fabricated part, this workflow also produces a *process data* file, which contains time-series outputs from the machine's sensors and the motion controller, alongside models that were used during the job, the input geometry and other metadata. This adds new value that I discuss later in the chapter, the most exciting of which is an ability to computationally identify errors and the opportunity to continuously improve models.

The parameters from the state-of-the-art workflow that control the motion and extrusion of the machine (i.e. from Table 5.2 and 5.3) are replaced with *models* of those aspects of the hardware. The extruder generates flow models, and those are used to select flowrates *within* the systems' real

physical boundaries. I do this with a preprocessor, where models are combined with heuristics (see Section 5.8) and with a cooling model, which allows for time-optimal nozzle temperature selection.

The machine itself is used to fit kinematic models to describe the motion system, the primary tool for which are the motors themselves — a process that I described in Section 4.5. These are combined with flow models to extend our motion planner, which then runs the machine in realtime such that speed is maximized while physical constraints and limits set in the preprocessor are respected.

Once models for flow and motion are fit (each takes tens of minutes to start, and can be re-used for subsequent prints using the same material, nozzle, and machine), the workflow to produce a new part follows these steps:

1. The desired **part file** is turned into a **path file**, where we can select layer height, infill densities and patterns, and shell thicknesses. This is a static path encoding that contains a series of path segments and widths, each segment of which is binned into one of five categories: *infill, perimeter, small, outer, and travel* moves.
2. The **preprocessor** (Section 5.8) is configured with material flow and cooling models, as well as user heuristics for motion, flow and cooling, and some estimates (see Table 5.5). It selects an optimal nozzle temperature for the combination and generates a **run file,** which is an annotated version of the *path file* whose data are described in Table 5.6.
3. The run file is passed to the **motion planner,** (Section 5.9) along with motor, motion, and flow models. This controls the machine in realtime — maximizing speed while respecting constraints laid out in the models and in the run file. It is connected to the machine via OSAP (Section 2.3), PIPES (Section 2.4) and MAXL (Section 2.5).
4. During the print, time series data are collected from the machine and it's instruments and stored in a **process data file.** This data can be used to improve models, detect errors, or analyze machine performance.

At first glance it might look like we replaced one set of parameters (Table 5.1, 5.3, 5.2) with another (Table 5.5), but in our workflow the heuristics are *mediated* by models: flowrates are set by targeting an *allowable pressure* that is *scaled by the flow model,* and motion (speed and acceleration together) is scaled *via the motion model,* by specifying how much current the machine should be allowed to deploy for different parts of the path. Whereas the state-of-the-art workflow requires us to re-tune parameters for any change to the machine, material, or nozzle, here we can instead modify the underlying models and cast the same set of heuristics through them: the preprocessor and planner work together to adjust real-time control outputs accordingly.

Manual Input	Units	Typ. Value	Mediating Models	Destination
Cooling Estimates	—	—	—	—
Chamber Temp	$^\circ$ C	30.0	Cooling	Preprocessor
Cooling Target ΔT	$^\circ$ C	-20.0	Cooling	Preprocessor
h_{air} est.		5.0	Cooling	Preprocessor
κ_{fil} est.		0.1	Cooling	Preprocessor
Direct Heuristics	—	—	—	—
Minimum Flow	mm^3/s	15.0	Flow	Preprocessor
First Layer Rate	mm/s	50.0		Both
Pressure Scalars	—	—	—	—
Infill	$0-1.0$	0.75	Flow	Both
Perimeters	$0-1.0$	0.65	Flow	Both
Details	$0-1.0$	0.45	Flow	Both
Current Scalars	—	—	—	—
XYZ (fast)	$0-1.0$	0.20	Motion	Planner
XYZ (fine)	$0-1.0$	0.25	Motion	Planner
E (fast)	$0-1.0$	0.75	Motion, Flow	Planner
E (fine)	$0-1.0$	0.60	Motion, Flow	Planner
Current Slew Rate Scalars	—	—	—	—
XYZ (fast)	$0-1.0$	0.125	Motion	Planner
XYZ (fine)	$0-1.0$	0.110	Motion	Planner
E (fast)	$0-1.0$	0.50	Motion, Flow	Planner
E (fine)	$0-1.0$	0.50	Motion, Flow	Planner
Extruder Heuristics	—	—	—	—
Pressure Offset	N	15.0		Planner
Reset Balance	ratio	0.80		Planner
E Rate Limit	mm^3/s	75.0		Planner
E Acceleration Limit	mm^3/s^2	40000.0		Planner

Table 5.5: A table of the heuristic inputs that are delivered to the feedback-based FFF workflow's *preprocessor*.

Run File Content	Units	Notes
Per-Segment Variables	—	
Max Overall Rate	mm/s	Enforced for part cooling and flow heuristics.
Max XYZ Current Deployment	$0 - 1.0$	
Max E Current Deployment	$0 - 1.0$	
Max XYZ Current Slew Rate	$0 - 1.0$	
Max E Current Slew Rate	$0 - 1.0$	
Fixed Variables		
Nozzle Temperature Setpoint	$^\circ$ C	
Retract Target Pressure	N	
Retract Reset Asymmetry	$0.0 - 1.0$	

Table 5.6: A table of the contents of the feedback-based FFF workflow's *run file*.

5.4 Hardware for Feedback-Based FFF

In order to develop and deploy this new workflow, we are going to need some hardware. In this section I will explain the two machines I developed for this chapter: the first was my development mule, the second is a minimally modified off-the-shelf machine.

5.4.1 The RheoPrinter

The *RheoPrinter* is a custom-built machine for the purposes of this project. The hardware is representative of most commercially available FFF 3D printers, except for the addition of instruments in the hotend, as detailed in the figure above.

I originally included a filament sensor on the machine to detect changes to filament width and slip between the extruder drive gears and filament, based on a design that was developed by Thomas Sanladerer [325]. Use of this instrument is not outlined in this thesis; it became mostly redundant once the extruder motor models were sufficiently well-developed.

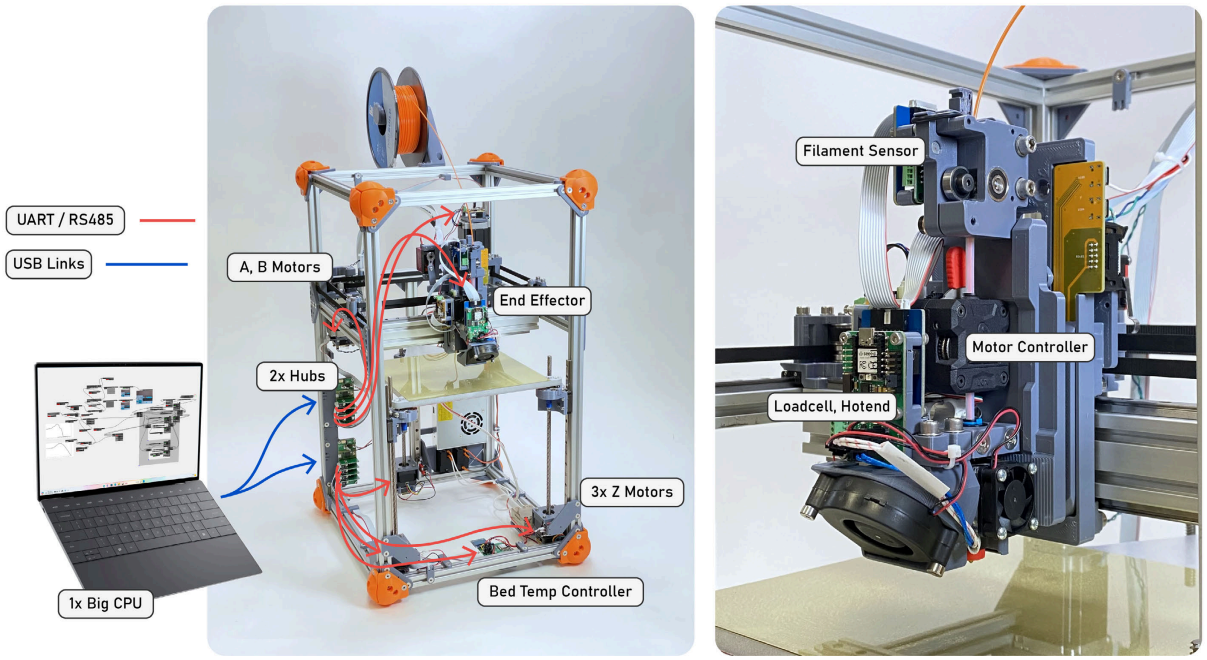


Figure 5.10: The RheoPrinter is an FFF 3D printer that deploys the controllers I have developed in this thesis, and includes instrumentation for flow- and motion model building. Here I roughly diagram the controller's physical layout.

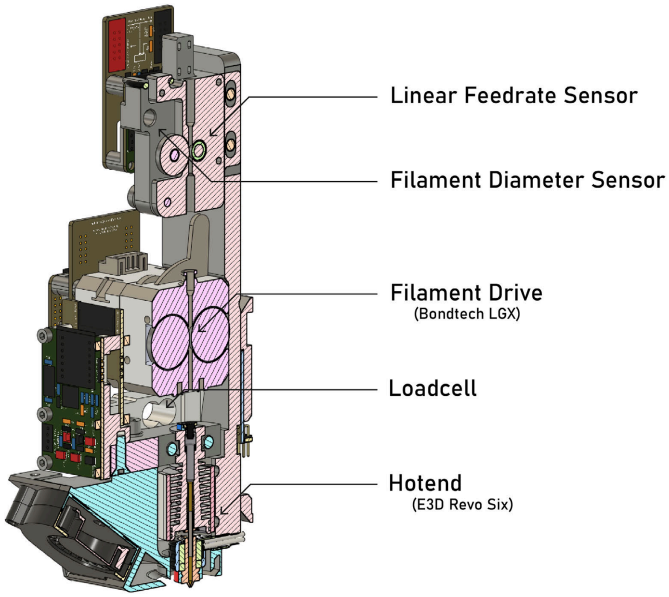


Figure 5.11: The RheoPrinter's hotend features a load cell between the hotend and extruder motor (which allows us to measure nozzle pressure), and a filament sensor that measures linear flowrate as well as filament thickness.

5.4.2 The FrankenPrusa

While the hardware on the RheoPrinter has a similar mechanical *architecture* to state-of-the-art printers, the particular implementation is of course different than any off-the-shelf printers available. Motors are larger with more rotor inertia, axes are heavier, and of course the friction values are different. It also uses an extruder and hotend combination that (while commercially available) is not implemented in any particular off-the-shelf machine.

In the evaluation of this printing workflow we are interested in comparing just this control system with the state-of-the-art. To do this without having to dissect which differences arise from actual hardware limits, I retrofit a *Prusa Core One* [149] with my own control boards (Chapter 3).

This involved fitting motor and kinematic models of the printer using the workflow I described in Section 4.3.1 and Section 4.3.2, and then fitting flow models using techniques described in this section. The lynch-pin for me here is that the Prusa print head (dubbed the *Nextruder* [326]) implements a load cell into its design - primarily used by them for bed leveling - this is not available on *all* SOTA printers, but it is the most important flow modeling tool in our kit.

Besides the load cell, I also added a microbolometer to the Prusa, since time series spatial thermal data is invaluable in print analysis (but not required for the work described here). All other sensing is done either with thermistors or with motor controller current measurements. For more detail on the retrofit process itself, see Section 3.6.2.

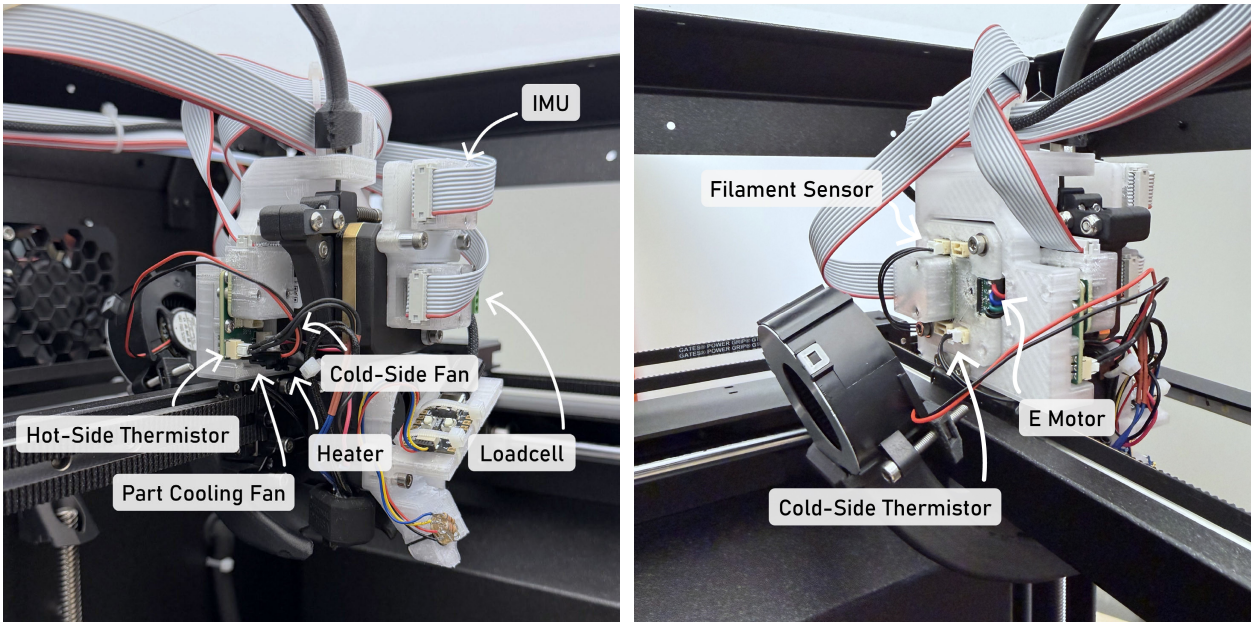


Figure 5.12: The retrofit Prusa Printhead with labelled control circuits.

In Table 5.7 I tally the values and data rates that are returned from this set of hardware as the machine operates.

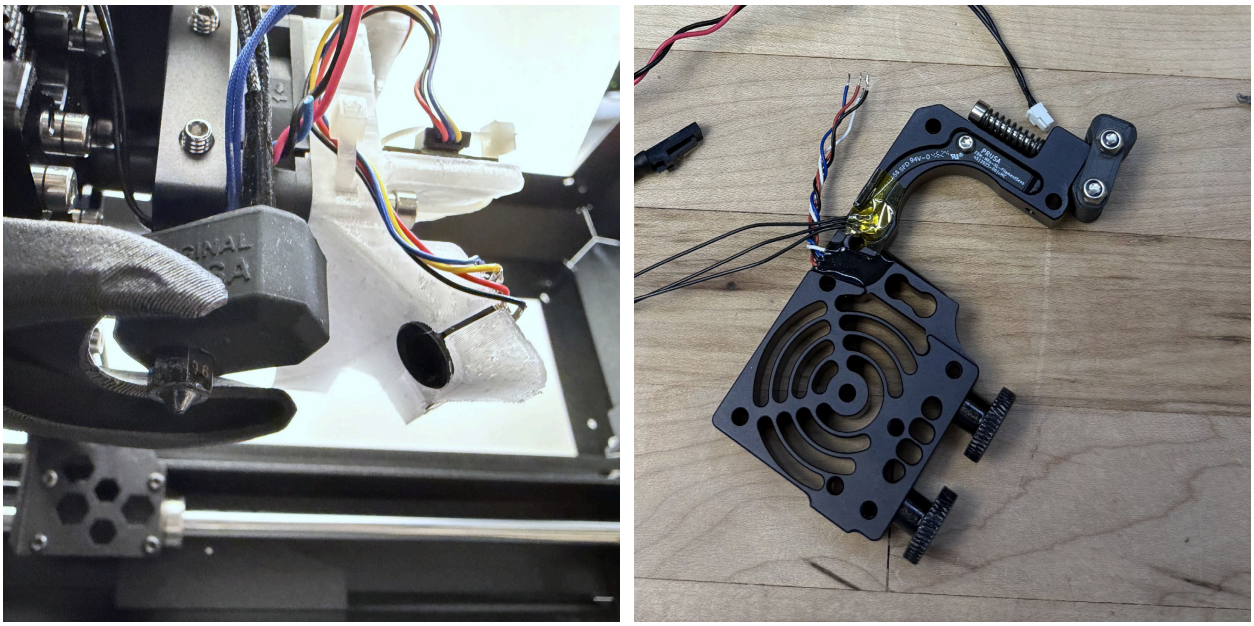


Figure 5.13: Left: A small microbolometer (the black cylinder at right, pointed at the nozzle) was added to record parts' thermal histories. The nozzle (middle) is surrounded by the print head's part cooling fan (PCF) shroud (left). At right: the Prusa extruder's integrated load cell (the small dog bone shaped cutout in the center of this part).

Device, Output	Units	Rate
Extruder Motor		
Velocity Target	$rads/s$	500 Hz
Velocity Measurement	$rads/s$	500 Hz
Motor Current	$Amps$	500 Hz
Loadcell		
Calibrated Force	N	250 Hz
H-Bridge Module		
Nozzle Temperature	$^\circ$ C	50 Hz
Nozzle Power Output	W	50 Hz
Microbolometer		
Pixel Values (24x32)	$^\circ$ C	16 Hz

Table 5.7: Data returned from the FrankenPrusa's hotend. Each measurement is sent alongside a timestamp. Here I include motor states that are most relevant to printing, the full list of values sent by the motor is in Table 4.3.

5.5 Models for FFF Flow

In this section I develop a series of models and model fitting routines for FFF melt flow operation. This includes steady state models in Section 5.5.2 and isothermal / dynamic models in Section 5.5.3. In practice flow is neither steady-state or isothermal, and so I develop a strategy to interpolate between these two models in Section 5.5.4.

This is an important set of methods in our work; models must have enough fidelity to capture the important physics (some of which are time-varying and all of which are nonlinear) and should be simple enough to fit well on the available data *and* to run at real-time in the machine's motion controller.

A key difference between these models and others from Section 5.2.4.1 is that these collapse material, nozzle, and machine physics into one set of models, i.e. they work to predict volumetric flow from the nozzle tip based on extruder motor inputs using the load cell's pressure measurement. Some internal parameters reflect real physics (i.e. the filament spring rate is measured in N/mm^3) and the inputs and outputs are measured in real physical values (torques, forces, and volumetric flowrates), but the models are not focussed on understanding *rheological properties* directly (i.e. material viscosities and shear rates), rather they try to predict how the system can be operated.

The reason for doing so is twofold: firstly, if we build models of our machine *with* our machine, we can easily turn around and use those models in control. The second is that *because* our ultimate goal here is to control the system, it is actually easier to model the system as a whole rather than as constituent components. Although these models do break apart into a few pieces (motor models, isothermal flow models and steady-state flow models), overall we limit the number of free parameters to fit by collapsing (some) internal states.

Models are evaluated in Section 5.10.1 by reporting model errors for real world prints (comparing the planner's pressure predictions to measured nozzle pressures), and in more detail in Section 5.10.6.

5.5.1 How Flows are Modelled in This Work

Modelling polymer flows for 3D printing was perhaps the most confounding task that I took on in this thesis, and the most surprising. Like with any good problem, each time I thought that I had understood it, another layer of complexity would reveal itself.

In building predictive flow models, we are trying to understand what flowrate will emerge from the tip of the nozzle at any given time (I will use Q_{out} for this value, in mm^3/s). The value depends on a handful of inputs, which I enumerate below.

1. The polymer has a viscosity that is dependent on it's temperature and shear rate (i.e. nozzle pressure) - polymers actually become *less* viscous when they are under pressure (this is called *shear thinning*).

2. The filament between the extruder's drive gears and the nozzle is compressible, and so we must build pressure before extrusion begins and measure the stiffness of the material (that stiffness also changes with temperature).
3. The extruder motor has its own dynamics that must be coupled into the flow system.
4. The thermal history of the filament in the melt zone needs be considered: the melt flow temperature T_{melt} takes time to come up to the nozzle's temperature T_{noz}. At high extrusion speeds the melt flow does not reach this temperature before it is extruded, leading to colder flows where we need more pressures to achieve the same flowrate.

This means that there are a number of *internal states* between the inputs to the system that we can *drive* (the nozzle temperature T_{noz} and the extruder motor's torque τ_e) and the *outflow* Q_{out} that we would like to predict and ultimately control. At an overview, the system has three core steps, that I try to diagram below.

1. The motor model (which is coupled to filament dynamics) tells us what inflow (flowrate at the *top* of the melt zone Q_{in}) will be generated over time, given torques τ_e.
2. Q_{in} generates pressure in the nozzle F while Q_{out} relieves pressure in the nozzle.
3. The nozzle pressure *and* the melt flow temperature can predict the actual outflow: $Q_{\text{out}} = f_{\text{flow}}(F, T_{\text{melt}})$
4. The *history* of flow velocity in the melt zone is used to predict the actual melt flow temperature T_{melt}

Below is a simplified diagram of these components.

$$\begin{aligned} \tau_e \rightarrow f_1(Q_{in}, \dot{Q}_{in}, F) \rightarrow Q_{in} \rightarrow f_2(Q_{in}, Q_{\text{out}}, T_{\text{melt}}) \rightarrow F \rightarrow f_3(F, T_{\text{melt}}) \rightarrow Q_{\text{out}} \\ T_{noz} \rightarrow f_4(Q_{\text{out}}) \rightarrow T_{\text{melt}} \end{aligned} \tag{5.1}$$

There are a number of interactions between these model components: nozzle pressure and inflow (and acceleration) affect the motor model, inflow, outflow and the *melt* temperature change the pressure (warmer flows are softer), and the melt temperature - the most difficult to predict - both predicts flow *and* is a function of flow history. Make the pressure measurement F directly is key, because it lies between the *pressure generating* half of these flow models, and the *flow generating* half.

The motor models were explained in Section 4.3.1; in 5.6 I show how those are connected to the extruder system.

Filament compressibility is modelled using a spring rate k_{sq}, which is interpolated across temperatures. The ODE that describes this component is below: change in nozzle pressure is the change in the volume of filament between the drive gears and the nozzle times our spring rate; the spring rate has units N/mm^3.

$$\dot{F} = (Q_{in} - Q_{\text{out}})k_{sq} \tag{5.2}$$

To model flowrates as a function of nozzle pressure I use a generic power function, with a linear parameter and power parameter: those two are also interpolated across temperatures.

$$Q = (F \cdot k_{lin})^{k_{pow}} \tag{5.3}$$

I fit *two* models for flow: behaviour in steady-state (Q_{ss}, where behaviour is dominated by thermodynamics) and in isothermal conditions (Q_{iso}, where it is dominated by the actual underlying rheology, see Section 5.5.2 and 5.5.3 for more details and fitting routines). Both take on the same shape as in the equation above, but with different parameters.

The most difficult step is in estimating T_{melt}, for that I interpolate between the two flow models - see Section 5.5.4.

I use these models to make print-scale planning choices (for nozzle temperature) in Section 5.8 (in combination with cooling models, 5.7), and in realtime machine control in Section 5.9.

5.5.2 Fitting Steady-State Flow

Given that we start with little or no information about our system (besides the motor model), our first modelling challenge is to find the maximum flowrate that can be achieved at any given temperature. This sets an initial boundary for operation within which our tests for dynamic parameters can be performed without fear that we will operate the machine in a failure mode; we want our tests to be non-destructive, so that we can proceed without having to manually reload or reset filament, etc.

I developed a simple open-loop test for this that is quick, repeatable, and provides us with enough information to determine a few key parameters: (1) a linear fit for maximum flowrate across temperatures and (2) a nonlinear fit that gives us the steady-state flowrate at any nozzle pressure and temperature. The data from this test is also used in Section 5.7.1 to estimate the filament's volumetric heat capacity, which can be used in conjunction with simple cooling models to generate minimum layer times (more on those later).

The test heats the nozzle to its maximum temperature, *turns the nozzle heater off,* and then drives the extruder motor at a set of constant rates while measuring all of our system states. The nozzle temperature decays naturally, pressure rises, and the test stops when the extruder motor's velocity controller fails to maintain the target rate, i.e. maximum torque cannot sustain flow. To avoid shearing brittle materials, the test can be configured to limit extruder torque.

Listing 5.1: The test procedure that I use to generate melt flow steady state data.

```
1. Heat the nozzle to its maximum temperature.
2. Set the extruder's velocity controller setpoint to our test rate.
3. Turn the nozzle heater off.
4. While the velocity controller's error is below some threshold:
  - Collect time series data for:
    - Loadcell Force
    - Extruder Motion States
    - Nozzle Temperature
    - IR Frames
5. Restart the test at the next flowrate.
```

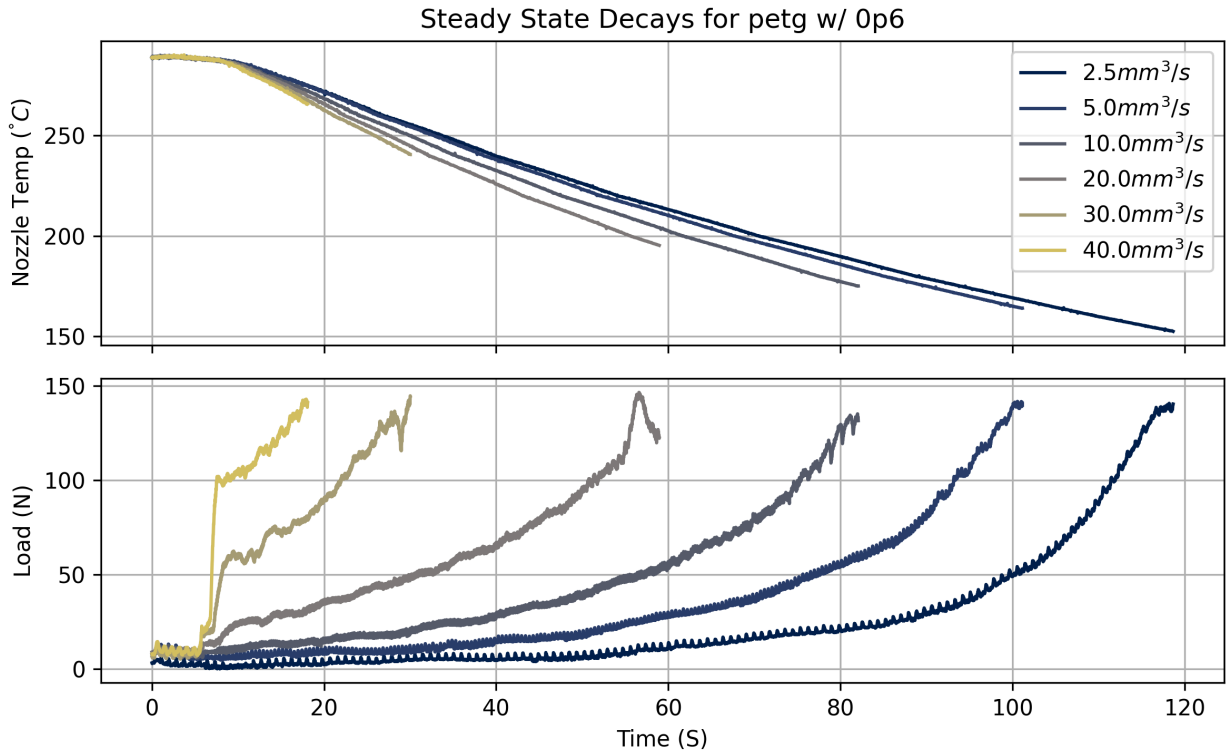


Figure 5.14: Time series data from the steady-state decay test, generated with the routine described in Listing 5.1. In this plot we see six flowrates (colour coded). Nozzle pressure (lower plot) rises to some power as the temperature (top plot) drops over the course of the test. We can also see that faster flowrates decay nozzle temperature faster, which we will use later on to characterize the materials' volumetric heat capacity.

After this test, I first fit a linear model for the minimum flow temperature across flowrates. The zero crossing of this linear fit is an initial estimate of the nozzle temperature where we might expect *any* flow to be possible T_{min} - which we will use to normalize temperature ranges in some of the next steps' fits.

As I described above in 5.5.1, we can fit a power function to the relationship for flowrate, given nozzle pressure and temperature:

$$Q_{ss} = f_{ss}(F, T_{noz}) = (F \cdot k_{lin})^{k_{pow}} \tag{5.4}$$

Where k_{lin}, k_{pow} vary across temperature, according to:

$$\begin{aligned} T_{norm} &= \frac{T - T_{\text{min}}}{T_{\text{max}} - T_{\text{min}}} \\ k_{ss,lin} &= T_{norm} \cdot a + b \\ k_{ss,pow} &= T_{norm} \cdot c + d \end{aligned} \tag{5.5}$$

I fit these four parameters a, b, c, d alongside T_{min} (which is initialized using the estimate generated in Figure 5.15) with scipy's `minimize` routine [175] using the L-BFGS-B solver [327]. Here, I can fit the entire state space at once (rather than in time-series) because there is no time dependence

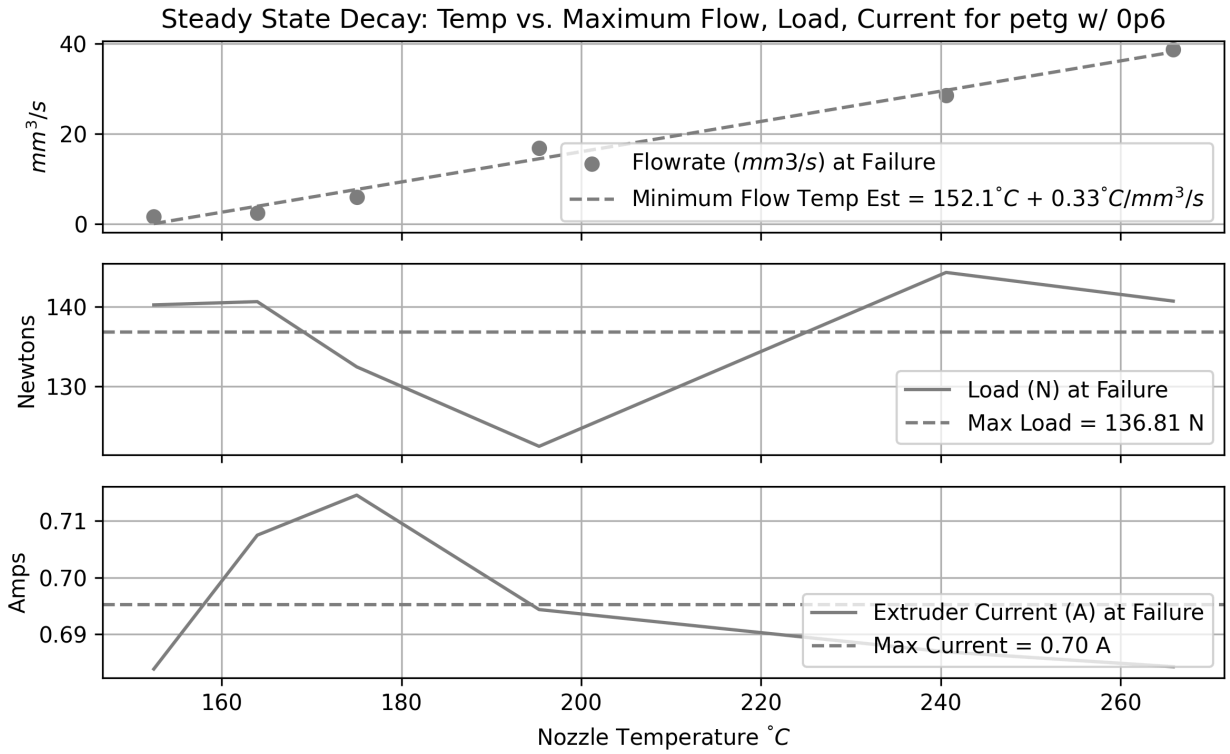


Figure 5.15: Estimates for preliminary extruder limits, calculated from data in Figure 5.14.

in the steady-state system. This function nets us the contour plot below, which maps the steady-state characteristics of a particular material *and* nozzle.

To re-iterate, this is not exactly a model of the material or nozzle - i.e. we cannot extract the material's *viscosity* from the model - instead it models the system as a whole. I call it a *map* because it shows us where it is possible to operate our machine in the space of flows, temperatures, and pressures.

5.5.3 Fitting Isothermal Flow and Compression

The steadystate model helps us to set operating boundaries for our machine, but it does not properly model operation of the system during short time intervals. In practice, most FFF printing is dynamic: machines are constantly accelerating and decelerating and flow targets are correspondingly increasing and decreasing such that track widths remain consistent.

The isothermal pressure to flow relationship itself can be modelled with a similar power function to the one we saw in the section above, but to a different set of parameters, and now using the *melt flow temperature* T_{melt} rather than the nozzle temperature. In isothermal operation, we also have to consider the filament's compression dynamics, which are modelled (as discussed) by the ODE in the second row of Equation (5.6) below.

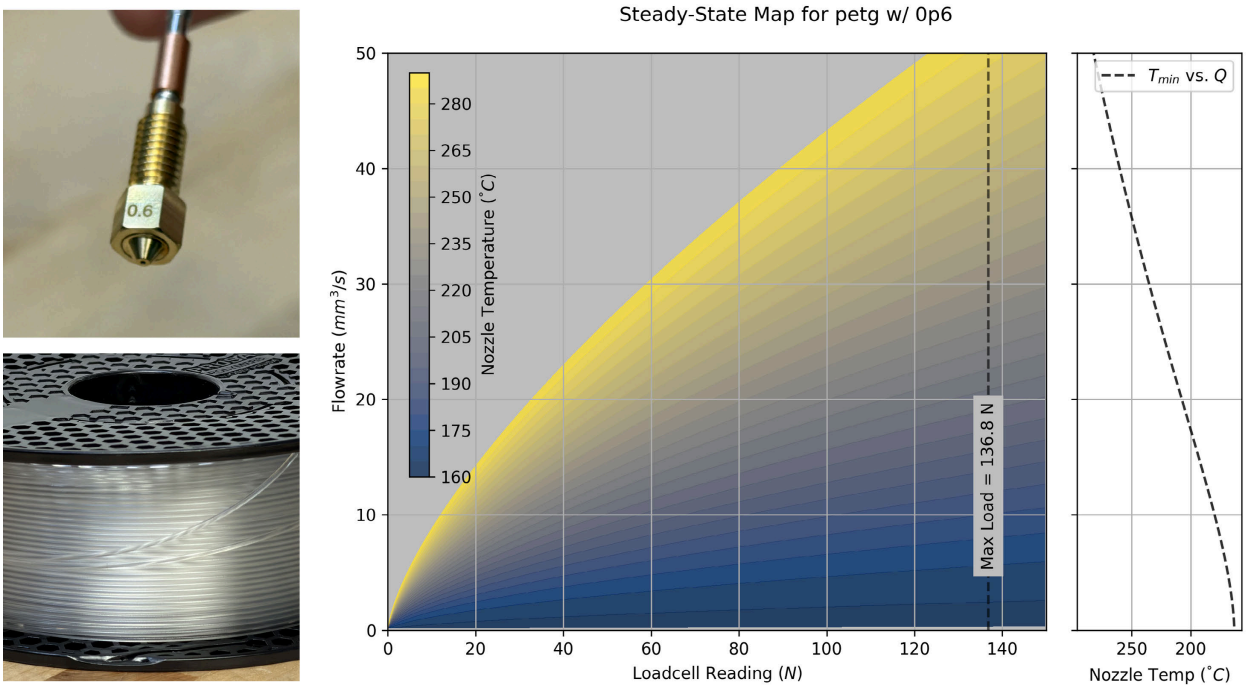


Figure 5.16: This contour plot (right, nozzle and material that it represents at left) predicts the steady-state flowrate (y axis) for any extrusion force (x) and nozzle temperature (z contour). The slice through maximum extruder load (at right) provides an operating boundary of flowrates (y) for any nozzle temperature (x).

$$\begin{aligned} Q_{iso} &= f_{iso}(F, T_{\text{melt}}) = (F \cdot k_{lin})^{k_{pow}} \\ \dot{F} &= (Q_{in} - Q_{\text{out}})k_{sq} \end{aligned} \tag{5.6}$$

Again, k_{lin}, k_{pow} vary across temperature, as does the spring rate, according to another set of parameters:

$$\begin{aligned} T_{norm} &= \frac{T - T_{\text{min}}}{T_{\text{max}} - T_{\text{min}}} \\ k_{sq} &= T_{norm} \cdot e + f \\ k_{iso,lin} &= T_{norm} \cdot g + h \\ k_{iso,pow} &= T_{norm} \cdot l + m \end{aligned} \tag{5.7}$$

To generate data for *these* fits, we have the problem that we cannot measure T_{melt} (which deviates from nozzle temperature after long periods of high flowrates) directly. To approximate isothermal operation, I run a quick *chirp* test (oscillating the extruder motor's velocity) through the system, after allowing filament to dwell in the nozzle for thirty seconds. This mostly avoids generating data whose phenomenology is dependent on filament heating. The chirps are operated at a predetermined set of temperatures (a simple range of 20° C intervals from $T_{\text{min}} \rightarrow T_{\text{max}}$). Chirp amplitudes are chosen as a proportion of the steady-state maximum flowrates possible at those temperatures, and their frequencies and duration are fixed via a test configuration.

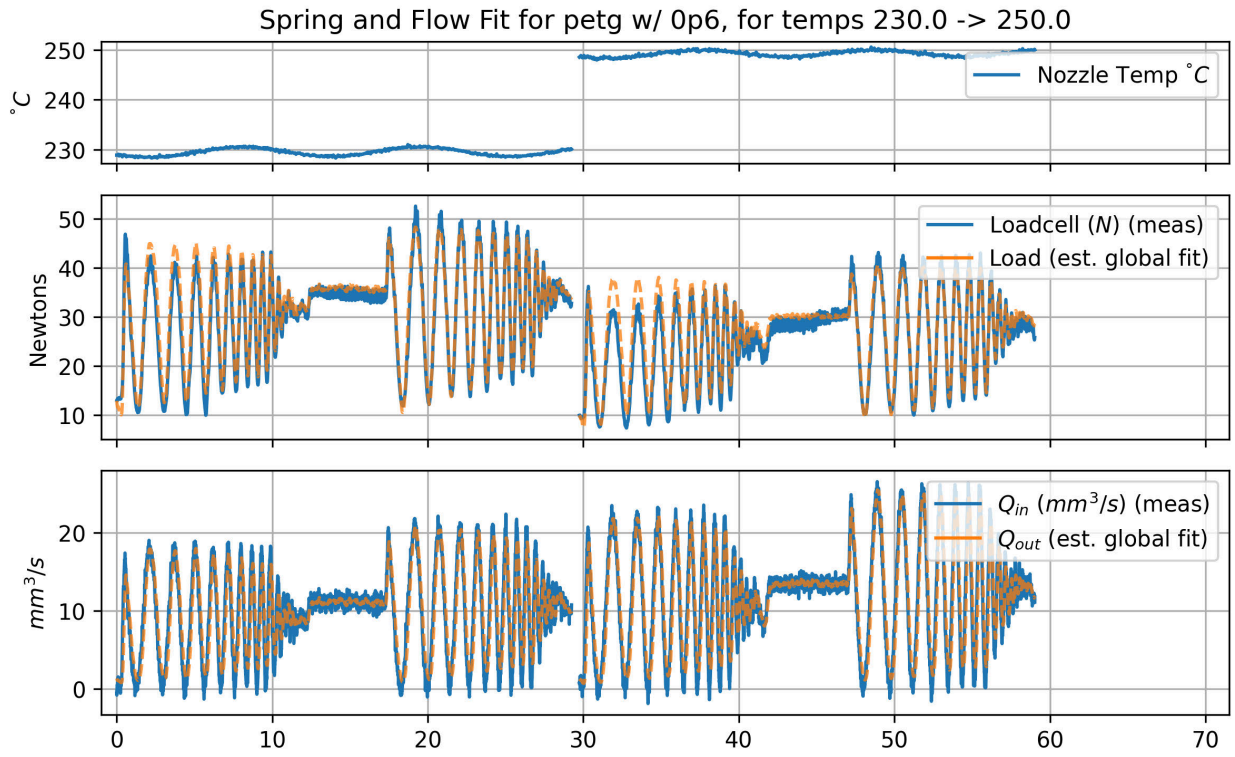


Figure 5.17: This plot shows time series data from one of the chirp tests used to fit parameters for the flow dynamics ODEs in Equation (5.6) and (5.7)

Because the filament compression is time-dependent, I fit these six parameters e, f, g, h, l, m (now using the T_{min} fit on the steady-state test) against time-series data using a simulation of the system that is driven with the same inputs as were recorded in the test routine. A simulated system is rendered below against time-series data.

The cost function for the fit (again using scipy's `minimize` utility with the L-BFGS-B solver) compares time-series of the simulated system's internal *pressure* state against the measurements. This modelling step is a key differentiator between this work and other modelling efforts, where direct measurements of Q_{out} are required. Those are difficult to make, as I discussed in Section 5.2.3.3. Using the internal pressure state is possible if we can trust the surrounding models; a key insight in this case is that the pressure F is related *both* to Q_{in} (which increases pressure, and we can measure directly) and to Q_{out} (which decreases pressure, but we cannot measure) - for our model to be consistent, it needs to explain pressure generation *and* decay.

Finally, we can generate an *isothermal map* of the same operating space (pressure, flow, temperature). Again, we make one of these maps for any material and nozzle combination.

5.5.4 Interpolating Between Steady-State and Isothermal Operation

The tests developed here are designed to provide explicitly steady-state and isothermal data separately. In practice, the machine operates somewhere between these two ranges: long, fast

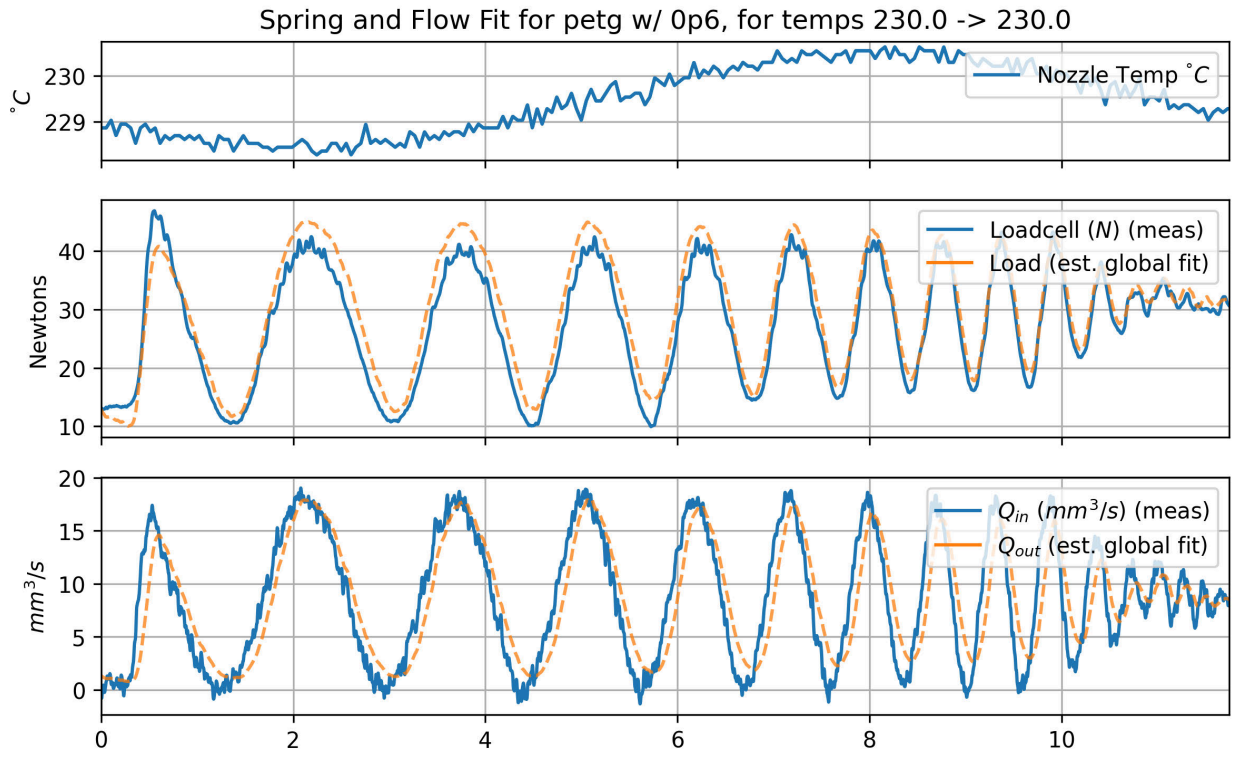


Figure 5.18: A time zoom of Figure 5.17. Looking closer at the fit and data, we can see that the outflow prediction Q_{out} lags the inflow Q_{in} due to the compression dynamics.

sections of infill might get us close to steady-state operation, whereas slow, detailed sections of a print are close to isothermal.

Modelling this properly as a history dependent thermodynamic system would be difficult because it would add a few new internal parameters like filament conductivity, heat capacity, and might require that we accurately estimate the nozzle heater's thermal mass, etc. I discuss the possibility of taking this on in Section 5.12.1, but in this cycle I take the approach of interpolating between steady-state and isothermal models using a measurement of the average flowrate *of the filament in the melt zone.*

First, let's look at both models, Figure 5.20 shows steady-state and isothermal maps for PETG using a 0.6mm brass nozzle.

In the steady-state model, as pressure increases, steadystate flow increases but *at a decaying rate* - i.e. for every additional N of load, we get less return on flow ($k_{pow} < 1.0$). However, on shorter time-scales, this is reversed: flow increases *at an increasing rate* for every additional N of load ($k_{pow} > 1.0$).

The decaying power in steadystate can be understood as the melt flow's temperature dropping at larger rates (thus increasing its viscosity, and increasing the force required to extrude). On short time scales, we see polymer shear thinning, a phenomenon observed in non-newtonian fluids where increases to pressure (… shear) *drop* the flow's viscosity [306].

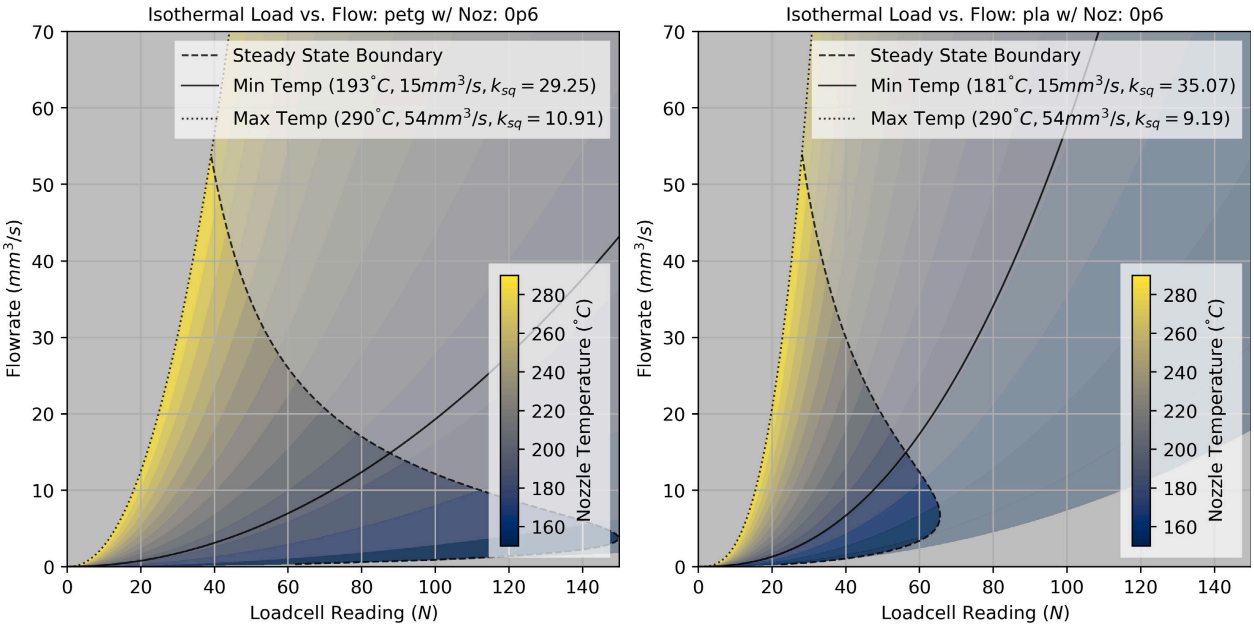


Figure 5.19: These plots render two flow models, which relate pressure (along the x axis, N) to flowrate (y, mm^3/s). These are complete models that span a range of temperatures (color mapped), and show three operating boundaries: maximum temperature (a hard limit related to hotend material choices), the minimum temperature (taken to be the temperature where flows less than $15mm^3/s$ are impossible) and a *steady state boundary,* which is the same boundary as presented at the right side of Figure 5.16 (it represents maximum continuous flowrate at that temperature). spring rates are also labelled at minimum and maximum temperature. Both models were taken on the FrankenPrusa, using the same 0.6mm diameter nozzle. PETG is on the left, with PLA on the right.

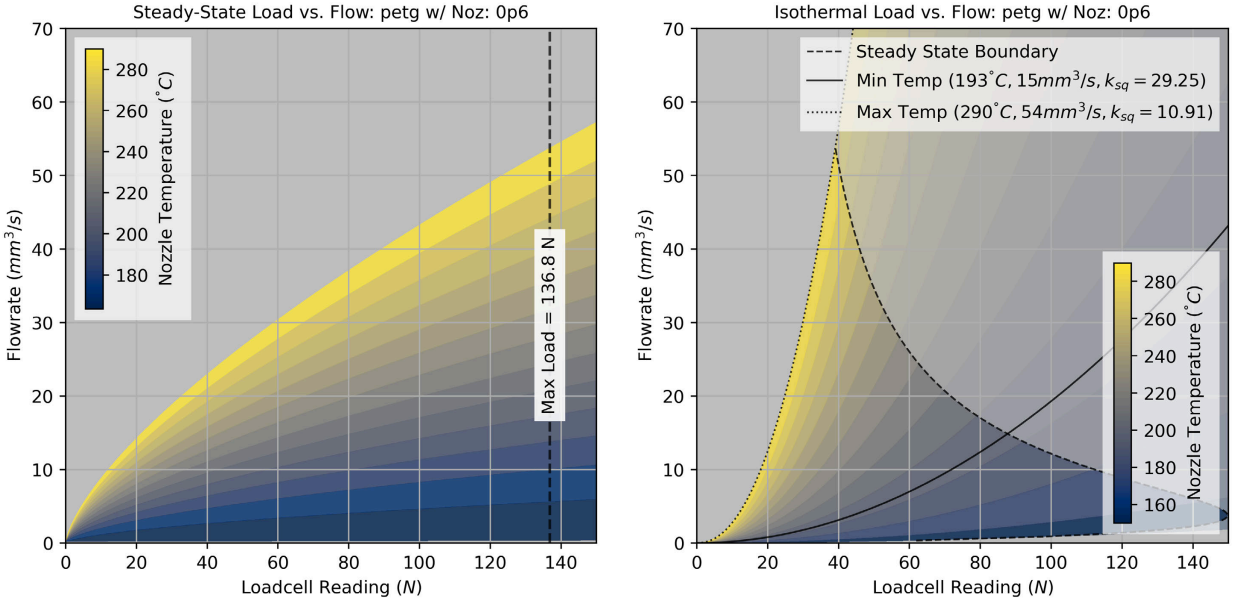


Figure 5.20: The steady-state model for flow in PETG with a 0.6mm brass nozzle (left), and the isothermal model (right).

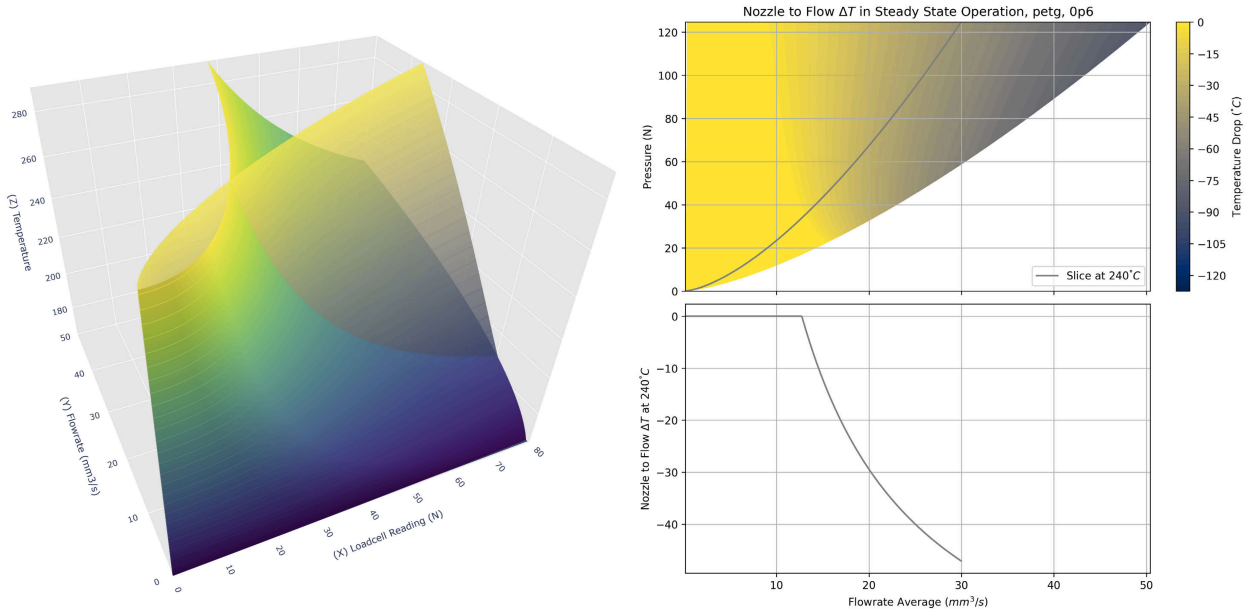


Figure 5.21: At left: steady-state (cividis, faded) and isothermal (viridis, opaque) models overlaid on one another in 3D. At right: the result of subtracting the isothermal temperature from the steady-state temperature across the space of (pressure, flowrate) (above), and taking a slice through a particular nozzle temperature (240° C) (below).

The steady-state and isothermal maps are both temperature contours in the (pressure, flowrate) space. If we pick any point here, we have two temperatures: one from the steady-state model, and one from the isothermal model.

While the two models mostly overlap near the origin, the steady state model predicts larger temperatures than the isothermal model as speeds and pressures increase. If we subtract the isothermal temperature from the steady-state temperature across the operating space, we get a third map that corresponds to *temperature drop* of the melt flow from the nozzle temperature. In that map (in the top right of Figure 5.21), the X axis is the *average flowrate* of the melt flow within the nozzle, the Y axis is the extrusion pressure, and the contour (color-mapped) is the drop between the nozzle temperature and the melt flow temperature.

By taking a slice of this model at the operating nozzle temperature, we can plot the expected temperature drop as a function of average flowrates at that temperature. That's above in Figure 5.21, and is the information we will need later when we deliver these models to the machine controller, which uses this curve to estimate the actual melt flow temperature during operation.

5.6 Models for Extruder Motors

The models above are sufficient to understand the flow dynamics in nozzle. However, we need to drive those dynamics with our extruder motor. I already covered the relevant motor physics (and our modelling strategy for the same) in Section 4.3.1, but for a short reprisal: motor torque τ is

proportional to stator current I. Current generation is limited by the motor's construction, our driver's power electronics, and by motor speed. In this section, I will describe how I couple motor models to flow models using a few additional parameters, and how those parameters are fit to data.

Motor dynamics are typically coupled into motion systems that have fixed mass and damping, and the extruder motor is similar: the filament sliding around between the spool and the motor (and on its way towards the melt zone) produces friction (damping), and it has some mass, as does the motor's own rotor. In the case of the extruder motor, I fit these terms against the volumetric flowrate at the motor, Q_{in}, which is proportional to the extruder motor velocity ($rads/s$) via the extruder's drive ratio (a scalar that describes the volume of filament that is extruded for each motor rotation). This scalar is measured and defined by hand, and used to calculate the motor rotor's actual speed for the electrical half of the motor model.

For flow, we need to add new parameters that describe the amount of current required to generate *pressure* in the nozzle. For this I fit a current offset k_{off} and proportion k_f. The total accounting for extruder current prediction is in Equation (5.8).

$$\begin{aligned} I_{\text{extr}} &= Fk_f + k_{off} + Q_{in}k_d + \dot{Q}_{in}k_a \\ k_f &= \text{Linear Load Factor } (A/N) \\ k_{off} &= \text{Load Offset } (A) \\ k_d &= \text{Damping } (A/mm^3/s) \\ k_a &= \text{Inertial } (A/mm^3/s^2) \end{aligned} \tag{5.8}$$

We have in total three states that are combined to predict extruder current, along with one offset. In Figure 5.22, I plot data points from the isothermal flow chirp test in two spaces: load vs. current and flowrate vs. current, with data point colours mapped to additional states.

Finding the parameters for this model is simple with time series data, where I can simply fit parameters such that the measured states properly predict the measured extruder motor current.

Extruder current is typically dominated by the nozzle load, with a smaller damping parameter and even smaller acceleration parameter. Even though it looks small, the acceleration parameter often becomes the critical limit during dynamic moves where optimal inflow acceleration requests can exceed $50000mm^3/s^2$. Parameters from an example fit are called out in Figure 5.23 below, where I compare time series measurements (in blue) to the model's estimates (in orange).

5.7 Models for FFF Cooling

So! Our models so far explain how our extruder operates as it is printing a new track of filament, but it is also invaluable to be able to estimate how long it will take for that track to cool to a temperature where it is solid enough to print the *next* track on top of it, as I discussed in Section 5.2.1.2. For this task, I will show how the steady-state test data can be used to (approxi-

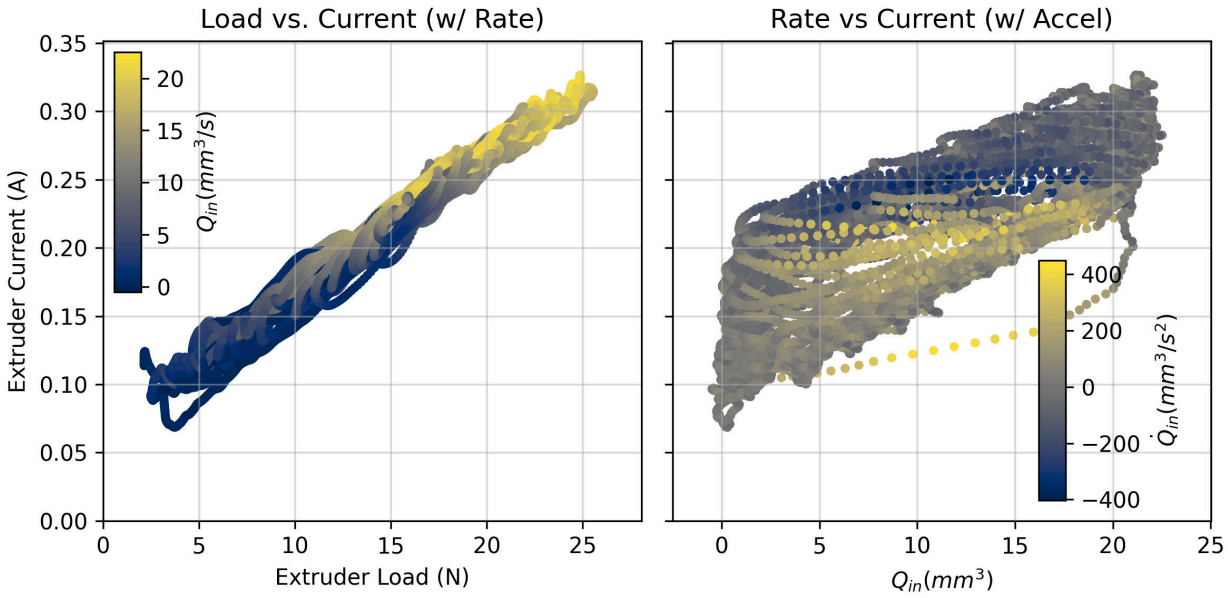


Figure 5.22: Here I plot extruder current (y axis, both plots) against the load cell reading (x-axis, left plot), rate (colormap, left plot, and x-axis right plot) and acceleration (colormap, right plot).

mately) measure our material's volumetric heat capacity, and then develop a simple cooling model using that measurement and some other estimates.

5.7.1 Volumetric Heat Capacity

While it is difficult to develop a complete thermal model of our filament, we can at least estimate its volumetric heat capacity (J/mm^3). We can make this estimate using data from our steady-state tests, where nozzle temperature decays due to losses into ambient environment *and* loss into the filament flow. For example in Figure 5.14, it is clear that nozzle temperature decays much faster at higher flowrates than at lower flowrates. To convert these losses into estimates that have real-world units, we use our knowledge of how much *power* is delivered to the nozzle at any given time, which we can measure with the nozzle's thermal controller.

We begin with a model that estimates the change in nozzle temperature over time Equation (5.9).

Symbol	Units	Value
T	$^\circ$ C	Nozzle Temperature
T_{amb}	$^\circ$ C	Ambient Temperature
Q	mm^3/s	Volumetric Flowrate
k_{loss}	$(C/s) \cdot \Delta T$	Heat Loss to Ambient
k_{flow}	$((C/s)/(mm^3/s)) \cdot \Delta T$	Heat Loss per unit of Flow

Table 5.8: Symbols and units for Equation (5.9).

$$\frac{dT}{dt} = (T_{amb} - T)k_{loss} + Q(T_{amb} - T)k_{\text{flow}} \tag{5.9}$$

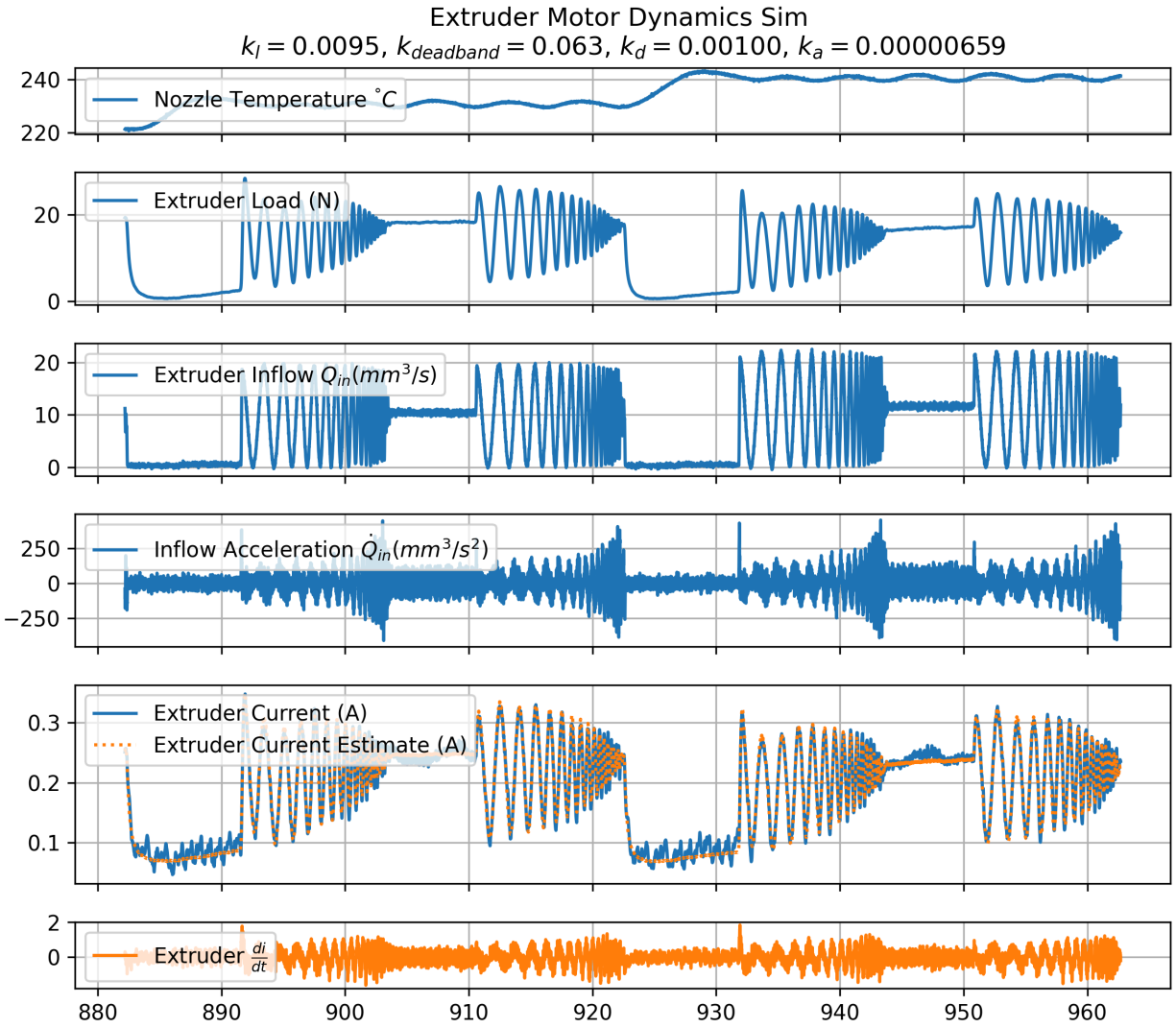


Figure 5.23: This is a time series plot of data used to fit the extruder coupling parameters from Equation (5.8). From the top, we have Nozzle Temperature, Extruder Load, Inflow, Acceleration, and the resulting motor Current, which we use to estimate the Extruder $\dot{I}$. The Extruder Current Estimate (in orange) is calculated using the model described here, after it has been fit using this time series data.

The ΔT we use for each of these losses is from ambient temp T_{amb} to the nozzle temperature T. This makes the assumption that filament heats from ambient up to the nozzle temperature before exiting the nozzle, which we know that is not always entirely true — especially at large flowrates — but it nets us an estimate that is sufficient. To fit, we first estimate the total loss across each flowrate, as in Figure 5.24.

We can then plot the total losses at each flowrate. This fits a simple linear relationship, where we estimate that the offset represents our loss to ambient, and the slope represents additional loss for every unit of filament flowrate.

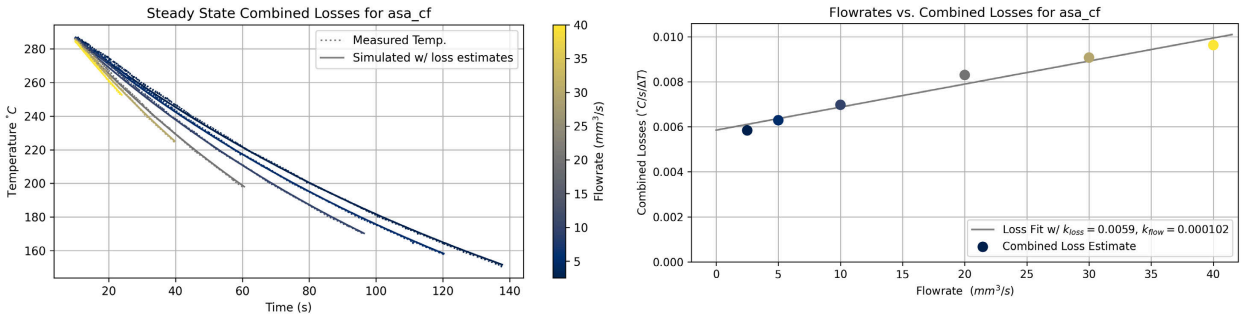


Figure 5.24: Left: the same steady-state data presented in Figure 5.14, now overlaid with simulations using a combined loss. Right: combined loss across rates, and our estimate for independent ambient loss and flowrate-dependent loss parameters.

Symbol	Units	Value
T	$^\circ$ C	Temperature of Flow
T_{amb}	$^\circ$ C	Ambient Temperature
k_{in}	$(C/s)/W$	Change in Temperature per Watt
k_{loss}	$(C/s) \cdot \Delta T$	Heat Loss to Ambient

Table 5.9: Symbols and units for Equation (5.10).

These losses are not yet power measurements. To back out a heat capacity, we need to measure the wattage required to move the nozzle temperature. For this, we can fit an additional parameter that is active when the nozzle is warming up.

$$\frac{dT}{dt} = k_{in} W_{in} + (T_{amb} - T) k_{loss} \tag{5.10}$$

We can measure the power into the nozzle (W_{in}) by measuring our heater's resistance and the controller's supply voltage.

We can now combine our measurements to produce an estimate for the filament's heat capacity. If we do a bit of units analysis (see (5.11)), we can see that the volumetric heat capacity C_v is equal to k_{flow}/k_{in}.

$$\begin{aligned} C_v &= k_{\text{flow}}/k_{in} \\ &= \left(\frac{C/s}{mm^3/s} \cdot \Delta T \right) / (\frac{C/s}{W}) \\ &= \frac{C/s}{mm^3/s} \cdot \Delta T \cdot \frac{J/s}{C/s} \\ &= \frac{J}{mm^3} \cdot \Delta T \end{aligned} \tag{5.11}$$

It is more common to measure *specific* heat capacity of materials (i.e. $J/kg \cdot \Delta T$), but in our case *volumetric* units are more useful since we extrude in terms of mm^3, not by weight, and we anyway don't have an estimate for material density.

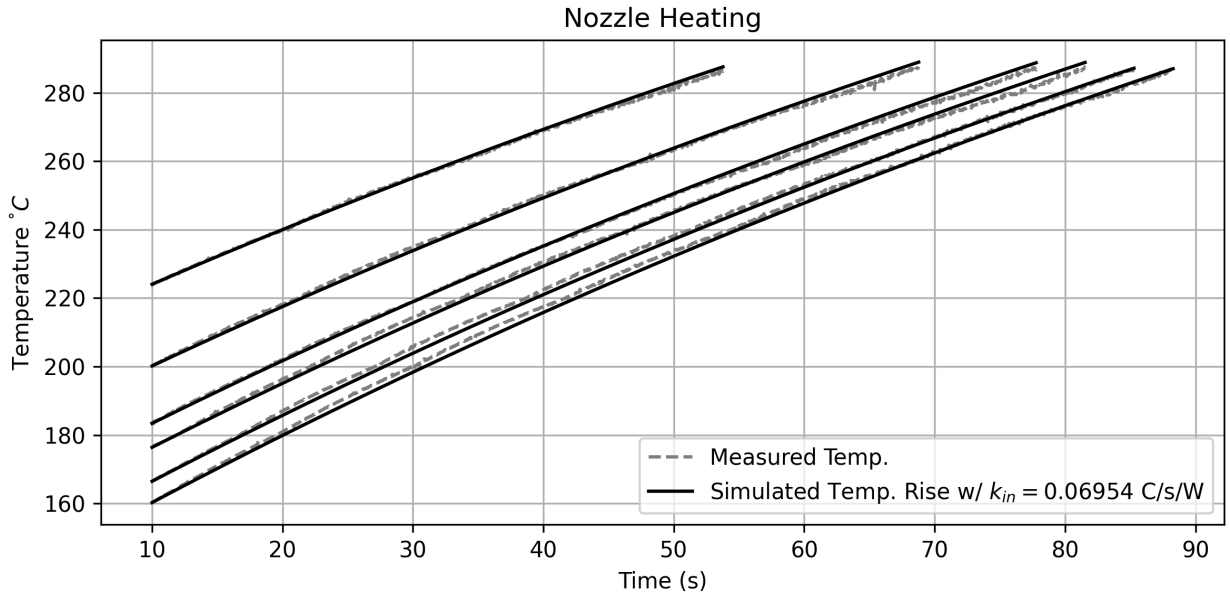


Figure 5.25: Here I plot time series data of the nozzle temperature rise (gray, dashed lines) during heating cycles when we are delivering full power to the nozzle. These data are used to fit k_{in}, which is used with previously fit k_{loss} to generate the simulated temperature rise plotted here in black.

Our measurements are not incredibly precise, but do put us in a reasonable ballpark for the real physical value. Polymer heat capacity is anyway nonlinear in practice, with e.g. different phases of each polymer having different heat-carrying capabilities (not to mention density changes). For our purposes an estimate is good enough.

5.7.2 Layer Cooling Time

Whereas slicers use direct parameters to control part cooling (see Table 5.3 and 5.2.1.2), I replace those with a model that estimates *minimum layer time* ($t_{\min}$, the time that we need to wait after printing a track before we can print on top of it without risk of slumping) for any nozzle temperature.

The rate at which the fan cools the layer is dependent on many things: the polymer's heat capacity, its temperature, the temperature of the surrounding layers, the print geometry, and the fluid dynamics of the cooling air, the chamber temperature, etc. We will build a simple model of this that excludes those fluid dynamics, and also ignores thermal history beyond the previous layer.

For material parameters, we have the filament's volumetric heat capacity (estimated in the section above), but we need also the filament's conductivity κ_{fil}, $(W/m \cdot \Delta T)$. It seems possible to estimate this value (since it factors in the filament's ability to absorb heat from the nozzle, see 5.5.4), but I resort to estimates for this value instead; the typical range for our common polymers is $0.1, 0.25$.

We also need to estimate the effectiveness of our printer's cooling system. This is encoded in h_{air}, $(W/m^2 \cdot \Delta T)$ — energy flow out of the filament into the print chamber. An intuitive sense for this value is that it lies somewhere in the range $10, 100$, and fan-driven cooling rates are relatively

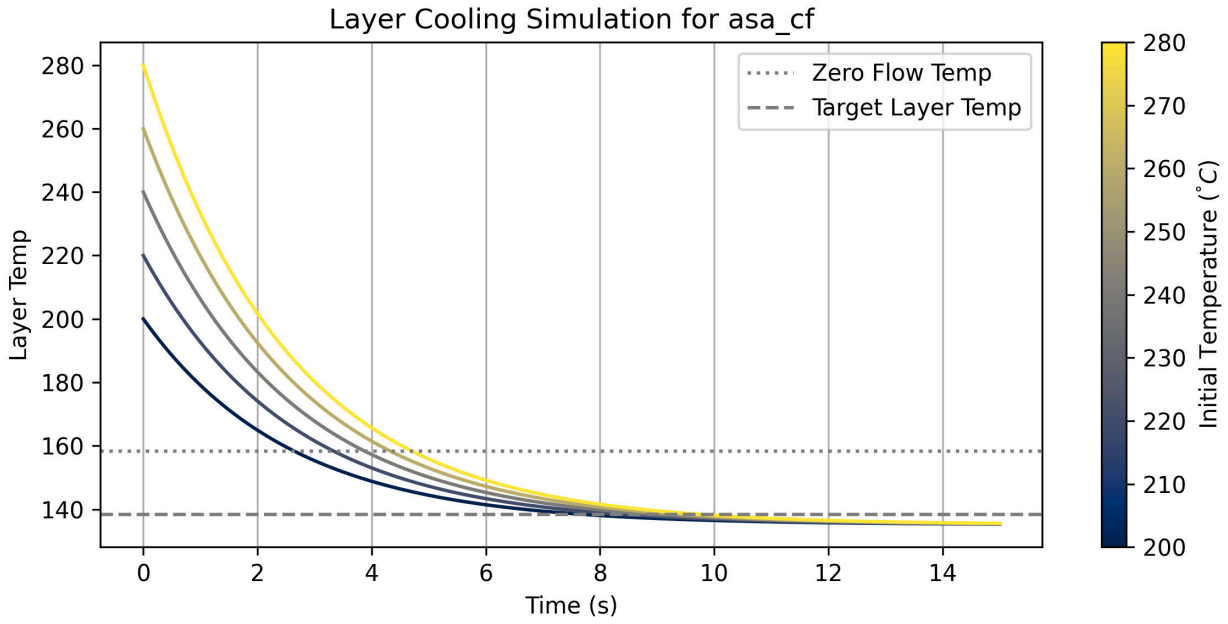


Figure 5.26: A time series simulation of layer cooling for ASA+CF across a range of nozzle temperatures, including target temperature and the material's zero flow temperature.

well tabulated in engineering references. Luckily this value is tied to the *machine* (not the material), so we can tune it over time.

We use the materials' conductivity to estimate the cooling rate *into the layer below* h_{layer}, ($W/m^2 \cdot \Delta T$) - this is a function of the layer height, and is further shimmed by an estimate of the layer to layer contact (l_{iface}), giving us $h_{layer} = (\kappa_{fil}/H) \cdot l_{iface}$.

Finally we need to pick a target cooling temperature T_{targ} - we want this to represent the temperature where our layer is going to be solidified enough to support subsequent layer geometry. This is a real decision variable: for intricate parts we want our previous layers to be relatively cold. With large, blocky prints we can get away with a bigger value here. I tune this *relative* to the material's *zero flow temperature* (from Section 5.5.2). My reasoning is that, when the material is cold enough that it cannot be extruded, it will be relatively stiff. This is exposed as one of the preprocessor inputs as a ΔT such that $T_{targ} = T_{\min} - T_{cool}$

We can work out a cooling time constant (τ) using Equation (5.12), and an equilibrium temperature (T_{eq}) using Equation (5.13). Finally, we use these to calculate the layer cooling time ($t_{\min}$) with Equation (5.15).

$$\tau = (C_{fil})/(h_{air} + h_{layer}) \tag{5.12}$$

$$T_{eq} = \frac{(h_{air} \cdot T_{amb} + h_{layer} \cdot T_{targ})}{(h_{air} + h_{layer})} \tag{5.13}$$

$$h_{layer} = (\kappa_{fil}/H) \cdot l_{iface} \tag{5.14}$$

Symbol	Value	Units	Source
H	Layer height	mm	Part Geo.
l_{iface}	Layer interface proportion	0-1.0	Part Geo.
κ_{fil}	Filament conductivity	$W/m \cdot \Delta T$	Estimated
C_{fil}	Filament heat capacity	$J/m^3 \cdot \Delta T$	Model Fit
T_{noz}	Nozzle temperature	$^\circ$ C	Direct Measurement
T_{targ}	Cooling target temperature	$^\circ$ C	From Flow Models
T_{amb}	Ambient temperature	$^\circ$ C	Known
h_{air}	Heat transfer to air	$W/m^2 \cdot \Delta T$	Estimated
h_{layer}	Heat transfer to layer below	$W/m^2 \cdot \Delta T$	Via fil_k

Table 5.10: Values and Parameters for the layer cooling model.

$$t_{\min} = -\tau \cdot log((T_{targ} - T_{eq})/(T_{noz} - T_{eq})) \tag{5.15}$$

This model is approximate. For example our equilibrium temperature calculation assumes that the previous layer's temperature is a boundary condition, when of course it actually continues cooling into the layer *below it* as time progresses. This means that we probably over-estimate layer cooling times, especially in large prints. Our model could be extended across each layer, but we would need to develop a much more complex (and probably geometric) solver for this — the approximation works well enough especially considering that we are already estimating h_{air} and the filament's conductivity. In Section 5.12.3.3 I discuss the possibility of using the microbolometer that I have attached to the RheoPrinter and FrankenPrusa's hotend to fit these cooling models (and other relevant melt flow physics).

5.8 High Level FFF Planning

OK. By now we understand the models that our machine can fit for flow (and its coupling into the extruder motor) and for layer cooling. The point of this exercise is not just about understanding our machine and materials, it is to enable better *operation* of our machine. So, we now return to the workflow itself (from Section 5.3), and look at the *preprocessor* step (see Figure 5.9). This is where we combine our part file (geometry) and models with heuristics to select high level operating parameters.

In this section I will show how I combine heuristics with flow models to generate processing parameters across any nozzle temperature, and then how nozzle temperature can be selected automatically for time optimality (i.e., printing as fast as possible). In Section 5.12.4, I discuss how this method could be extended to select parameters as a trade-off between a number of optimalities (speed, strength, and precision).

5.8.1 Selecting Flowrates using Models and Heuristics

I use the isothermal map (5.5.3, and shown below) to select flowrates for printer operation. These work well to encode the extruder system's *real physical constraints* which are defined by two boundaries. First (small dashed line) is the nozzle's maximum temperature, a pure hardware limit (hotter temperatures would soften some of the extruder's own construction). Second we have (longer dashed lines) the steady-state flow boundary: this is extracted from the steady-state flow model and plots the maximum sustainable flowrate across nozzle temperatures (see Figure 5.16).

Models do well to describe these boundaries, but I found that actually printing with these models required that I mix in some heuristics to make selections *within* those boundaries. In Section 5.12.4 I discuss how the methods in this chapter could be extended to completely remove this heuristic selection step, but I also discussed in the introduction that turning our digital fabrication workflows into completely autonomous black boxes may not be so valuable: heuristics can be extremely valuable.

Rather than applying heuristics directly (picking flowrates outright as is done in the state-of-the-art), we can apply them *through* models, so that heuristic choices are updated to reflect changes in the underlying constraints. For example in this step I add a lower temperature boundary (solid black line), which is defined by selecting a *minimum* $Q_{\max}$*;* this is the temperature where the steady-state boundary crosses a selected flowrate that we know (heuristically) represents a reasonable lower bound for flowrates. For example in all of the printing that I did with this system, I used $15mm^3/s$ to select minimal operating nozzle temperatures, which is rendered in Figure 5.27.

These boundaries set us up with a space of nozzle temperatures and flowrates that we know will be viable. We still want to select actual flowrates for the part geometry. Here, we apply another set of heuristics. Recall from 5.3 that the path geometry components are binned into line segments for part *infill, perimeters, and details.* For each segment type, we pick a scalar in $0 - 1.0$ where 1.0

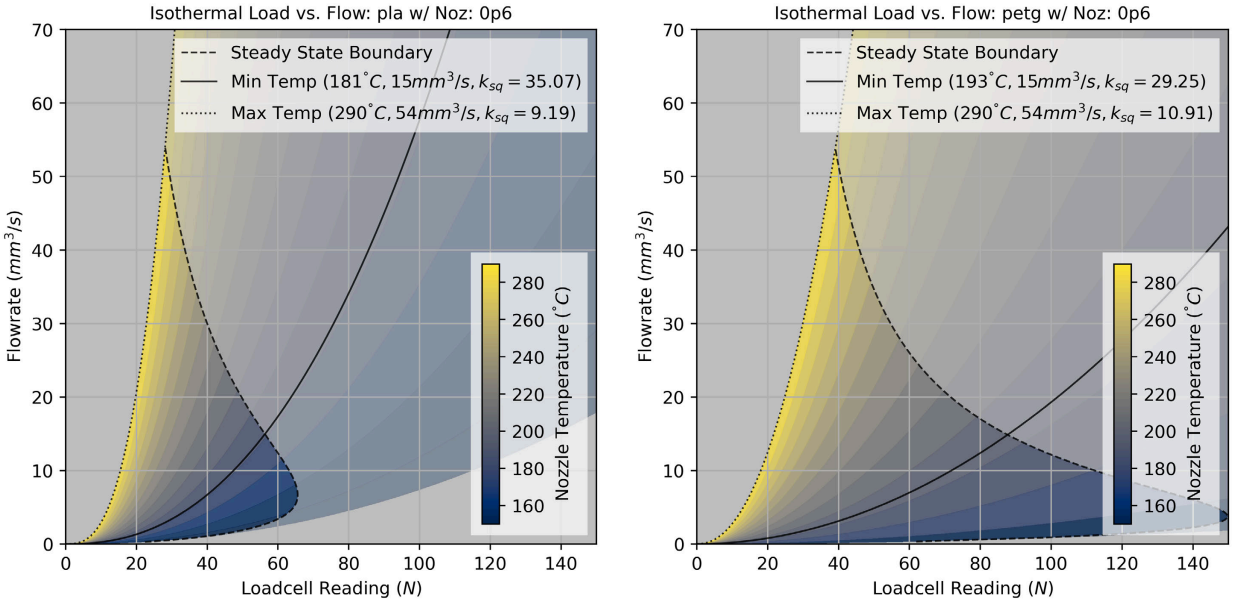


Figure 5.27: Isothermal models for PLA and PETG (both using 0.6mm brass nozzles), showing boundaries of operation: short dashes at the maximum nozzle temperature, longer dashes at the steady-state boundary and a solid black line at the minimum temperature (defined with a minimum flowrate heuristic).

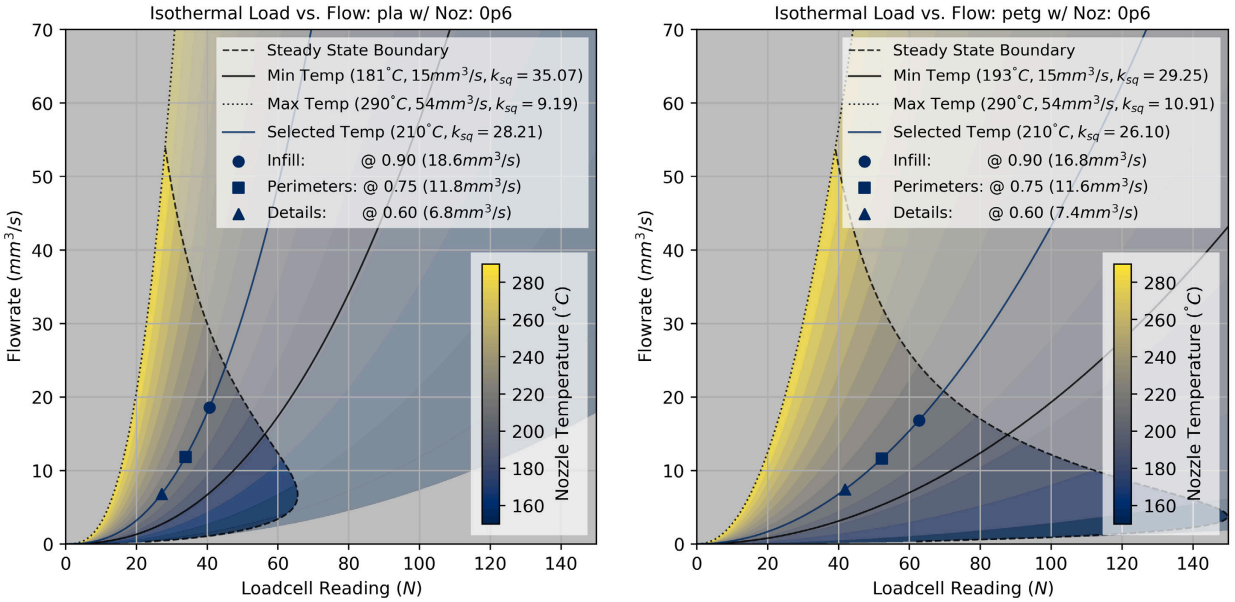
represents the pressure required to generate the maximum flowrate at the selected temperature. The scalar selects for a flowrate between zero flow / zero pressure, and this maximum. Figure 5.28 demonstrates this process, using PLA and PETG models and applying the same temperature and flowrate heuristics to both.

This process nets us maximum flowrates per segment type, and maximum *translational* rate for each individual segment (dividing the segment's width (which is effectively mm^3/mm) by the selected flowrate for that segment type). This encodes both flow heuristics (flow is tidier and more linear at lower pressures) and motion heuristics: we care more about the quality of our part's *perimeters and details* than we do about its infill, and lower speeds will generate cleaner motion trajectories. By selecting larger pressure scalars for *infill,* we allow the machine to use more translational velocity there *but* that velocity is carefully calibrated with respect to flow. Accelerations are also model / heuristically tuned using current scalars, as I discuss under the next heading.

5.8.2 Applying Acceleration and Jerk Heuristics

As I mentioned above, we typically care more about the quality of our print's outer tracks (perimeters) than its infill. We also know that when a motion system is operating near its limits, it will *generally move with less grace.* This is a relatively imprecise statement, but we are talking about heuristics here.

The state-of-the-art workflow lets us tune translational rates and accelerations (across segment types) directly. As I've discussed, this presents the issue of scaling flowrates *alongside* translational



Input	Selection	Resulting Flowrate
Temperature	210 ° C	—
Infill Scalar	0.90	$18.6mm^3/s$
Perimeter Scalar	0.75	$11.8mm^3/s$
Detail Scalar	0.60	$6.8mm^3/s$

Input	Selection	Resulting Flowrate
Temperature	210 ° C	—
Infill Scalar	0.90	$16.8mm^3/s$
Perimeter Scalar	0.75	$11.6mm^3/s$
Detail Scalar	0.60	$7.4mm^3/s$

Figure 5.28: Input heuristics and model-mediated flowrates (tables, bottom), plotted in the isothermal map for PETG with a 0.6mm brass nozzle (left) and for PLA (right, same nozzle).

rates; making modifications to the motion system parameters changes flow parameters even though the change is not reflected anywhere. It also does not show us *how close* to the motion system's actual limits those rate selections are, we must guess and check. Our strategy here uses flow models and normalized pressure targets for the same task, as I described above.

We also want to scale acceleration and jerk (its derivative). We have an excellent basis for this in the motion models of our machine, whose fundamental limits come from current saturations in the actuators (see Section 4.3). If we scale the maximum current that the machine is allowed to *deploy*, we scale its acceleration. Here I use the same strategy as in the flowrate selection: using a scalar to de-tune the system from the model-defined limits, rather than setting limits directly.

To tune jerk, I scale the actuators' deployment of $\dot{I}$, the rate of change of current. Reducing jerk further smooths motion and reduces the frequency of the disturbances that the actuators will send into the mechanical system. In Section 4.8.5 I discuss the missing component of these models, which would address resonant excitations of our systems directly.

In all, we have eight heuristics that we apply at this level, all listed in Table 5.5 under *Current Scalars* and *Current Slew Rate Scalars.* The preprocessor applies those on a per-segment basis to the geometry, which can be seen in Table 5.6.

5.8.3 First Layer Speed

Finally, we have this last *pure* heuristic to apply. FFF veterans know well that the quality of a print's first layer is critical. In fact, there is an entire subreddit dedicated to images of high quality first layers[93]. The main difference between this layer and the rest is that it must adhere to the printer's bed, rather than to itself. Depending on how readily our print material sticks to our bed material, we want to reduce the speed at this layer so that adhesion is sure to work. It is almost always the case that bed adhesion is much worse than layer-to-layer adhesion: bed are engineered to stick *well* (but not so well as to prevent us from retrieving our prints!). To ameliorate this, first layers are normally printed at ~ 20% of the "normal" speeds. Lowering the speed gives the polymer more time to bond to the bed, and reduces vibrations or other errors that arise from translating around too fast.

While it should be possible to build a model for bed (and inter-layer) adhesion, I have not tried to do so. So, we stick with heuristics: first layer speeds are capped at some direct value, which can be calculated as 10% of the maximum flowrate given by our extruder model, or set directly as a maximum traversal rate in mm/s.

5.8.4 Time-Optimal Nozzle Temperature Selection

OK - flow models show us where operation of our hardware is possible, and we just developed some methods to pick actual parameters for operation using heuristics $\times$ models at any nozzle temperature. We still need a good way to actually select that temperature, which is probably the single most important parameter in FFF printing: changes to nozzle temperature change almost all other aspects of the system.

One of the surprising advantages of the model-based workflow is that it enables us to select good parameters *across* nozzle temperatures: everything else is updated automatically. In the state-of-the-art, it is most common for values to be tuned by hand *around some temperature,* i.e. the nozzle temperature is refined first, and then everything else is dialled in around it.

While I discuss how we might get smarter about multi-objective print optimizations in Section 5.12.4, for now I am framing this as a time-optimal selection with heuristics applied such that we also produce parts of reasonable quality. But! If we were to select based on flowrates alone, we would surely always pick the hottest nozzle temperature (since this is where flowrates are maximized with minimal pressures). Surely this is not really optimal, because (as discussed in Section 5.2.1.2) part *cooling* is also important: using excessively hot print temperatures leads to part slumping.

As it turns out, there is a very clear optimization to run here: hot nozzles give us large flowrates, but subsequently increase layer cooling times. Colder nozzles reduce flowrates, but also decrease layer cooling times. So the trade-off emerges around part *geometry;* large parts (with already long

[93]The subreddit in question is reddit.com/r/FirstLayerPorn.

layer times) should tend towards hotter print temperatures, whereas smaller parts (where new layers appear faster) should tend towards colder nozzle temperatures.

I already explained how the minimum layer time is calculated for any material and nozzle temperature (5.7). This enables us to run the per-part optimization, but we first need to develop a tool that scales layer speeds such that minimum layer times are achieved. That is next in Section 5.8.4.1, and is a method that is common in state-of-the-art slicers (although they use heuristically tuned minimum layer times, whereas ours are estimated with the cooling model). I then describe and show some results from the nozzle temperature selection optimization in 5.8.4.2; this optimization is new work.

5.8.4.1 Adjusting Segment $V_{\max}$ for Minimum Layer Times

Given a nozzle temperature and the velocity selections described above, we can estimate the total layer time for any layer in the part at any temperature. To enforce minimum layer times, we skew velocities within each layer such that the layer's total time increases. However, we prefer to skew velocities in segments that are less critical to our overall dimensions and appearances, i.e. the infill is modified first, if this does not suffice we move to the perimeters, and if this is still not enough we modify the external perimeters. This insight comes from Prusa Research's note on the topic [328]: polymer tracks extruded at higher speeds / pressures have slightly different visual properties (more matte) than those printed at lower speeds (which tend to be glossier). They also have different rates of die swell. To reduce these effects, we try to avoid skewing external perimeters unless it is truly necessary for the target lap time. Finally, we have a global limit to the skew, such that no segments have a translational velocity of less than $10mm/s$.

5.8.4.2 Optimal Nozzle Temperature for Minimum Print Times

So by now we can pick a viable temperature range from some minimum up to the nozzle's maximum, and then at any temperature we can select flowrates for each segment type, and calculate and then enforce a minimum layer time. Having done all of that work, we can easily pick a nozzle temperature that will give us the minimum total print time.

This optimization turns out to be exceptionally simple: rather than optimizing over nozzle temperature, we simply evaluate the total expected print time at every nozzle temperature between the minimum and maximum, and pick the temperature with the smallest overall time - evaluating each is relatively cheap computationally (it takes about thirty seconds to do so on a modern laptop).

The outcome is also intuitive; in smaller parts, total print time tends to be dominated by the minimum layer time and so colder nozzle temperatures (less cooling) are preferred. As parts increase in size, print time is dominated more by the flowrate / time to actually deposit the layer - so we want to increase the temperature to increase these flowrates.

There is a crossover *print size* that depends on our cooling models, flow models, and selected flow heuristics. i.e. in the RheoPrinter with PETG (and a 0.6mm Nozzle) I found that prints about Benchy sized (60x30mm) clearly favour cold flows, whereas even slightly larger prints (90x70mm) clearly favour maximally hot flows. Parts in between exhibit a clear time-optimum temperature somewhere between our minimum and maximum temperatures. In these cases, flowrates and

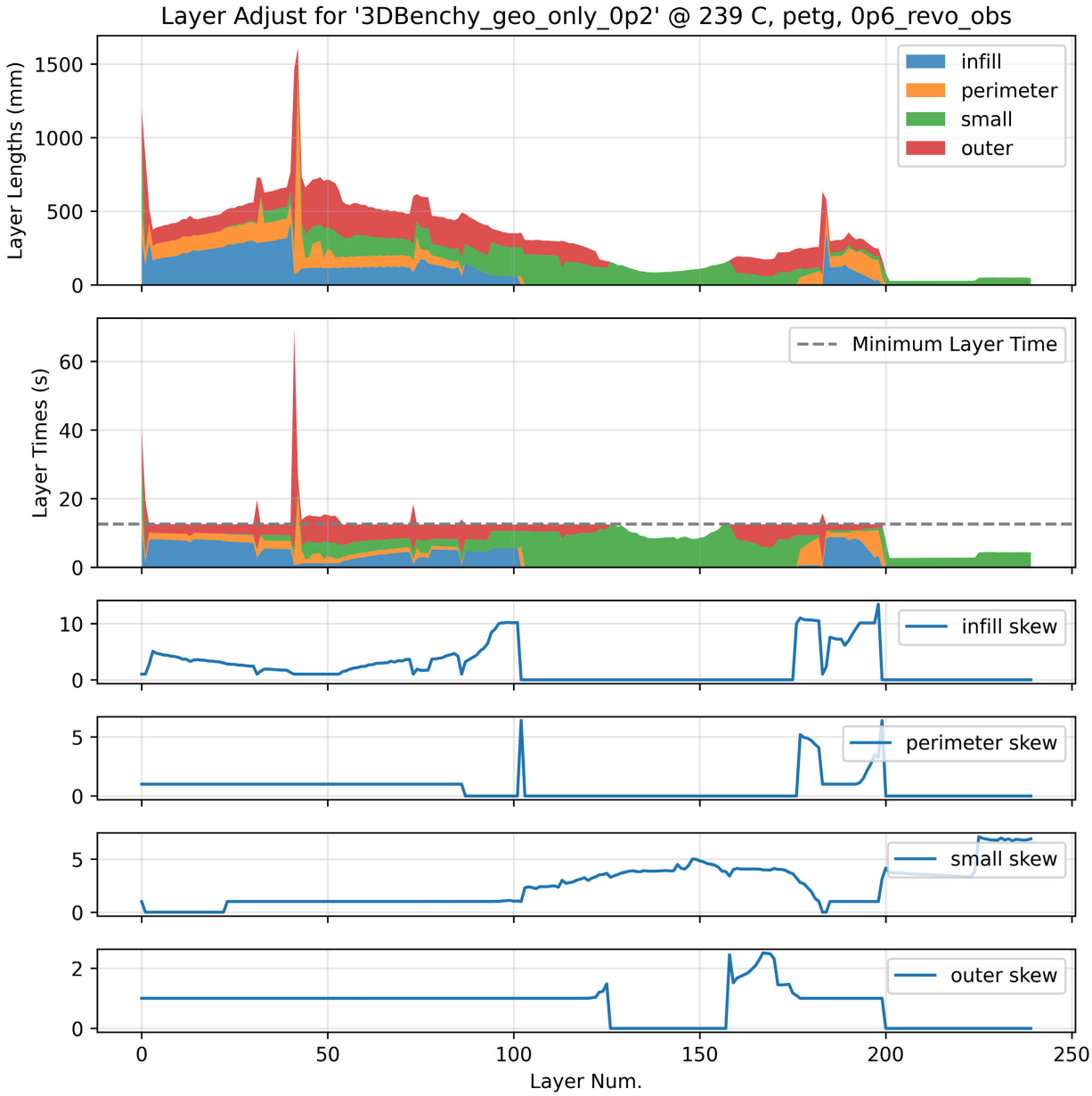


Figure 5.29: This figure shows layer-by-layer (horizontal axis) segment lengths (top row) and times (second from top), colored by segment types: infills, perimeters, small perimeters, and outer perimeters. To meet a minimum layer time (horizontal dashed line, second from top), rates for these segments are skewed: skew proportions are plotted for each category in the bottom four rows.

layer times increase together and balance such that the average layer time *naturally* meets the minimum layer time (Figure 5.31). If we increase our estimate of the printer's cooling capability, the crossover size gets smaller (and vice versa). For polymers where temperature increases don't lead to exceptional gains in flow, colder temperatures overall are preferred, etc.

I had always thought that printers were over-optimized for producing small, detailed parts. The 3DBenchy is the de-facto parameter tuning object partially because it encapsulates lots of different

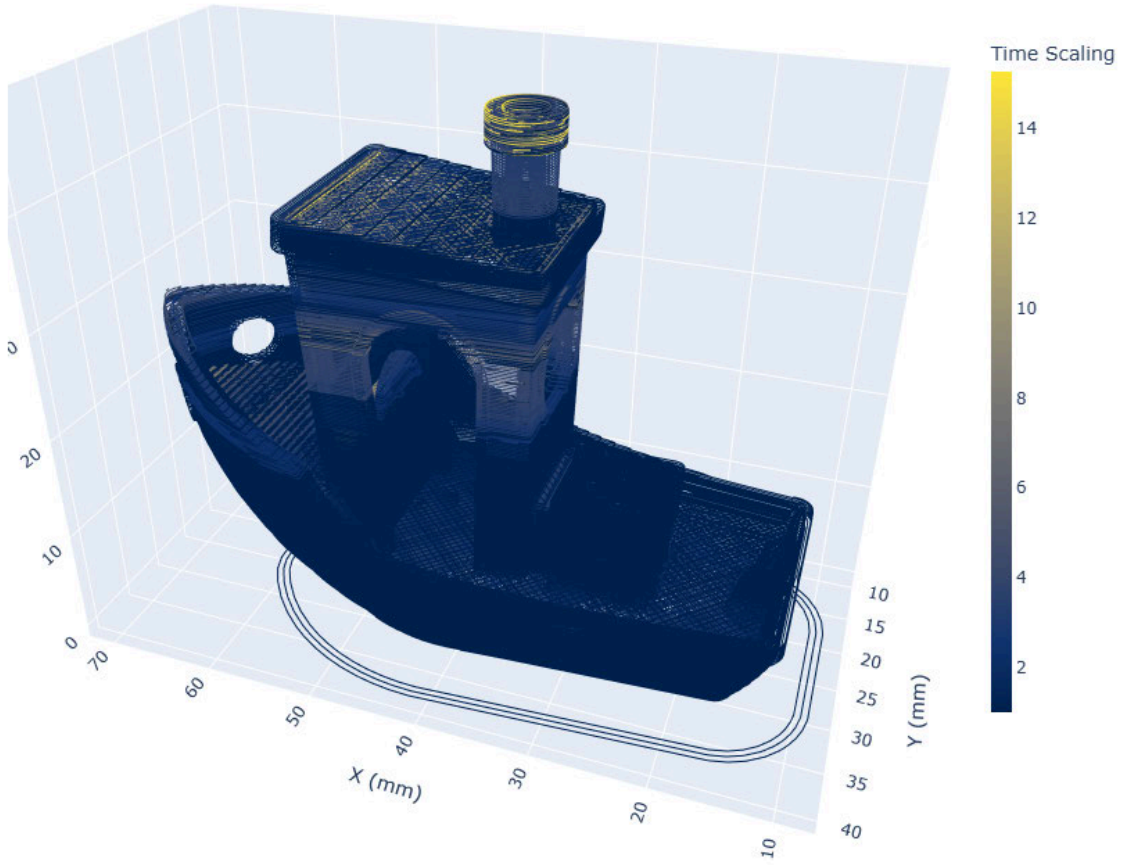


Figure 5.30: This visualizes the layer skews in the figure above in a 3D plot. Most of the skewed segments are within the infill, but we can see that smaller layers (near the top) have larger skews than others: these have been slowed to account for layer cooling time.

features but also because it prints relatively quickly which decreases the time for human-in-the-loop tuning iterations. However, the 3DBenchy does not necessarily represent users' most common print jobs. In mechanical prototyping especially, we typically print larger beds with i.e. multiple small parts or larger single parts. This means that print parameters have been hand tuned tend to favour colder temperatures overall, even though there are huge speed gains to be had by increasing the nozzle temperature. For example we can select a very hot flow for the larger part in Figure 5.31 (second row) that decreases the total print time from 30 minutes to 20 minutes, a substantial speedup. In the case of this work, because we develop print parameters over the entire range of viable temperatures, we can more flexibly adjust that single parameter and let our models and "model mediated heuristics" dial out the rest of the knobs.

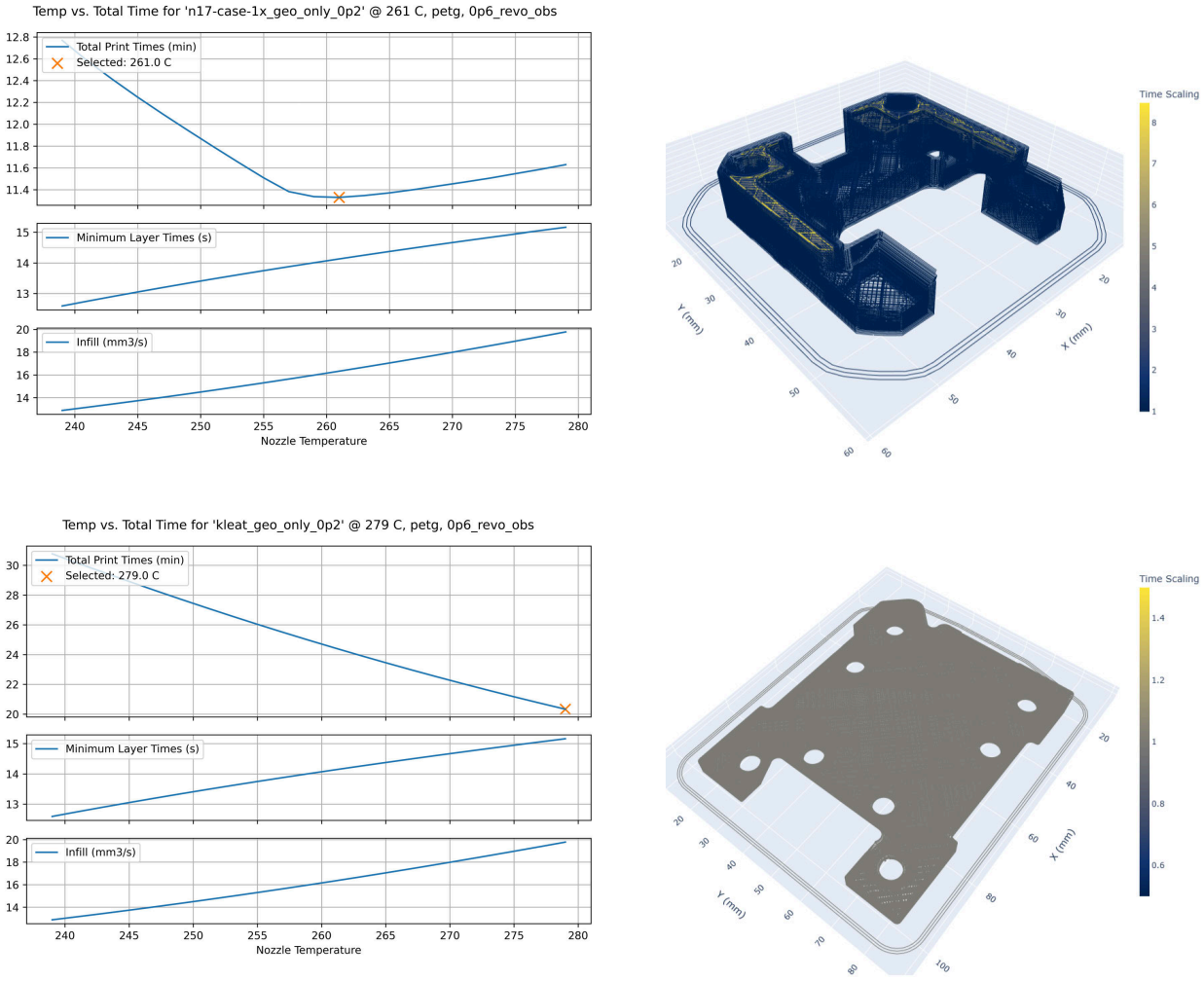


Figure 5.31: For a circuit enclosure (top) and the RheoPrinter's toolchanger plate (bottom), we see the total print time across temperatures, as well as the minimum layer times and infill flowrates (left plots). These are the same plots as in Figure 5.29 and 5.30 for the 3D Benchy. The Benchy (~60x30mm) has a minimum print time at the minimum nozzle temperature, while the plate (~60x80mm) has a minimum print time around the maximum nozzle temperature. The circuit enclosure (~40x40mm) shows that at crossover sizes, a clear optimum emerges.

5.9 Real-Time FFF Planning

Finally! It is time to actually operate the printer.

I have mentioned *constrained optimization* in a number of places in this thesis. In the last few sections, I have shown how we can build 3D printing hardware (Section 5.4) that can be used to fit models (5.5, 5.7), and how those models can be used to select and adapt print parameters, mixing heuristics with feedback (5.8.1). I also showed that nozzle temperature selection can be framed explicitly as an offline optimization (5.8.4).

Using models for high-level tasks and optimizations like those is in itself productive; in an earlier paper I used similar methods to pick key slicer parameters for state-of-the-art printers automatically [305], but they were still *operated* with their existing GCode interpreters. As I've mentioned, these are responsible for low-level control of the hardware and managing our machine's dynamical constraints, which is primarily done with a velocity planner. Those were the topic of Chapter 4, where I developed a new type of planner that can be configured directly with models for motion. In that chapter (and here, in Sections 5.2.3 and 5.2.4.2) I often discussed how machine control is really constrained by *both* motion and process physics even though these are managed independently in the state-of-the-art and in most other research.

So, I wanted to see if it were possible to extend the motion controller from Section 4.6 to also optimize over constraints that emerge from *process control.* Doing so would show that models of machine operation can be used across multiple levels of a digital fabrication workflow. This could simplify machine systems configuration by organizing it around a shared representation of constraints rather than via multiple different hand-tuned parameter sets that each encode different aspects of the same underlying physics. It could also unlock some performance; process constraints as articulated in CAM tools need to hold in steady-state, but we have different flow dynamics in short time scales than in steady-state (see Sections 5.5.4 and 5.11.5).

In this section, I explain how I extend my velocity planner to include flow control (Section 5.9.1, 5.9.1.2, 5.9.2) and how it is connected to printer hardware via PIPES / MAXL (5.9.3). I use this planner to operate the printer in 5.10. It helps to reduce the number of parameters we must tune in that workflow (5.10.3) and its outputs can be used to learn about our hardware (5.11.1) and to more clearly connect system tuning to system constraints (5.11.2). They can also be used to inspect modelling errors' correlation to print job errors (5.10.5).

5.9.1 Optimization Formulation

I explained how the velocity planner is structure overall in Section 4.6.1. Here we are just extending that system to add models for flow, so this section will be difficult to understand if you have not read that one.

The planner is one part of a larger FFF workflow that I described in Section 5.3. It consumes a *run file* from that workflow's preprocessor. The run file contains an unplanned target trajectory of line segments. The maximum velocity $v_{\max}(s)$ for each was set using flow models and flow heuristics in 5.8.1. Each segment also has actuator current deployment scalars $I_{\text{prop}}(s)$, $\dot{I}_{\text{prop}}(s)$, these will scale each actuator's permitted current limit from their underlying model-based maximum (each scalar is within $[0, 1]$).

Here we also have FFF-specific values. First is a target flowrate for each segment $Q_{\text{targ}}(s)$ which is defined spatially in mm^3/mm; this is the extrudate track width: when we divide it by layer height we get mm^2/mm i.e. just mm. We also have extruder motor current scalars $I_{\text{prop,e}}(s)$, $\dot{I}_{\text{prop,e}}(s)$.

The preprocessor also adds definitions of flow models for the part and four heuristics for the extruder system. These are two direct limits to the extruder motor's motion e_{vmax}, e_{amax} and two values for retractions F_{off}, e_{assym}. I found that these were required to run the printer successfully because there seems to be some loss in two parts of the controller.

The first loss is in the extruder dynamics and the bandwidth of the extruder motor's velocity controller. Near the end of each track the extruder needs to retract filament very quickly. This causes the extruder motor's velocity to spike and then fall quickly back to zero. The extruder motor coupling from Section 5.6 should impose the velocity limit here based on the motor and flow system's underlying dynamics, but the velocity controller in the motor firmware needs to track the change accurately. However, it has limited bandwidth (see Section 4.4.0.2), hence the manually imposed limit.

The retraction heuristics are a negative pressure offset F_{off} that is applied to the nozzle pressure target during non-extruding moves and then a scalar e_{assym} that is applied to the extruder motor velocity during de-retraction. I found that these were required to avoid stringing and oozing for two reasons. The first is that even at low or zero measured pressure we see some extrusion because the filament in the melt zone expands as it heats up. This is especially problematic when we finish a track that was extruded at high speeds where the melt flow's temperature can be quite a bit colder than the nozzle temperature. The second is that the flow models do not always predict nozzle pressure perfectly: where they under-predict pressure, the planner does not retract filament sufficiently to drive the real pressure to zero.

It should be possible to remove the retraction heuristics if the flow models can be improved across more dynamical states. I discuss that task in Section 5.12.1, which would allow us to update flow models from data generated by the machine during normal operation (5.12.2). The velocity controller in the extruder could certainly be improved or should actually track cumulative position of the filament so that negative velocity spikes are integrated properly over time. The best solution overall would be to close the pressure control loop in realtime: this could be done at all times if the models for flow are well-matched to reality, or it could be implemented just during non-extruding moves to drive the pressure back to zero regardless of modelling errors. I discuss this in some more detail in Section 5.12.3.1.

OK, back on topic: we take those new target values and limits and add them to the target trajectory. The planner otherwise works in the same basic manner as previously described (4.6.1): it selects $v(s)$ for each segment, calculates required accelerations and time-steps between each, and then broadcasts those states through machine models to compute required internal states. These are compared in a cost function to their limits, and the $v(s)$ are optimized such that speed is maximized while constraints are not exceeded.

The basic layout of this process is below, adding flow-related components: calculating output flowrates and the melt flow temperature (Section 5.9.1.1), calculating flow dynamics (5.9.1.2) and then extruder current requirements (5.9.1.3), finally adding those new values to the cost function (5.9.2).

$$\overbrace{\begin{bmatrix} Q_{\text{trg}} \\ \vdots \\ s \end{bmatrix} \begin{bmatrix} I_{\text{prop,e}} \\ \vdots \\ s \end{bmatrix} \begin{bmatrix} \dot{I}_{\text{prop,e}} \\ \vdots \\ s \end{bmatrix}}^{\text{Additions to Input Trajectory}} \rightarrow \overbrace{\begin{bmatrix} v \\ \vdots \\ s \end{bmatrix}}^{\text{Select}} \rightarrow \overbrace{\begin{bmatrix} \vec{a} \\ \vdots \\ s \end{bmatrix} \begin{bmatrix} \Delta t \\ \vdots \\ s \end{bmatrix}}^{f_a()} \rightarrow \overbrace{\begin{bmatrix} Q_{\text{out}} \\ \vdots \\ s \end{bmatrix}}^{v(s)\cdot Q_{trg}} \overbrace{\begin{bmatrix} T_{\text{melt}} \\ \vdots \\ s \end{bmatrix}}^{\text{NRG}} \rightarrow \text{Below} \tag{5.16}$$

$$\text{Above} \rightarrow \overbrace{\begin{bmatrix} F \\ \vdots \\ s \end{bmatrix} \begin{bmatrix} \Delta F \\ \vdots \\ s \end{bmatrix} \begin{bmatrix} Q_{\text{in}} \\ \vdots \\ n \end{bmatrix} \begin{bmatrix} \dot{Q}_{\text{in}} \\ \vdots \\ n \end{bmatrix}}^{\text{Flow Dynamics}} \rightarrow \overbrace{\begin{bmatrix} I_{\text{req,e}} \\ \vdots \\ n \end{bmatrix} \begin{bmatrix} I_{\text{lim,e}} \\ \vdots \\ n \end{bmatrix} \begin{bmatrix} \dot{I}_{\text{req,e}} \\ \vdots \\ n \end{bmatrix} \begin{bmatrix} \dot{I}_{\text{lim,e}} \\ \vdots \\ n \end{bmatrix}}^{\text{New Intermediate Values}} \tag{5.17}$$

The planner's internal structure is diagrammed above in Equation (5.16) and (5.17), and a screen capture of the planner's realtime plotter is in Figure 5.32, showing the motion trajectory (left) and internal states (right).

5.9.1.1 Output Flowrate and Melt Flow Temperature Calculation

To work backwards from the chosen solution for $v(s)$ towards the system's constraints the planner first computes the cartesian acceleration and time step required in each segment ($\vec{a}_{\text{req}}, \Delta t$). That was explained in Section 4.6.1.1 and the method for broadcasting those values back through motion models is in 4.6.1.2.

In this extension of the planner, we also compute flow dynamics so that they can be compared to their related constraints. This begins with finding the flowrate at the nozzle tip, which is a simple product of the target flowrate $Q_{\text{targ}}, mm^3/mm$ and the selected velocity $v(s), mm/s$. The result is Q_{out} which is in mm^3/s.

We then have the more substantial challenge of estimating the melt flow's internal temperature T_{melt}, which can deviate from the nozzle temperature significantly after periods of sustained outgoing flowrates, see Section 5.5.4.

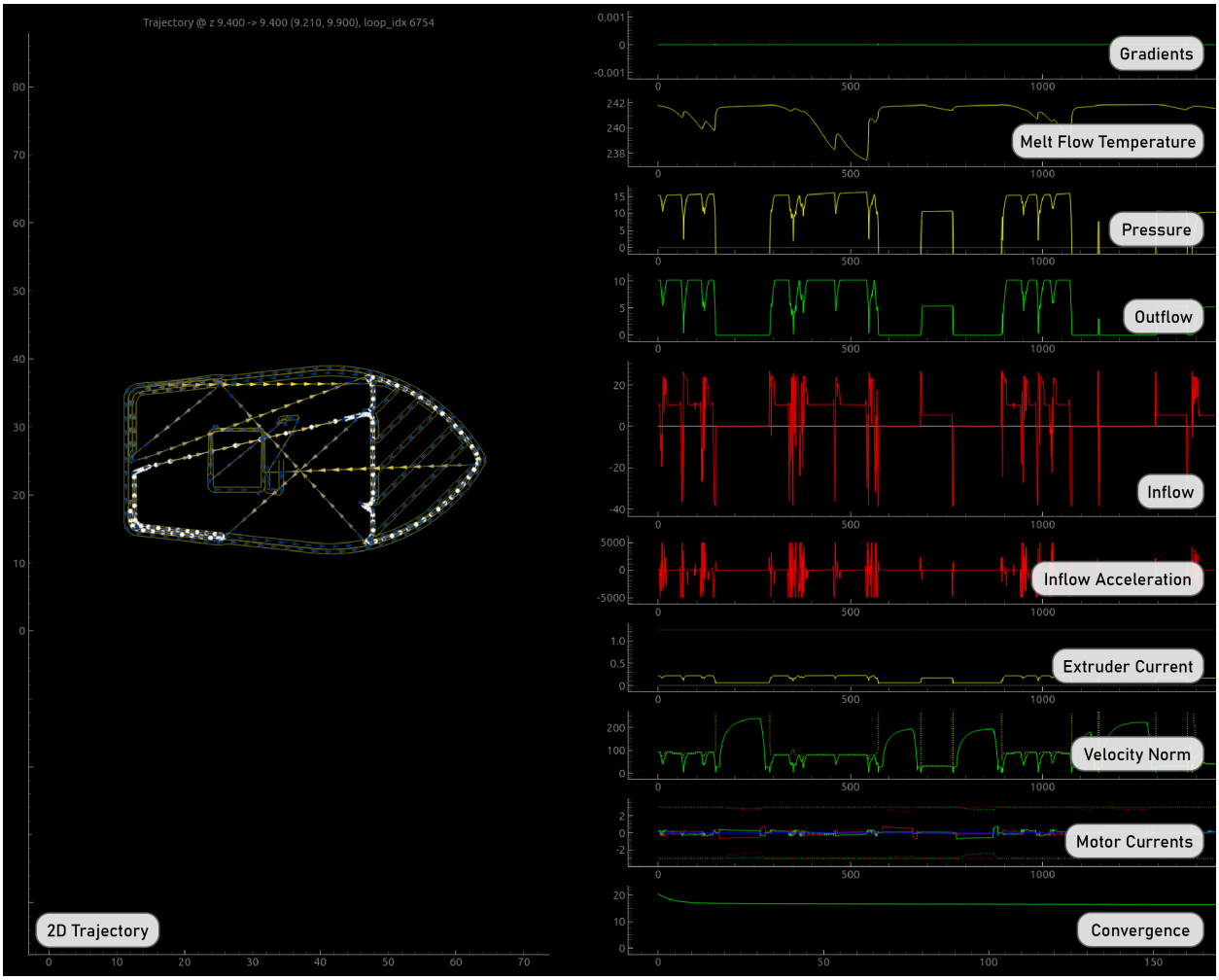


Figure 5.32: This is a screen capture of the planner's real-time output. The target trajectory is rendered at left and plots on the right show planner outputs. From the top: gradients of $v(s)$ with respect to the cost function, the melt flow temperature estimate, extruder pressure estimate, outflow, inflow, inflow acceleration, extruder current, $v(s)$, actuator current estimates, and the planner's convergence.

I do this using an approximate energy model that cumulatively sums the "dwell time" for the filament in the melt zone, i.e. how long on average the filament in the melt zone has spent between the start of the heater block and the nozzle tip. Let's call this t_{dwell}, it is measured in seconds. This method uses a measurement of the total *volume* of the melt zone $\mathcal{V}_{\text{melt}}, mm^3$ which is a simple diameter $\times$ length of the melt zone. Special warning: $\mathcal{V}$ here is volume, not velocity v.

We cannot explicitly parallelize this calculation because it is history dependent. However, we can use an associative scan [329] which reduces time complexity from $O(N)$ to $O(logN)$ using a tree-reduction algorithm. This is implemented natively in JAX which also handles differentiation of the scan for us [265].

The associative scan requires that we formulate the computation with two prefixes, here I have one that describes a proportion of time that is added to the integral at each step and a proportion that is removed from the integral at each step:

$$\begin{aligned} \text{out} &= 1.0 - Q_{\text{out}} \Delta t \\ \text{in} &= \mathcal{V}_{\text{melt}} \Delta t \end{aligned} \tag{5.18}$$

This is perhaps a bit non-intuitive, my best explanation is this: at each step we add an amount of time that is scaled by the melt zone's volume, and we remove the proportion of that time that has flowed out of the nozzle tip. After running these operators to the scan, we get the melt flow's dwell time for each step along the trajectory. We can use that to calculate the average flowrate within the melt zone:

$$Q_{\text{avg}} = \mathcal{V}_{\text{melt}} / t_{\text{dwell}} = mm^3/s \tag{5.19}$$

In Section 5.5.4 I showed that we can extract a curve at the nozzle's temperature that plots the temperature difference between the steady-state and isothermal flow models for any Q_{avg} (Figure 5.21). Evaluating that curve returns an estimate of the actual melt flow temperature T_{melt}.

5.9.1.2 Flow Dynamics Calculation

Given the melt flow's temperatures, we can use the isothermal flow model from Section 5.5.3 to find the pressure $F(s)$ required to generate the target flowrate. We also get the filament's spring rate at that temperature k_{sq}.

We can use these to calculate the total volume of filament that must be compressed at this point such that the target pressure is reached, $\mathcal{V}_{\text{sq}} = F/k_{sq}$. To maintain this pressure from step to step, the extruder must drive filament such that this balance is maintained:

$$\Delta \mathcal{V}_{\text{sq}_{(s)}} = \mathcal{V}_{\text{sq}_{(s+1)}} - \mathcal{V}_{\text{sq}_{(s)}} + Q_{\text{out}_{(s)}} \Delta t_{(s)} \tag{5.20}$$

That is, the change in volume is effectively the pressure delta *plus* the flow out from the nozzle, or is the dynamic step plus the steady-state step. That can be converted back to the extruder motor inflow:

$$Q_{\text{in}} = \Delta \mathcal{V} / \Delta t \tag{5.21}$$

I want to make a quick note here about the key difference in the extruder vs. the motion system framing: the motion dynamics are carefully articulated in *phase space,* splitting the problem up into very many small segments that are each some distance apart. While this works well for motion, it makes for nonuniform Δt at each step and the extruder's dynamics change more dramatically with varying time steps. This introduces some trouble when we actually solve this system; I discuss the trade-off and possible solutions in Section 4.8.2.

5.9.1.3 Extruder Current Calculation

I described how motor models are coupled to the extruder system in Section 5.6. At this point we have all of the states required to calculate the current that will be required by the extruder.

$$I_{\text{req,e}} = f(Q_{\text{in}}, \dot{Q}_{\text{in}}, F)$$

We take a discrete derivative of extruder current to get $\dot{I}_{\text{req,e}}$, and calculate the motor's current limit as usual (see Section 4.3.3.2). The extruder motor's velocity is scaled by the extruder's effective gear ratio (converting from inflow to motor rotor velocity).

5.9.2 Extending the Cost Function, Update, and Output Steps

Extending the cost function is perhaps the most straightforward step, we simply add new terms. Those new states include:

- Extruder inflow and inflow acceleration $Q_{\text{in}}(s), \dot{Q}_{\text{in}}(s)$
- Extruder current requirements $I_{\text{req,e}}(s)$
- Extruder current slew requirements $\dot{I}_{\text{req,e}}(s)$

Each of those states also has associated limits. Exceedances of these values are calculated using the methods described in Section 4.6.1.3. The maximum extrusion force F limit is implicit via the extruder motor's current requirement.

The planner generates new $v(s)$ using gradient-descent over this cost, which now includes contributions from the motion system alongside those from the extruder system. That update step is described in Section 4.6.1.4 and no changes from this chapter are required.

Finally, we need to output the new control value: this is just the extruder motor's velocity i.e. Q_{in}. For this step I simply add one more output to the method from Section 4.6.1.5.

5.9.3 Connecting the Planner to Hardware

This step is also the same as in Section 4.6.2; the planner is instantiated as a PIPES (Section 2.4) block via an interface class that relays data into- and out-of PIPES using sockets. It is connected as a MAXL (Section 2.5) component, and solution components emerge as spline control points, metered by a MAXL timer block. Those go to actuators, which each runs low-level closed-loop controls — PIDs for position control and PI for velocity control of the extruder motor, in which case the basis spline is used as a velocity reference instead of a position reference. Data from motors and sensors are piped back into *data collector* classes, which store timestamps and samples.

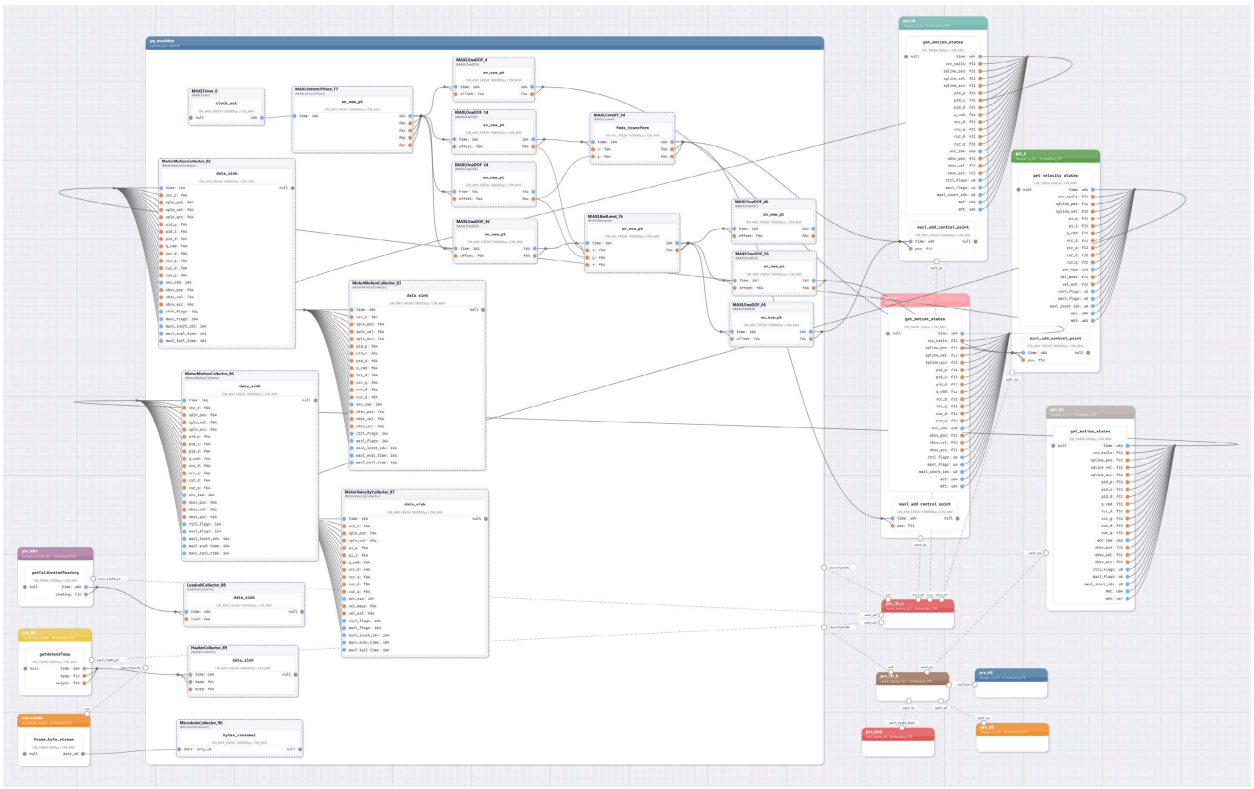


Figure 5.33: The rest of the PIPES graph that runs the machine. The large blue block contains objects running in Python, others are hardware.

5.10 Evaluating Feedback-Based FFF

So, up to this point I have explained the feedback-based FFF workflow (5.3) and its constituent parts: the preprocessor (5.9), the motion controller (5.9), and the models used in both (5.5 and 5.7).

To evaluate these new tools, I used the *FrankenPrusa, (5.4.2)* to print an array of part geometries using a range of materials and nozzles. In Section 5.10.1 I show the resulting flow models and images of prints, alongside tables that render process *saturations* (i.e. which underlying machine constraint most limited printing time: see Section 5.11.1) and modelling errors. In Section 5.10.3, I explain how the feedback-based workflow compares to the state-of-the-art in reducing the number of parameters that must be tuned by hand and allows us to print in configurations that are not covered by off-the-shelf *presets.*

Because the FrankenPrusa's underlying hardware is identical to one state-of-the-art machine (a Prusa Core One, (2025)), I can also compare this workflow's output to the state-of-the-art quantitatively on print speed (in Section 5.10.4) and on qualitative part precision (Section 5.10.5).

Finally, I examine model-fit quality in Section 5.10.6 by comparing the planner's predictions of sensor readings real-world measurements made during operation.

I discuss all of these results, and some other ways in which feedback-based FFF differs from the state-of-the-art, in Section 5.11.

5.10.1 Printing with the Model-Based FFF Workflow

I printed parts using a set of materials and nozzles that I intended to be representative of state-of-the-art FFF operation. It includes the incredibly common "default materials" like PLA and PETG (which are useful to see because they are well understood), as well as fiber-filled versions of the same, (because it is often difficult to ascertain exactly what these blends are and our system can operate well even with that uncertainty). I also included some high performance materials that are classically difficult to print like Nylon and PolyCarbonate-ABS blends (and both with glass or carbon fill), since these are excellent engineering materials, and I was curious to see if our system could manage to learn how to print with them. Finally, I included natural materials that are interesting for their renewable properties and an Aluminum Fill material because I was interested to see a thermodynamic outlier; it's conductivity and heat capacity should be much larger than other polymers.

I also printed parts using three different nozzles; a 0.6mm brass nozzle, and two "high flow" nozzles: these have a copper core, hardened steel tip, and more complex internal geometry that is designed to improve flowrates. They are also finished with a high hardness coating [330] in order to resist abrasion from filled filaments. Along this axis, I show how our flow models adapt not only to changes in material but also to changes in nozzles.

I've organized these prints by filament, roughly in order of most- to least-common. I present a flow model and at least one test part per filament, and (where they are available) compare those to prints on an unmodified Prusa Core One. In all cases, I use geometry generated by *Prusa Slicer* [296] (version 2.9.4) using the *0.2mm Speed* preset, with *50% infill*. Comparison prints use parameter presets for these settings, and our prints use the feedback-based workflow (diagrammed in Figure 5.9) to operate.

5.10.1.1 PLA, 0.6mm Brass Nozzle

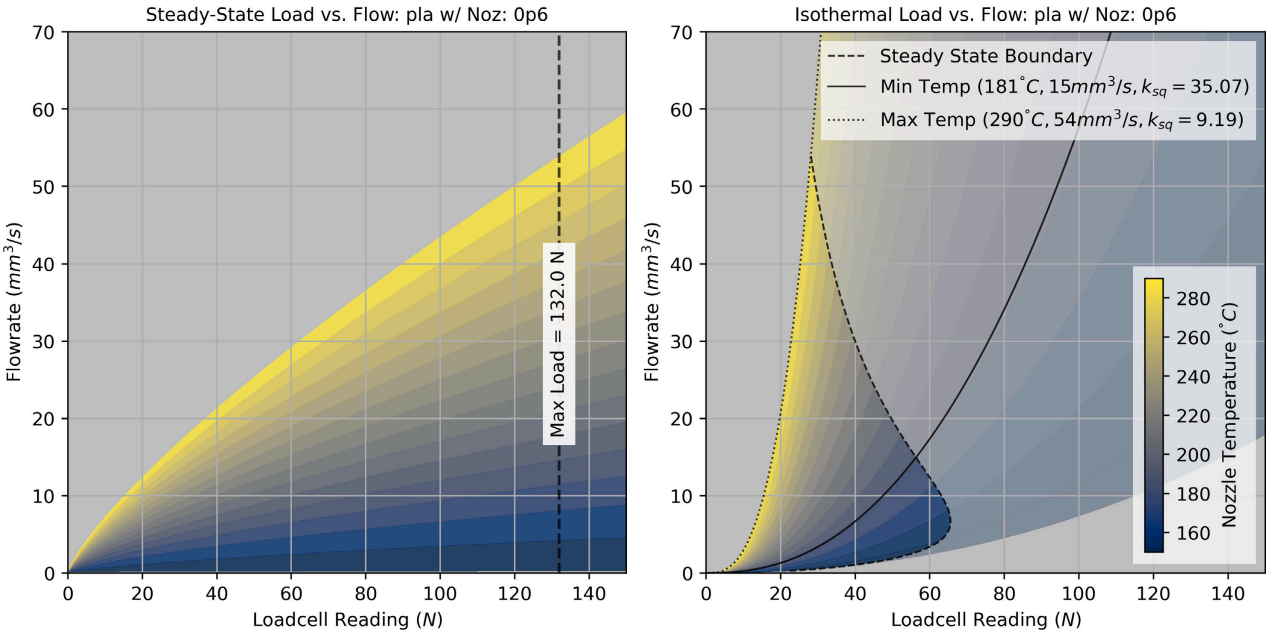
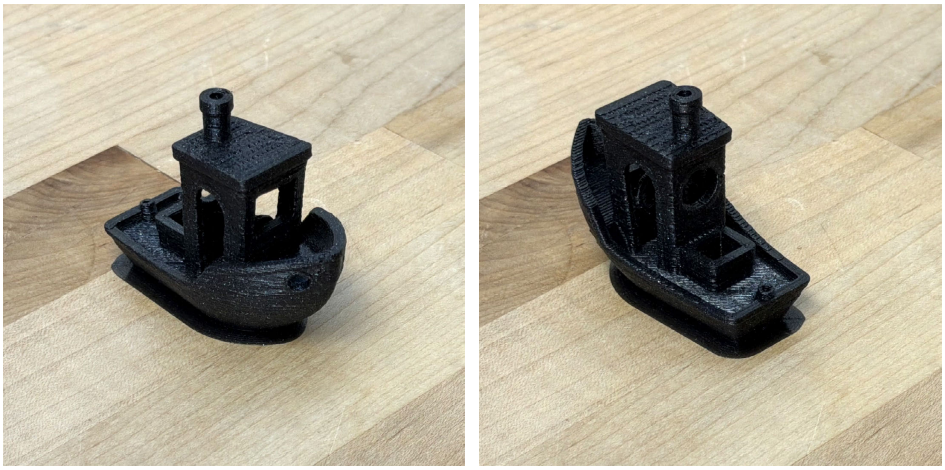


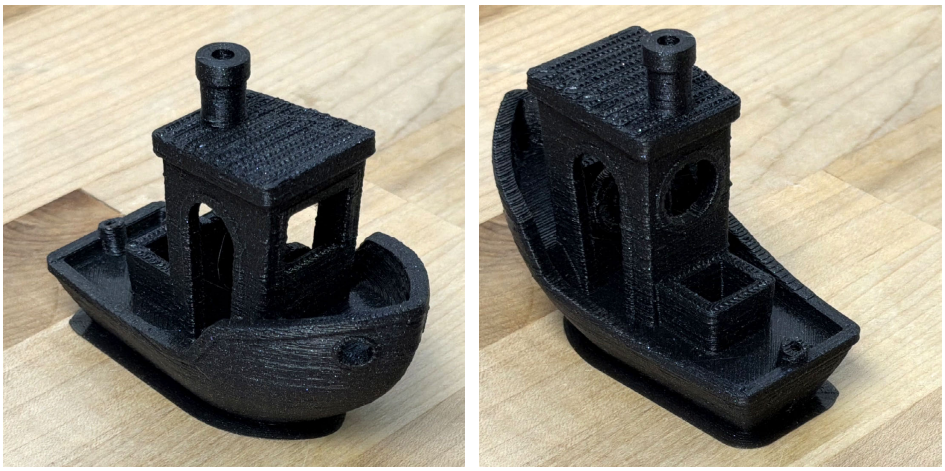
Figure 5.34: Flow Models (left: Steady-State, right: Isothermal) for PLA, using a 0.6mm Brass Nozzle.

PLA or *Polylactic Acid* is an extremely common 3D printing material that is derived from plant sugars. It is easy to print, has low warping, but is relatively brittle and is not resistant to chemical or UV exposure and softens at only 60° C.



Total Time	48.0*min*
Nozzle Temp	251 $^\circ C$
Avg Flowrate	4.27mm^3/s
Saturations:	
Cooling	44.88%
Flow & Heuristics	41.84%
Flow Dynamics	4.90%
Motion Dynamics	8.38%
Errors (avg)	
XY Positioning	0.063*mm*
XY Current Estimate	0.146*A*
Pressure Estimate	3.83*N*

Figure 5.35: 3DBenchy in PLA, using a 0.6mm Brass Nozzle.



Total Time	93.4*min*
Nozzle Temp	269 $^{\circ}C$
Avg Flowrate	6.75mm^3/s
Saturations:	
Cooling	28.24%
Flow & Heuristics	54.78%
Flow Dynamics	6.26%
Motion Dynamics	10.72%
Errors (avg)	
XY Positioning	0.064*mm*
XY Current Estimate	0.210*A*
Pressure Estimate	4.55*N*

Figure 5.36: 3DBenchy at 1.5x it's normal size in PLA, using a 0.6mm Brass Nozzle.

5.10.1.2 PETG, 0.6mm Brass Nozzle

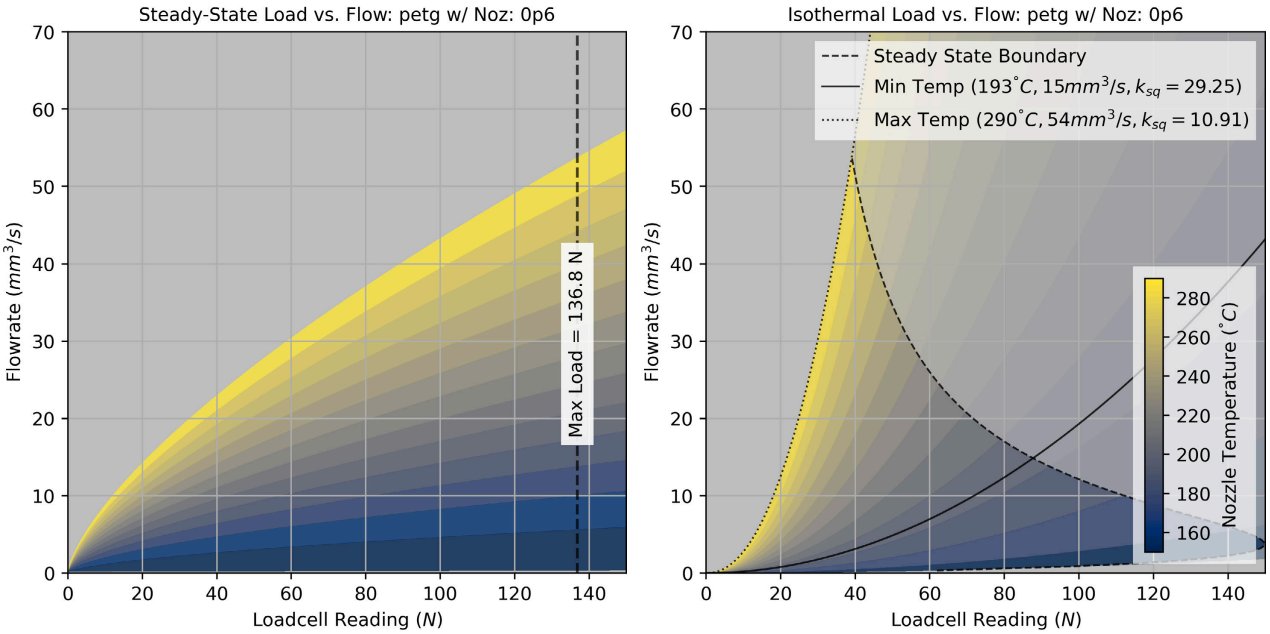
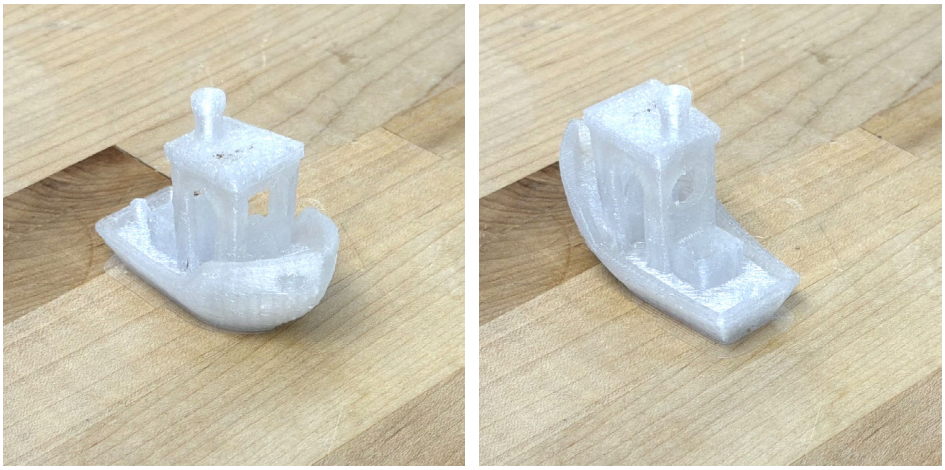


Figure 5.37: Flow Models (left: Steady-State, right: Isothermal) for PETG, using a 0.6mm Brass Nozzle.

PETG or *Polyethylene Terephthalate Glycol* is derived from the more common PET (which is what most single-use water bottles are made with), adding Glycol improves its printing performance and toughens it slightly. PETG has excellent inter-layer adhesion and provides a good balance between strength, flexibility and chemical resistance, and softens around 80° C.



Total Time	35.3*min*
Nozzle Temp	277 $^\circ C$
Avg Flowrate	5.80mm^3/s
Saturations:	
Cooling	43.28%
Flow & Heuristics	28.95%
Flow Dynamics	12.19%
Motion Dynamics	15.59%
Errors (avg)	
XY Positioning	0.068*mm*
XY Current Estimate	0.169*A*
Pressure Estimate	5.90*N*

Figure 5.38: 3DBenchy in PETG, using a 0.6mm Brass Nozzle.

5.10.1.3 PETG (with Carbon Fiber Fill), 0.4mm High Flow Nozzle

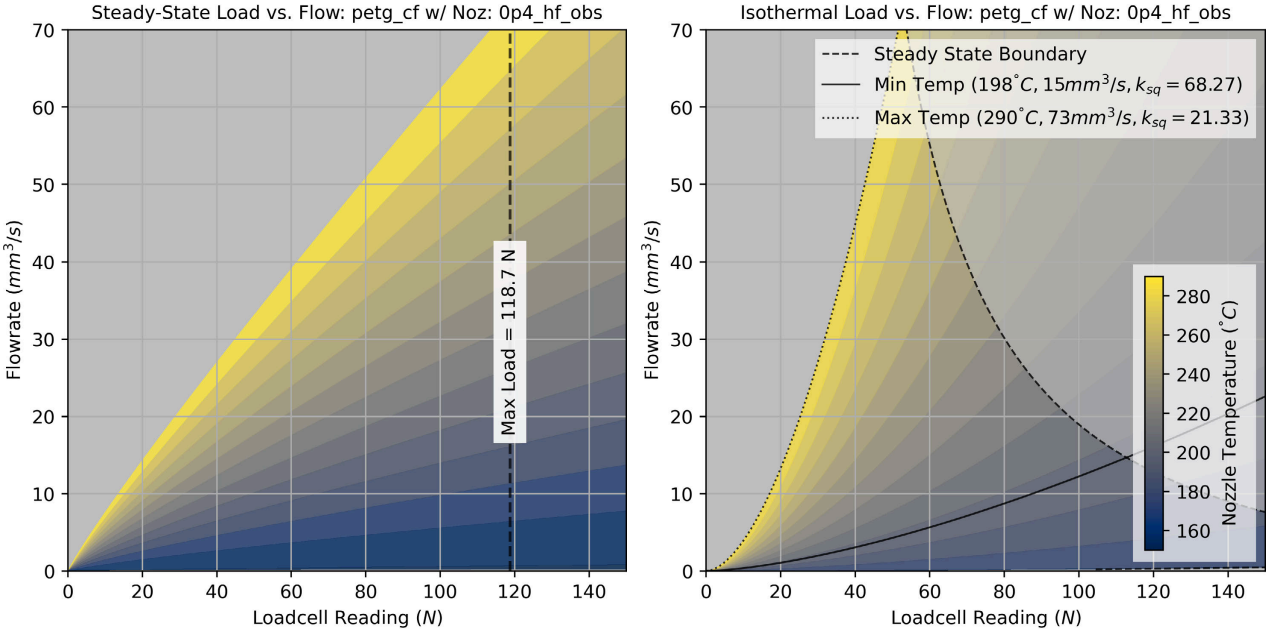
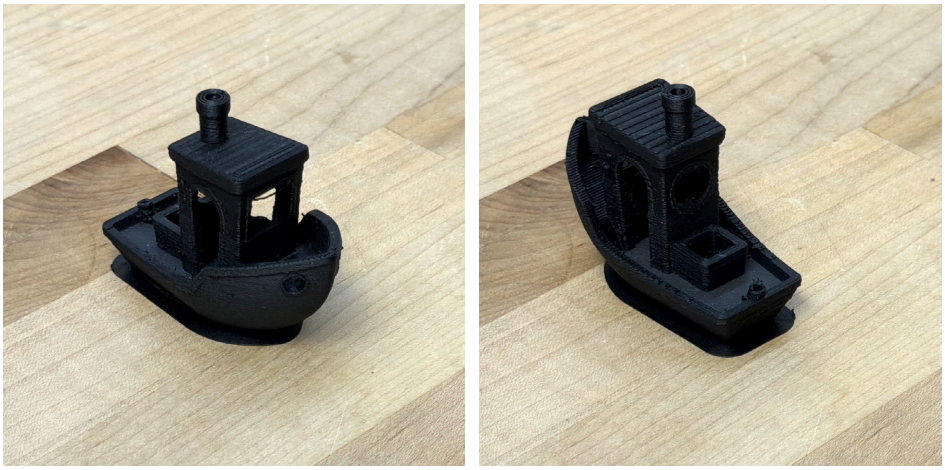


Figure 5.39: Flow Models (left: Steady-State, right: Isothermal) for PETG (Carbon Fiber Filled), using a 0.4mm High Flow Nozzle.

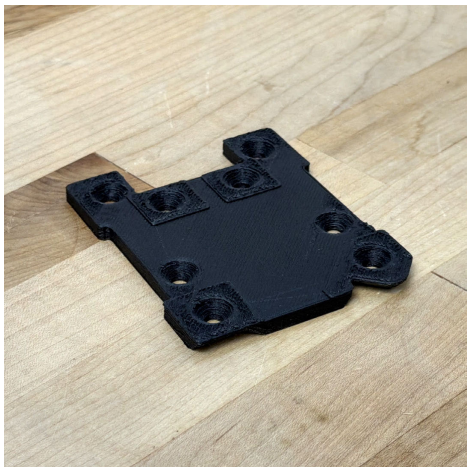
Adding Carbon Fiber fill to PETG increases its stiffness and decreases warping, but increases cost and requires that we use hardened nozzles, as the carbon fibers can abrade soft brass. Due to PETG's excellent layer adhesion, this material is a good basis for i.e. mechanical prototypes.



Total Time	44.4*min*
Nozzle Temp	213 $^\circ C$
Avg Flowrate	4.61mm^3/s
Saturations:	
Cooling	49.98%
Flow & Heuristics	36.13%
Flow Dynamics	8.43%
Motion Dynamics	5.46%
Errors (avg)	
XY Positioning	0.065*mm*
XY Current Estimate	0.141*A*
Pressure Estimate	24.45*N*

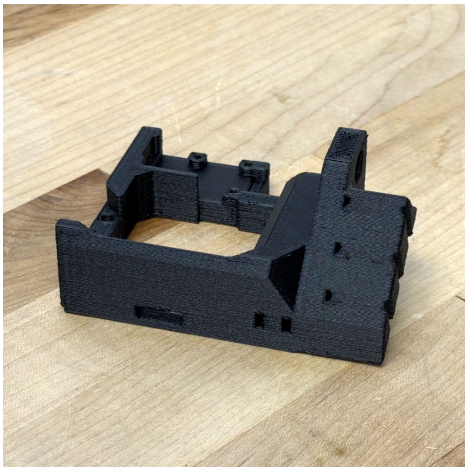
Figure 5.40: 3DBenchy in PETG with Carbon Fiber Fill, printed using a 0.4mm High Flow Nozzle. Geometry for this part was mistakenly generated for a 0.6mm nozzle.

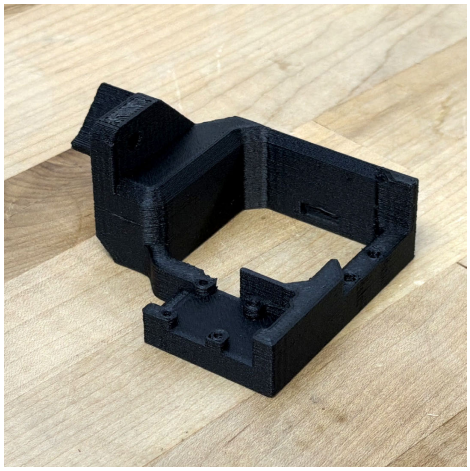




Total Time	25.4*min*
Nozzle Temp	279 ˚*C*
Avg Flowrate	10.82mm^3/s
Saturations:	
Cooling	11.73%
Flow & Heuristics	36.82%
Flow Dynamics	45.61%
Motion Dynamics	5.84%
Errors (avg)	
XY Positioning	0.062*mm*
XY Current Estimate	0.171*A*
Pressure Estimate	11.71*N*

Figure 5.41: The Kleat in PETG with Carbon Fiber Fill, printed using a 0.4mm High Flow Nozzle. Geometry for this part was mistakenly generated for a 0.6mm nozzle.





Total Time	64.7*min*
Nozzle Temp	257 ˚*C*
Avg Flowrate	9.04mm^3/s
Saturations:	
Cooling	39.77%
Flow & Heuristics	33.80%
Flow Dynamics	21.08%
Motion Dynamics	5.35%
Errors (avg)	
XY Positioning	0.069*mm*
XY Current Estimate	0.167*A*
Pressure Estimate	6.03*N*

Figure 5.42: A Circuit Mounting Component in PETG with Carbon Fiber Fill, printed using a 0.4mm High Flow Nozzle. Geometry for this part was mistakenly generated for a 0.6mm nozzle.

5.10.1.4 ASA (with Glass Fill), 0.4mm High Flow Nozzle

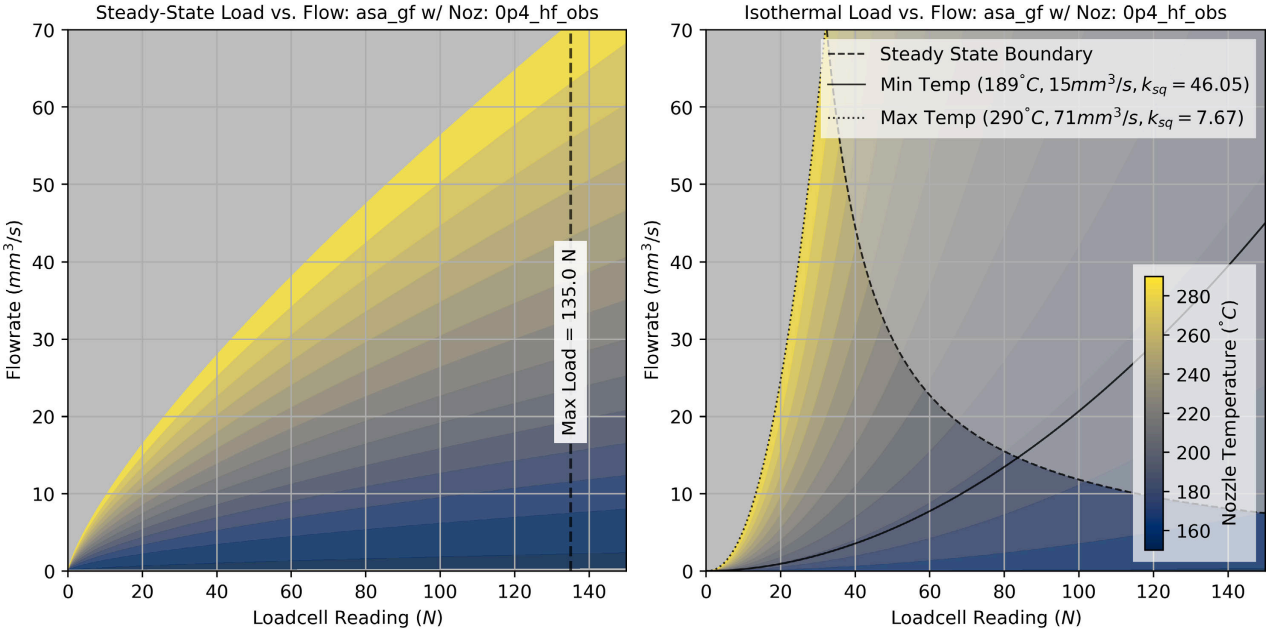
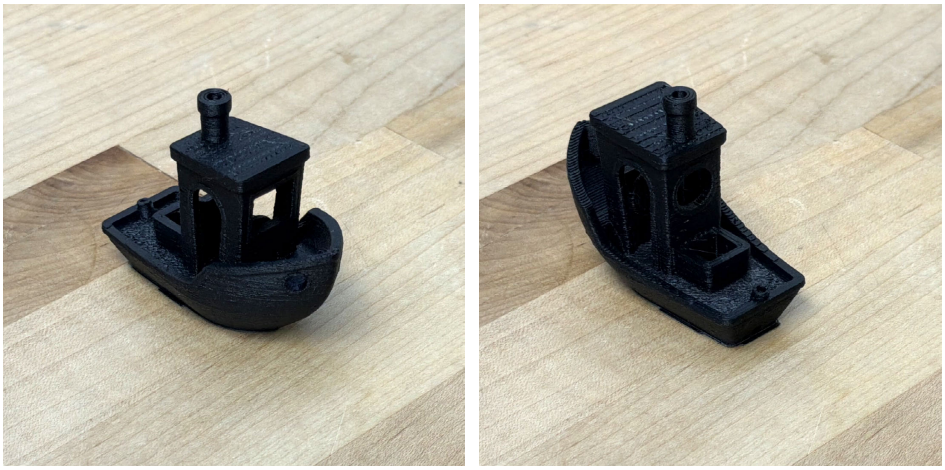


Figure 5.43: Flow Models (left: Steady-State, right: Isothermal) for ASA, using a 0.4mm High Flow Nozzle.

ASA or *Acrylonitrile Styrene Acrylate* is billed as an update to ABS *(Acrylonitrile Butadiene Acrylate),* it is UV stable and has similar (excellent) mechanical properties but warps less, which makes it more suitable for 3D printing.



Total Time	44.8*min*
Nozzle Temp	219 $^\circ C$
Avg Flowrate	4.57mm^3/s
Saturations:	
Cooling	46.27%
Flow & Heuristics	41.57%
Flow Dynamics	6.50%
Motion Dynamics	5.67%
Errors (avg)	
XY Positioning	0.063*mm*
XY Current Estimate	0.135*A*
Pressure Estimate	8.92*N*

Figure 5.44: 3DBenchy in ASA with Glass Fiber Fill, printed using a 0.4mm High Flow Nozzle. Geometry for this part was mistakenly generated for a 0.6mm nozzle.





Total Time	142.3*min*
Nozzle Temp	279 $^{\circ}C$
Avg Flowrate	11.74mm^3/s
Saturations:	
Cooling	22.80%
Flow & Heuristics	59.11%
Flow Dynamics	10.62%
Motion Dynamics	7.47%
Errors (avg)	
XY Positioning	0.076*mm*
XY Current Estimate	0.310*A*
Pressure Estimate	14.52*N*

Figure 5.45: A Brake Duct in ASA with Glass Fiber Fill, printed using a 0.4mm High Flow Nozzle. Geometry for this part was mistakenly generated for a 0.6mm nozzle.

5.10.1.5 Nylon (with Glass Fill), 0.6mm High Flow Nozzle

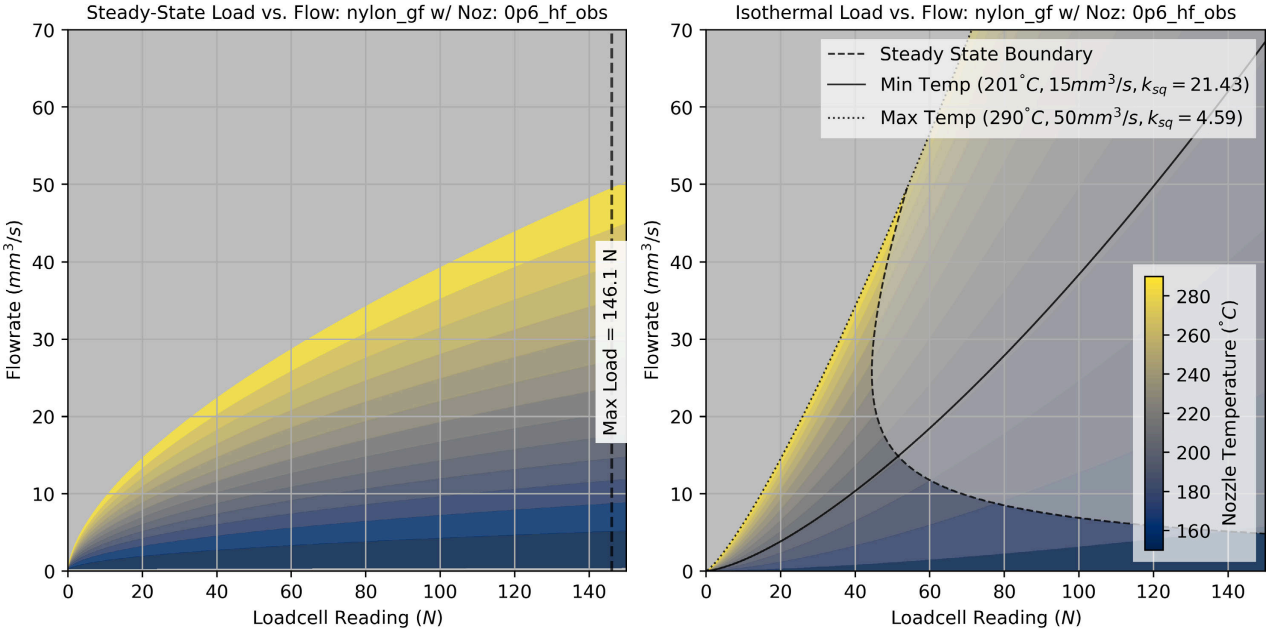
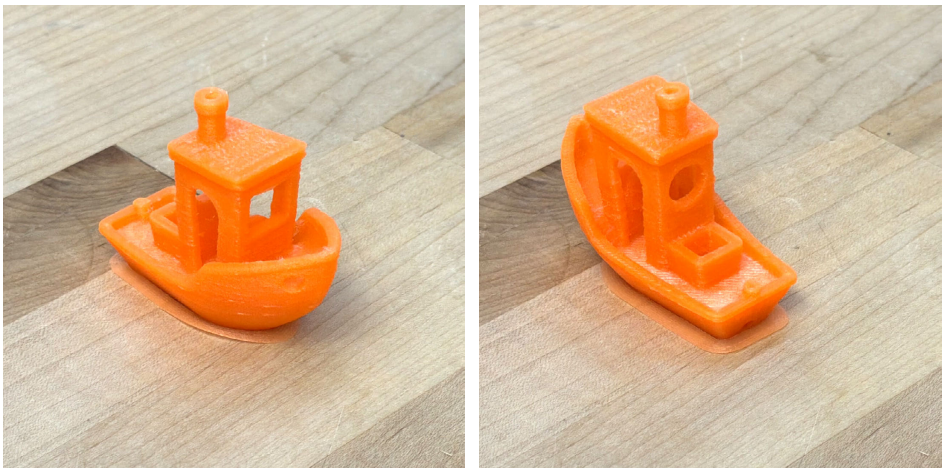


Figure 5.46: Flow Models (left: Steady-State, right: Isothermal) for Nylon (Glass Filled), using a 0.6mm High Flow Nozzle.

Nylon is an exceptional engineering polymer, used extensively in industry to produce i.e. power tool chassis, car parts, etc. It is normally combined with glass fill (as is this material) to improve stiffness and toughness. It has good wear resistance and fatigue life but is highly hygroscopic (it absorbs a lot of water from the air) so it must be *dried* before printing, which I did in order to process the material. It is difficult to print mostly because of the water absorption (the water boils as the material is heated, which causes obvious problems), but also requires a material-specific build plate - I used a phenolic composite bed. Its difficulty means that it is uncommon in 3D printing and so parameter presets are less developed for it than for other plastics, this was one of the materials for which there was no state-of-the-art preset available.



Total Time	40.8*min*
Nozzle Temp	230 $^\circ C$
Avg Flowrate	5.03mm^3/s
Saturations:	
Cooling	47.62%
Flow & Heuristics	27.02%
Flow Dynamics	18.09%
Motion Dynamics	7.27%
Errors (avg)	
XY Positioning	0.063*mm*
XY Current Estimate	0.141*A*
Pressure Estimate	5.61*N*

Figure 5.47: 3DBenchy in Nylon (PA6) with Glass Fiber Fill, printed using a 0.6mm High Flow Nozzle.





Total Time	14.5*min*
Nozzle Temp	268 ˚*C*
Avg Flowrate	9.13mm^3/s
Saturations:	
Cooling	33.27%
Flow & Heuristics	20.32%
Flow Dynamics	32.50%
Motion Dynamics	13.91%
Errors (avg)	
XY Positioning	0.068*mm*
XY Current Estimate	0.175*A*
Pressure Estimate	7.21*N*

Figure 5.48: A BMW E36 Hood Latch Handle in Nylon (PA6) with Glass Fiber Fill, printed using a 0.6mm High Flow Nozzle.





Total Time	38.7*min*
Nozzle Temp	238 ˚*C*
Avg Flowrate	6.40mm^3/s
Saturations:	
Cooling	39.01%
Flow & Heuristics	38.65%
Flow Dynamics	11.72%
Motion Dynamics	10.61%
Errors (avg)	
XY Positioning	0.060*mm*
XY Current Estimate	0.148*A*
Pressure Estimate	5.41*N*

Figure 5.49: A machine's motor mount in Nylon (PA6) with Glass Fiber Fill, printed using a 0.6mm High Flow Nozzle.

5.10.1.6 PCBlend (with Carbon Fiber), 0.6mm High Flow Nozzle

(Polycarbonate + ABS)

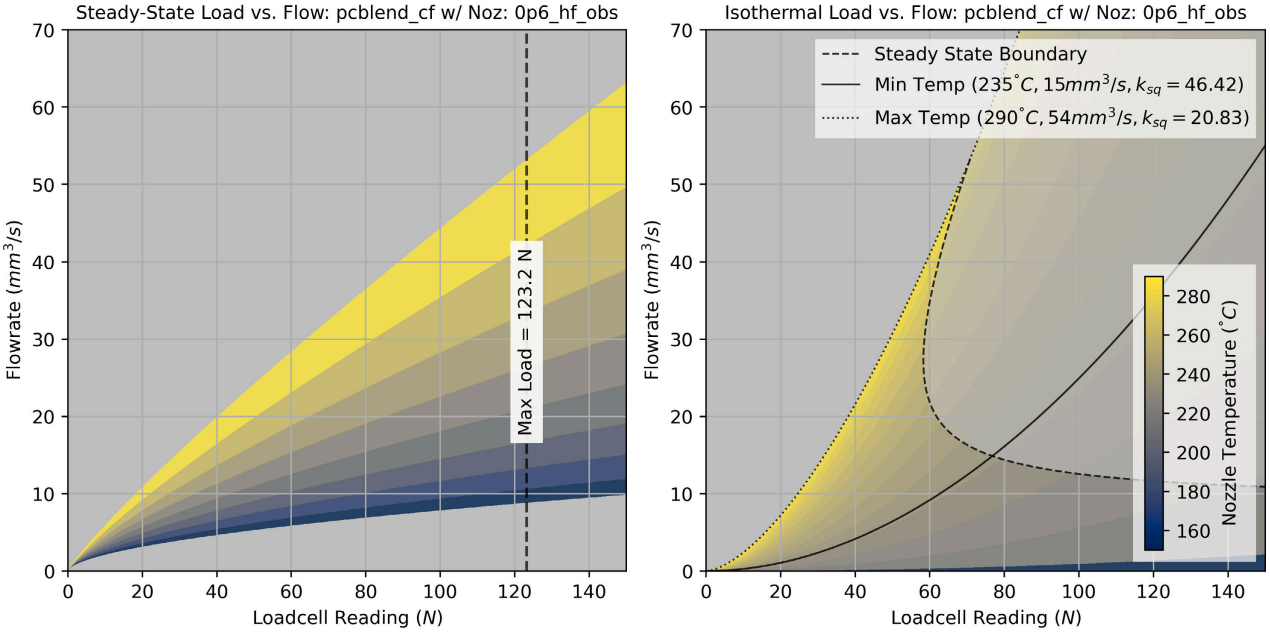
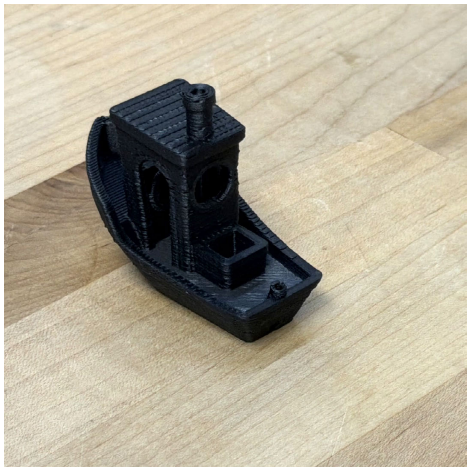


Figure 5.50: Flow Models (left: Steady-State, right: Isothermal) for PCBlend (Carbon Fiber Filled), using a 0.6mm High Flow Nozzle.

PCBlend is Polycarbonate with some unspecified additive [290] - although normally it is ABS. PC on its own is an excellent engineering material with high stiffness and toughness, but is also highly hygroscopic and warps a great deal; the blend is added to ameliorate both issues. It has excellent heat resistance, around 113° C.



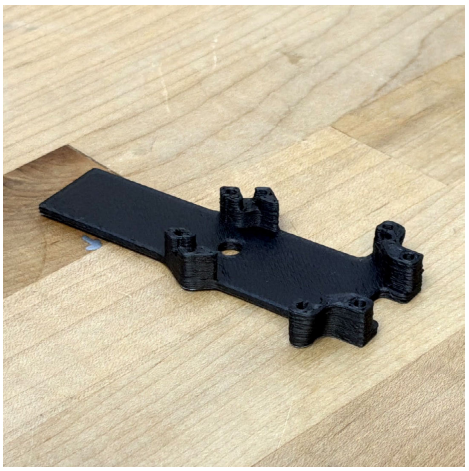


Total Time	38.3*min*
Nozzle Temp	261 $^\circ C$
Avg Flowrate	5.36mm^3/s
Saturations:	
Cooling	49.41%
Flow & Heuristics	33.36%
Flow Dynamics	5.79%
Motion Dynamics	11.45%
Errors (avg)	
XY Positioning	0.067*mm*
XY Current Estimate	0.164*A*
Pressure Estimate	7.10*N*

Figure 5.51: 3DBenchy in PCBlend with Carbon Fiber Fill, using a 0.6mm High Flow Nozzle.



Figure 5.52: An FPV Racing Drone, printed using this workflow in PCBlend with Carbon Fiber Fill on a 0.6mm High Flow Nozzle.





Total Time	15.4*min*
Nozzle Temp	289 $^{\circ}C$
Avg Flowrate	6.34mm^3/s
Saturations:	
Cooling	40.35%
Flow & Heuristics	32.29%
Flow Dynamics	9.59%
Motion Dynamics	17.77%
Errors (avg)	
XY Positioning	0.063*mm*
XY Current Estimate	0.179*A*
Pressure Estimate	6.41*N*

Figure 5.53: Drone Fuselage Lower Cap in PCBlend with Carbon Fiber Fill, using a 0.6mm High Flow Nozzle.

5.10.1.7 Timberfill, 0.4mm High Flow Nozzle

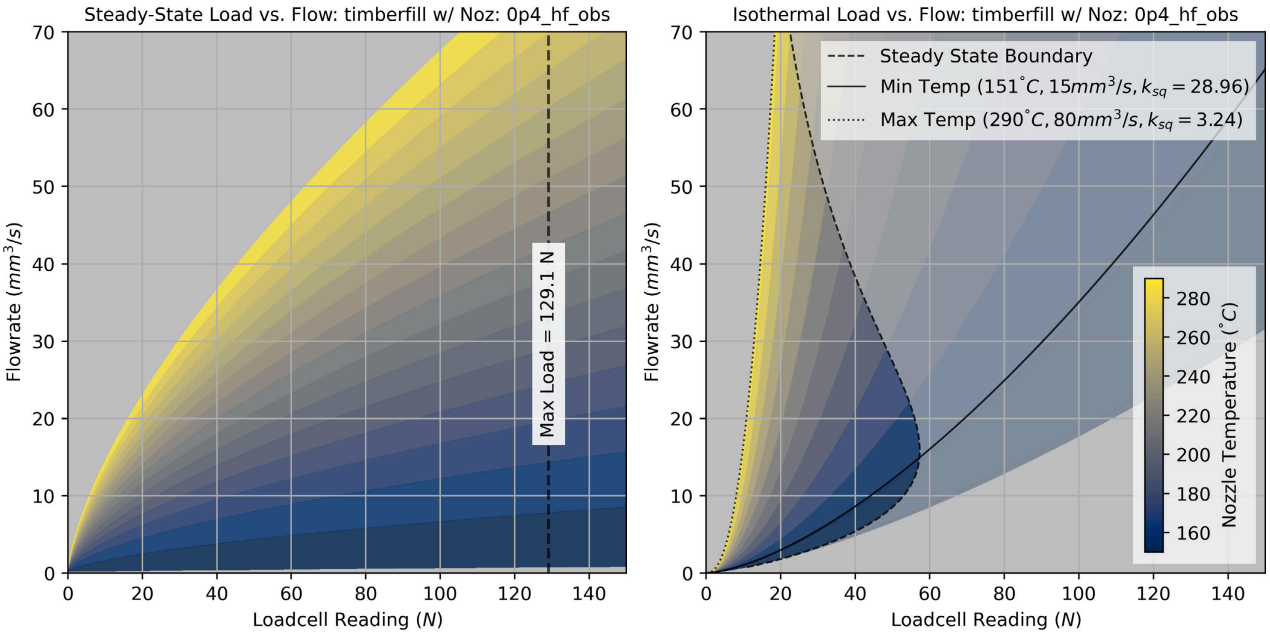


Figure 5.54: Flow Models (left: Steady-State, right: Isothermal) for Timberfill, using a 0.4mm High Flow Nozzle.

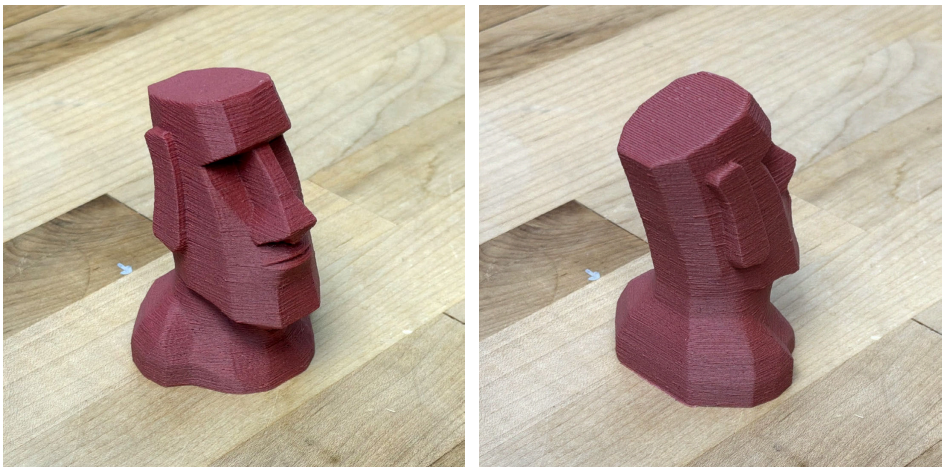
This is a material developed by Fillamentum [331] that is "similar to PLA" and includes natural fibers that have been "isolated from spruce" (i.e., sawdust). It is biodegradable and designed primarily for use in household objects.





Total Time	43.9*min*
Nozzle Temp	166 $^\circ C$
Avg Flowrate	4.67mm^3/s
Saturations:	
Cooling	42.38%
Flow & Heuristics	41.39%
Flow Dynamics	10.32%
Motion Dynamics	5.91%
Errors (avg)	
XY Positioning	0.065*mm*
XY Current Estimate	0.141*A*
Pressure Estimate	13.40*N*

Figure 5.55: 3DBenchy in Timberfill, printed using a 0.4mm High Flow Nozzle. Geometry was mistakenly generated for a 0.6mm nozzle, see -Section 5.11.6.



Total Time	87.1*min*
Nozzle Temp	184 ˚*C*
Avg Flowrate	9.06mm^3/s
Saturations:	
Cooling	41.97%
Flow & Heuristics	43.60%
Flow Dynamics	11.65%
Motion Dynamics	2.78%
Errors (avg)	
XY Positioning	0.060*mm*
XY Current Estimate	0.162*A*
Pressure Estimate	8.83*N*

Figure 5.56: An Easter Island Statue in Timberfill, printed using a 0.4mm High Flow Nozzle. Geometry was mistakenly generated for a 0.6mm nozzle, see -Section 5.11.6.



Total Time	90.3*min*
Nozzle Temp	248 ˚*C*
Avg Flowrate	13.68mm^3/s
Saturations:	
Cooling	6.78%
Flow & Heuristics	66.59%
Flow Dynamics	22.26%
Motion Dynamics	4.36%
Errors (avg)	
XY Positioning	0.079*mm*
XY Current Estimate	0.256*A*
Pressure Estimate	5.87*N*

Figure 5.57: A kitchen knife case in Timberfill, printed using a 0.4mm High Flow Nozzle. Geometry was mistakenly generated for a 0.6mm nozzle, see -Section 5.11.6.

5.10.1.8 AluFill (67% 6061 Aluminum + Binder), 0.6mm High Flow Nozzle

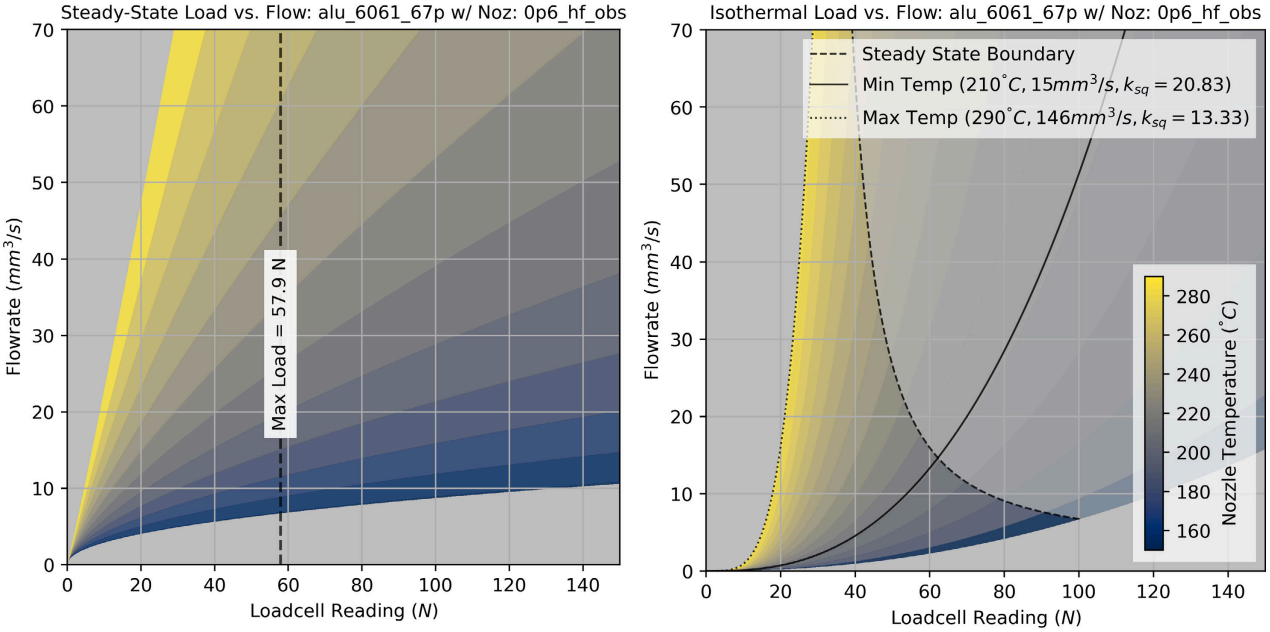
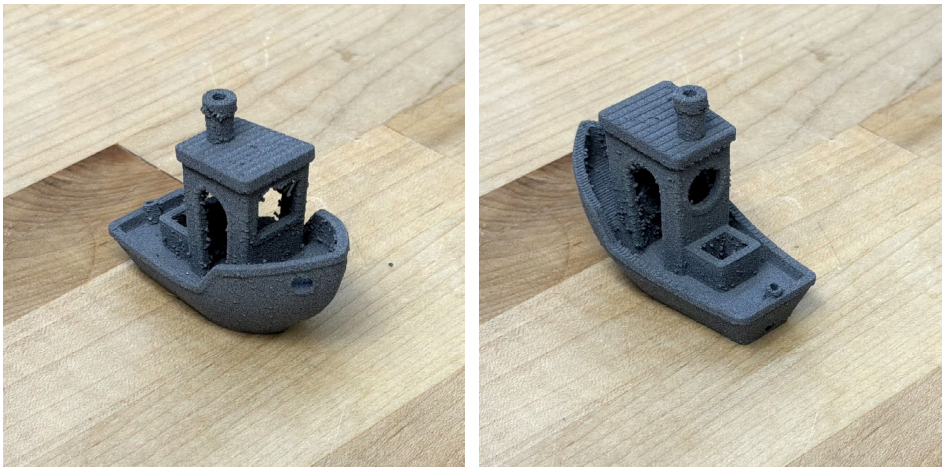


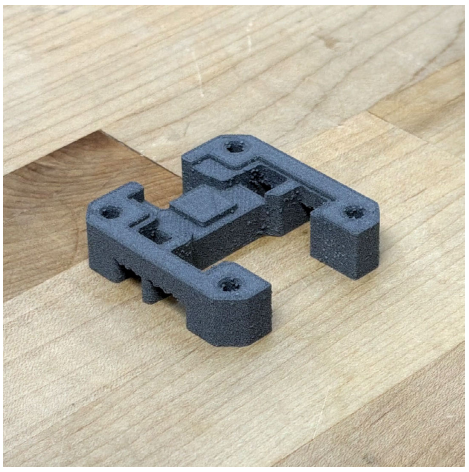
Figure 5.58: Flow Models (left: Steady-State, right: Isothermal) for AluFill, using a 0.6mm High Flow Nozzle.

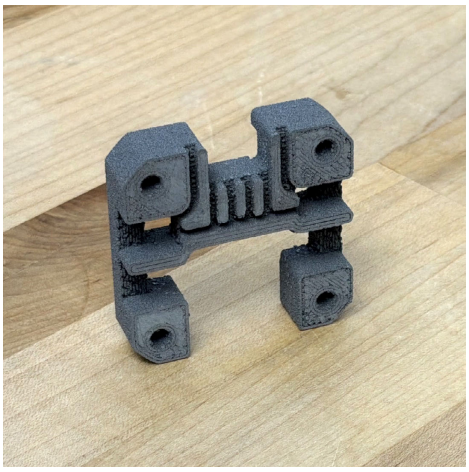
AluFill from The Virtual Foundry [332] is designed to be sintered post-printing to form a solid metal part. They claim 67% aluminum content but do not disclose the binder material and merely specify that it prints "similar to PLA." Materials like this are challenging to print because they lack internal cohesion — they can act closer to tightly packed sand than to a polymer — and so extruder drive gears can easily shear them. In other printer configurations, articulating that the extruder should exert *less force overall* is difficult because we tune extruder motor *velocity and acceleration* parameters, but not force. In this case I de-rated the extruder motor's current limit (from $0.7A \rightarrow 0.3A$) in both the flow model test and in printing. This prevents the extruder from exerting too much force across all dynamical states (while extruding, retracting, etc), and is easily captured in the model that relays this limit to the planner.



Total Time	85.1*min*
Nozzle Temp	246 $^\circ C$
Avg Flowrate	2.41mm^3/s
Saturations:	
Cooling	11.10%
Flow & Heuristics	70.34%
Flow Dynamics	14.13%
Motion Dynamics	4.44%
Errors (avg)	
XY Positioning	0.056*mm*
XY Current Estimate	0.153*A*
Pressure Estimate	8.75*N*

Figure 5.59: 3DBenchy in AluFill (67%), printed using a 0.6mm High Flow Nozzle.





Total Time	83.8*min*
Nozzle Temp	246 $^{\circ}C$
Avg Flowrate	2.88mm^3/s
Saturations:	
Cooling	2.16%
Flow & Heuristics	76.69%
Flow Dynamics	18.23%
Motion Dynamics	2.92%
Errors (avg)	
XY Positioning	0.049*mm*
XY Current Estimate	0.160*A*
Pressure Estimate	8.19*N*

Figure 5.60: A motor controller circuit enclosure / heatsink in AluFill (67%), printed using a 0.6mm High Flow Nozzle. Printed at 1.18x original size, to sinter to 1.0x.

Material	Noz.	$T_{\min}$ (°C)	$Q_{\max}$ (mm^3/s)	k_{sq}@220° C (N/mm^3)	k_{sq}@290° C (N/mm^3)
Timberfill	0.4 HF	151.2	80.3	16.21	3.24
PLA	0.6	181.1	54.0	25.83	9.19
PLA	0.6 HF	175.0	69.3	26.66	7.89
PETG	0.6	193.4	53.9	24.20	10.91
PETG	0.6 HF	200.5	57.7	32.82	12.41
PETG CF	0.4 HF	197.7	73.1	56.91	21.33
PETG CF	0.6 HF	188.0	77.5	42.44	16.92
ASA	0.6 HF	193.8	58.4	28.19	7.32
ASA GF	0.4 HF	188.9	71.1	34.24	7.67
PCBlend CF	0.6 HF	235.4	53.5	53.61	20.83
Nylon GF	0.6 HF	201.5	49.7	17.90	4.59
AluFill	0.6 HF	209.9	146.4	19.88	13.33

Table 5.11: A complete table of flow models generated, including key model parameters.

5.10.2 Tables of FFF Prints

5.10.2.1 Complete Table of Flow Models

Key parameters from each flow model that I generated are in Table 5.11. Here, a flow model's Minimum Temperature $T_{\min}$ is the temperature at which Maximum Flowrate is $15mm^3/s$, the maximum flowrate ($Q_{\max}$) is taken at 290° C (the nozzle's maximum safe temperature), and the spring rate (k_{sq}) is shown at both 220 ° C and 290 ° C to show variance across temperatures

5.10.2.2 Complete Table of Prints

Tables with key values from all the prints completed using the feedback-based workflow are in Table 5.12 (prints that were compared to the state-of-the-art workflow) and Table 5.13 (prints that were not compared). These show the nozzle temperature selection, average flowrate, total print time t, and saturation values that describe the percentage of print time in each part that was dominated by one of four physical constraints: s_{cool} (cooling limited), s_{flow} (flowrate / flow heuristics limited), s_{extr} (extruder dynamics limited), and s_{motn} (motion system dynamics limited). Section 5.11.1 describes how these saturations are produced using the planner's outputs.

Mat.	Noz.	Geometry	Sys.	T_{noz}	Q_{avg}	t	s_{cool}	s_{flow}	s_{extr}	s_{motn}
				$^\circ$ C	mm^3/s	$min.$	%	%	%	%
PLA	0.6	3DBenchy	Ours	251	4.27	48.0	44.9	41.8	4.9	8.4
			SOTA	230	4.02	53.0				
PLA	0.6	3DBenchy 1.50x	Ours	269	6.75	93.4	28.2	54.8	6.3	10.7
			SOTA	230	6.37	104.0				
PETG	0.6	3DBenchy	Ours	277	5.80	35.3	43.3	29.0	12.2	15.6
			SOTA	250	3.67	58.0				
PETG	0.6	Roadballs	Ours	289	11.03	49.3	5.9	50.0	20.0	24.1
			SOTA	250	8.50	68.0				
PCBlend CF	0.6 HF	3DBenchy	Ours	289	5.36	38.3	49.4	33.4	5.8	11.5
			SOTA	280	3.26	67.0				
PCBlend CF	0.6 HF	Drone Cover	Ours	289	6.34	15.4	40.4	32.3	9.6	17.8
			SOTA	280	4.43	23.0				
PCBlend CF	0.6 HF	Drone Arms	Ours	289	9.25	60.4	3.11	45.5	19.8	31.6
			SOTA	280	6.72	80.0				
PCBlend CF	0.6 HF	Drone Fuselage	Ours	289	10.24	65.1	2.9	54.5	12.6	30.0
			SOTA	280	7.65	92.0				

Table 5.12: A table of prints completed using the feedback-based workflow where state-of-the-art comparisons were completed.

Mat.	Noz.	Geometry	Sys.	T_{noz}	Q_{avg}	t	s_{cool}	s_{flow}	s_{extr}	s_{motn}
				$^\circ$ C	mm^3/s	$min.$	%	%	%	%
PETG CF	0.4 HF	3DBenchy[a]	Ours	213	4.61	44.4	50.0	36.1	8.4	5.5
PETG CF	0.4 HF	Kleat[a]	Ours	279	10.82	25.4	11.7	36.8	45.6	5.8
PETG CF	0.4 HF	Circuit Mount[a]	Ours	257	9.04	64.7	39.8	33.8	21.8	5.6
ASA GF[d]	0.4 HF	3DBenchy[a]	Ours	219	4.57	44.8	46.3	41.6	6.5	5.7
ASA GF	0.4 HF	Brake Duct[b]	Ours	279	11.74	142.3	22.8	59.1	10.6	7.5
Nylon GF[d]	0.6 HF	3DBenchy	Ours	230	5.03	40.8	47.6	27.0	18.1	7.3
Nylon GF	0.6 HF	Hood Latch	Ours	268	9.13	14.5	33.3	20.3	32.5	13.9
Nylon GF	0.6 HF	Motor Mount	Ours	238	6.40	38.7	39.0	38.7	11.7	10.6
Timberfill	0.4 HF	3DBenchy[a]	Ours	166	4.67	43.9	42.4	41.4	10.3	5.9
Timberfill	0.4 HF	Moai[a]	Ours	184	9.06	87.1	42.0	43.6	11.7	2.8
Timberfill	0.4 HF	Knife Holder[a]	Ours	248	13.68	90.3	6.8	66.5	22.3	4.4
AluFill[c,d]	0.6 HF	3DBenchy	Ours	246	2.41	85.1	11.1	70.3	14.1	4.4
AluFill	0.6 HF	Circuit Heatsink	Ours	238	2.29	59.4	2.9	77.9	16.5	2.7
AluFill	0.6 HF	Circuit Heatsink 1.18x	Ours	246	2.88	83.8	2.2	76.7	18.2	2.9

Table 5.13: A table of prints completed using the feedback-based workflow, where state-of-the-art comparisons were not completed or not possible.

[a] These prints were mislabelled: I mistakenly inserted a *0.4mm* high flow nozzle, but thought I had a 0.6mm high-flow nozzle installed. This means that I printed them with geometry for a 0.6mm nozzle, rather than 0.4mm. However, the *flow models* were produced using the 0.4mm nozzle, so the parts still worked, *but* it means that I did not complete comparisons to the state-of-the-art for these parts. I discuss this error in Section 5.11.6.

[b] This part was printed *with a 0.6mm nozzle* re-installed, *but* the model used for flow was generated with the 0.4mm nozzle.

[c] As I noted in Section 5.10.1.8, this material was printed with a de-tuned extruder motor (limited current deployment), to avoid shearing the material.

[d] These materials were not compared to the state-of-the-art because no slicer presets were available for them at the time of writing.

5.10.3 Reducing Parameter Spaces

To quantitatively compare feedback-based operation to the state-of-the-art, we can compare the total number of human-tuneable values required in state-of-the-art presets to the values that this workflow exposes.

In the introduction of this section, I rendered tables of printing parameters taken from a state-of-the-art slicer (Prusa Slicer 2.9.4 (2024)), (Tables 5.1, 5.2, 5.3, and 5.4). That set contains 47 parameters, of which:

- 16 are either geometric or easily calculated based on nozzle width,
- 12 are translational velocities
- 7 are accelerations
- 3 are temperatures (nozzle, bed, chamber)
- 1 is directly filament flow related (maximum flowrate)
- 4 are retraction related

Our workflow does not modify the geometry generation step of the state-of-the-art, and does not set bed or chamber temperatures, so we are left with 29 free parameters to replace with the feedback-based workflow. It still relies on some human-in-the-loop selections and I lay out a table of those parameters in Table 5.5. It contains 21 heuristics or "manually" estimated values:

- 4 related to cooling
- 1 first layer rate
- 1 minimum flowrate
- 3 pressure scalars
- 8 motion scalars (current, and current slew rates)
- 4 extruder heuristics

In all, we have a similar total count of parameters. However, parameters in the state-of-the-art workflow should be modified by hand for every new material, nozzle, or machine. Our workflow uses models to mediate between this subset of heuristics and actual control inputs to the machine. While I was printing the parts above in Section 5.10.1, I dialled these heuristics in once for the machine (using PLA, over the course of about one day) and then left them alone for each new material or nozzle. Those values are the third column of Table 5.5. To be explicit, that means that tuning each of the print systems in Section 5.10.1 (11 nozzle and material combinations in total) from scratch in the state-of-the-art workflow would have involved tuning up to 319 individual parameters. In our workflow, we can tune 21 parameters on one material and nozzle, and use models to extend that work automatically across new materials and nozzles.

5.10.4 Print Speed vs. SOTA

In the small set of prints that were compared to the state-of-the-art, the workflow developed here increased print speed over on average by 141%. In Table 5.14 I have sorted these within their respective materials from smallest to largest speed-up and included our nozzle temperature selections vs. the state-of-the-art.

I expected there to be a rough correlation between print size and speed increase because one of the main advantages over the state-of-the-art is that our workflow can select hotter nozzle temperatures for larger parts (as in Section 5.8), but in this small set there is no clear relation. Geometry also plays a role here so e.g., tall and skinny parts will not see as strong of a speed-up as wide and flat parts.

Mat.	Noz.	Geometry	Total Vol.	Our Temp	SOTA Temp	Δ Temp	Our Time	SOTA Time	Δ Time
			mm^3	° C	° C	%	*min.*	*min.*	%
PLA									
PLA	0.6	3DBenchy	12773	251	230	109.1	48.0	53.0	110.4
PLA	0.6	3DBenchy 1.50x	39737	269	230	117.0	93.4	104.0	113.5
PETG									
PETG	0.6	Roadballs	34687	289	250	115.6	49.3	68.0	137.9
PETG	0.6	3DBenchy	12773	277	250	110.8	35.3	58.0	164.3
PCBlend CF									
PCBlend CF	0.6 HF	Drone Arms	32247	289	280	103.2	60.4	80.0	132.5
PCBlend CF	0.6 HF	Drone Halves	42233	289	280	103.2	65.1	92.0	141.3
PCBlend CF	0.6 HF	Drone Cover	6119	289	280	103.2	15.4	23.0	149.4
PCBlend CF	0.6 HF	3DBenchy	13125	289	280	103.2	38.3	67.0	175.0

Table 5.14: Table of prints that were compared to the state-of-the-art workflow, evaluating time optimality.

5.10.5 Qualitative Print Quality vs. SOTA

I did not develop a workflow to rigorously evaluate print quality against the state-of-the-art, but photographs below (our prints on the left, SOTA on the right in Figures 5.61, 5.62, 5.63, 5.64 and 5.65) provide a simple basis for qualitative assessment. Overall, prints using our controller have many more visual artefacts than the state-of-the-art. I discuss the alignment between those artefacts and modelling errors in Section 5.10.5.1.

5.10.5.1 Flow Modelling Errors Correlate with Part Errors

One promising result is that areas of prints that are clearly imprecise correlate well with areas where the planner's flow model predictions deviated from actual measured values. I show this in Figures 5.66 and 5.67: both include images of printed parts (at left) and plots (right) that map error between the planner's predicted pressure measurement and the load cell's real measurement. In Figure 5.66 patches of the part that have clear over-extrusion artefacts correlate to areas in the dataset where measured pressure was much higher (by about $50N$) than the planner predicted. Two slices of the time-series dataset from this part are in Figure 5.68, at left is an area where models did well to predict measurements, and at right is an area (corresponding to the lower-right corner of the part) where models did poorly. Figure 5.67 shows two over-extrusion artefacts on the printed drone arm, which correlate directly to two moments in the dataset where pressure was again much higher than the planner predicted.

This means that where measurements *do* correlate to predictions, the planner works well to operate the machine — if predictions deviated from measurements in areas where prints looked good, something would clearly be wrong with the method overall. If the planner is to be trusted, locations



Figure 5.61: The 3DBenchy in PLA, on a 0.6mm Brass Nozzle, left: SOTA, right: ours.

and process states where we see deviations from its predictions mean one of two things: *either* the models are wrong (which leads the planner to produce incorrect outputs and thus produces the observable errors) *or* that the process inputs changed unexpectedly. In the examples I show here it is almost certainly the former: the models are fit on a limited set of data from the tests performed in Section 5.5 which does not properly map through all the very many time-varying states that the printer operates within. For example all errors highlighted in Figure 5.66 take place where the machine is rapidly accelerating and decelerating at relatively high flowrates and nozzle

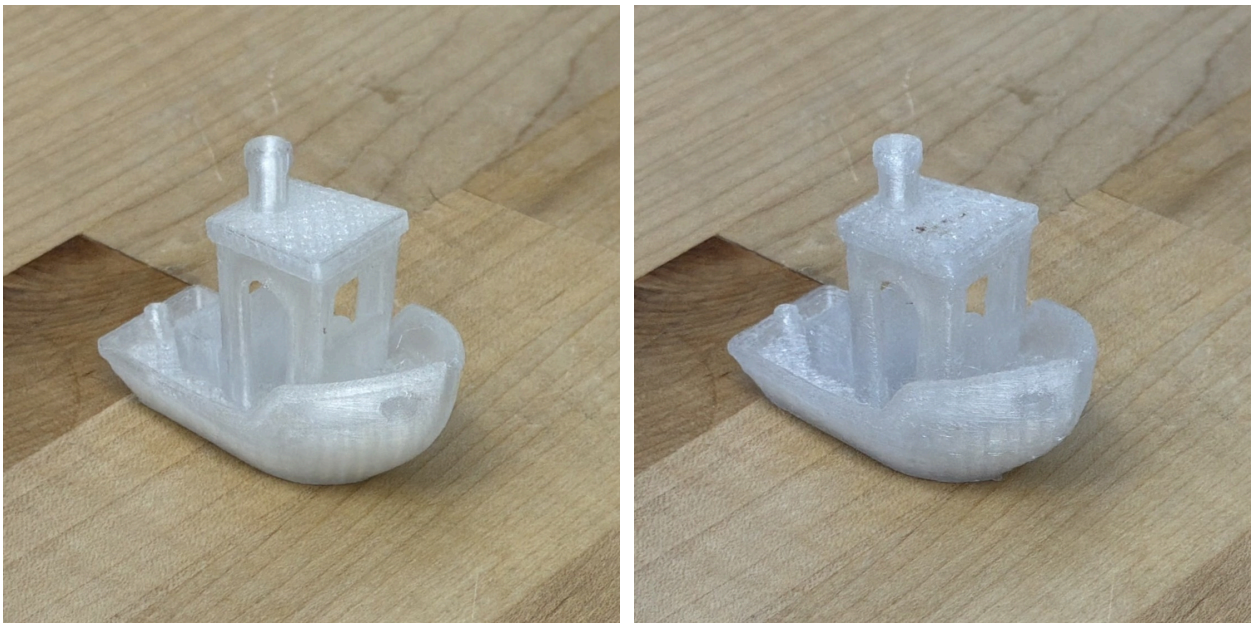


Figure 5.62: The 3DBenchy in PETG, on a 0.6mm Brass Nozzle, left: SOTA, right: ours.

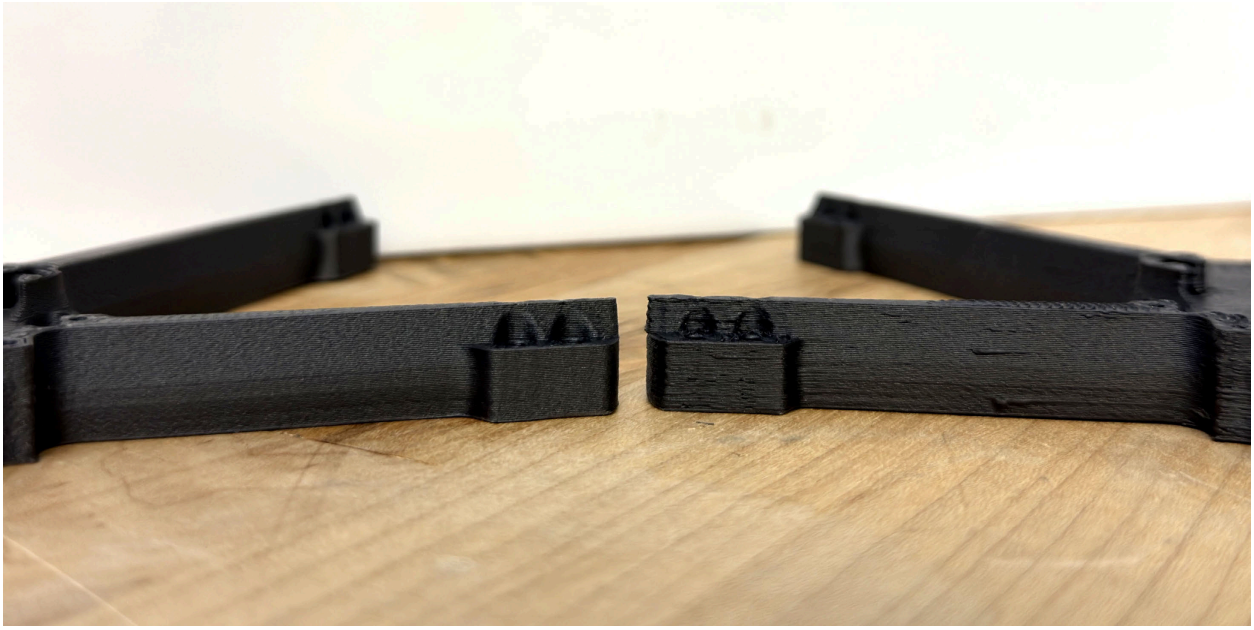


Figure 5.63: Arms of the drone, left: SOTA, right: ours.

temperatures and there are only about 30 seconds of total data in this regime that has been applied to the model fit (see the fitting method in Section 5.5.3).

In particular, I think that the time-varying dynamics are poorly fit because collecting data that spans all the underlying states is difficult and the approximation that I use to interpolate between the steady-state and isothermal models (in Section 5.5.4) is lossy. In the next Section 5.10.6 I will discuss this at more length.

5.10.6 Flow Model Fit Quality

In each print above, I rendered a small table that includes average errors for *XY Current* and *Pressure Estimate*. These are end-to-end evaluations of our modelling workflow, and they are made using time-series data collected from the printer's hardware and from the planner simultaneously. I look more closely at the XY Current error evaluations in the motion control chapter at Section 4.7.2. A table of model errors from each of these prints is here in Table 5.15, both take the mean of the absolute errors between the planner's predicted value and the measured values across all samples in the print time series.



Figure 5.64: Fuselage cap of the drone, left: SOTA, right: ours.

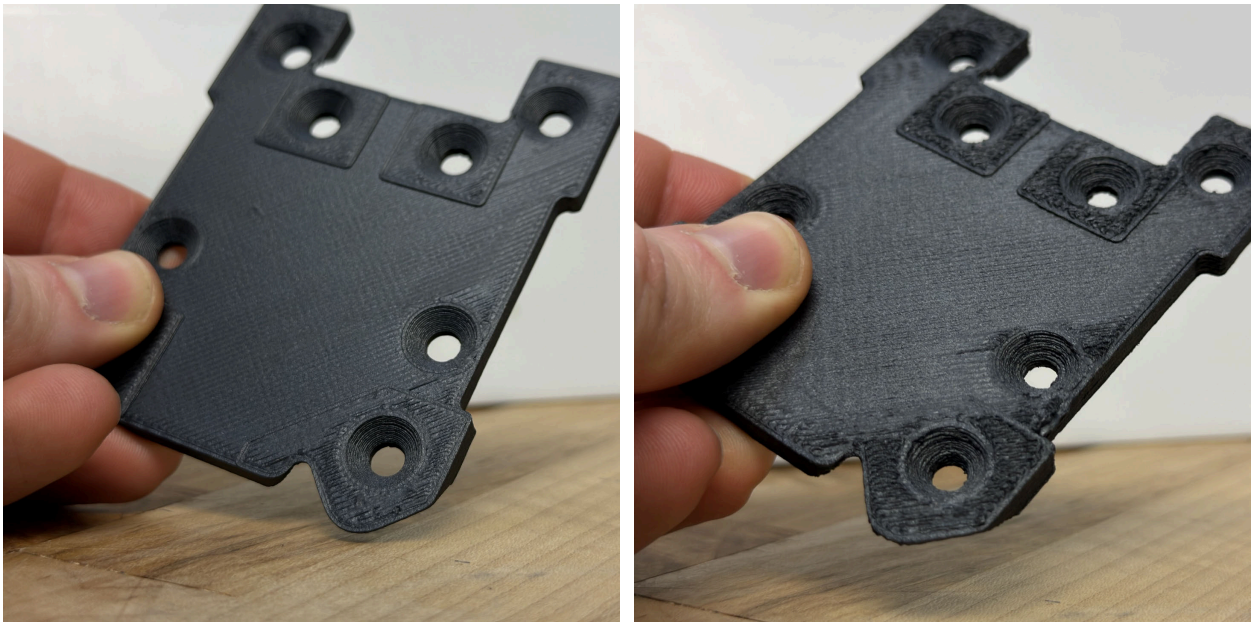


Figure 5.65: The Kleat in PETG with Carbon Fill, 0.4mm High Flow Nozzle. Left: SOTA, right: ours.

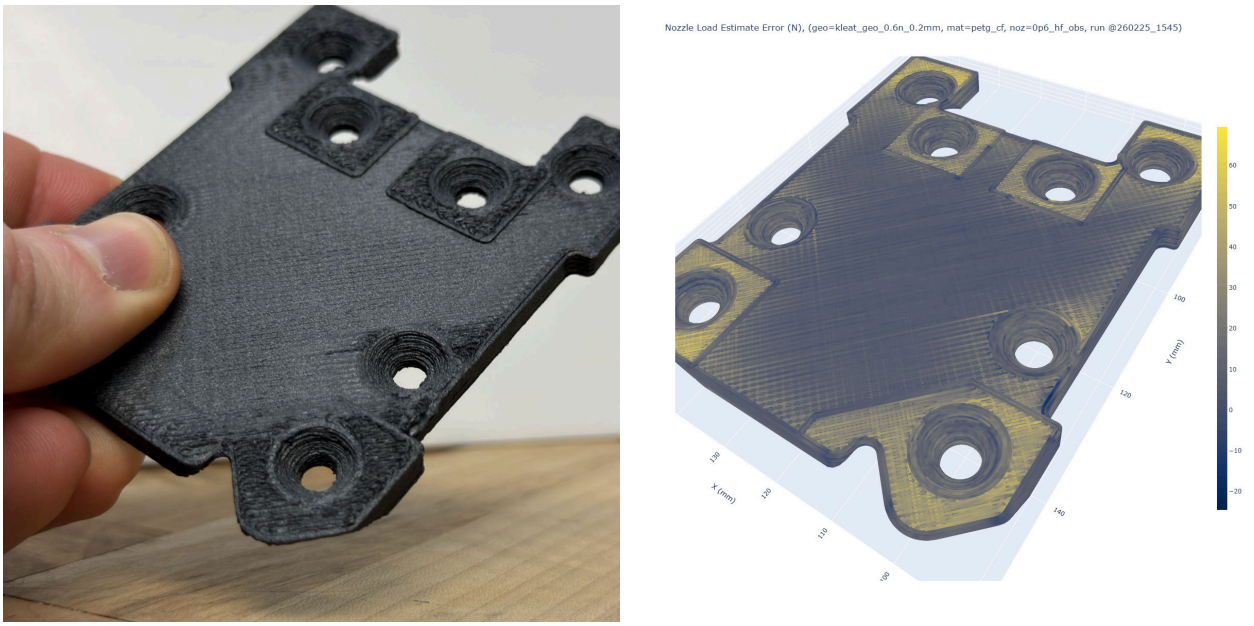


Figure 5.66: Our print of the Kleat in PETG with Carbon Fill using a 0.4mm High Flow Nozzle. Left: the print, showing errors in areas of short surface infill. At right, deviations from the planner's estimate of nozzle pressure, $err = F_{meas} - F_{slvr}$, with the color map scaling from -25 to 70 Newtons, blue to yellow. Figure 5.68 shows time-series data from two parts of this trajectory, one (left) where models fit well and the other (right) where the model fit poorly.

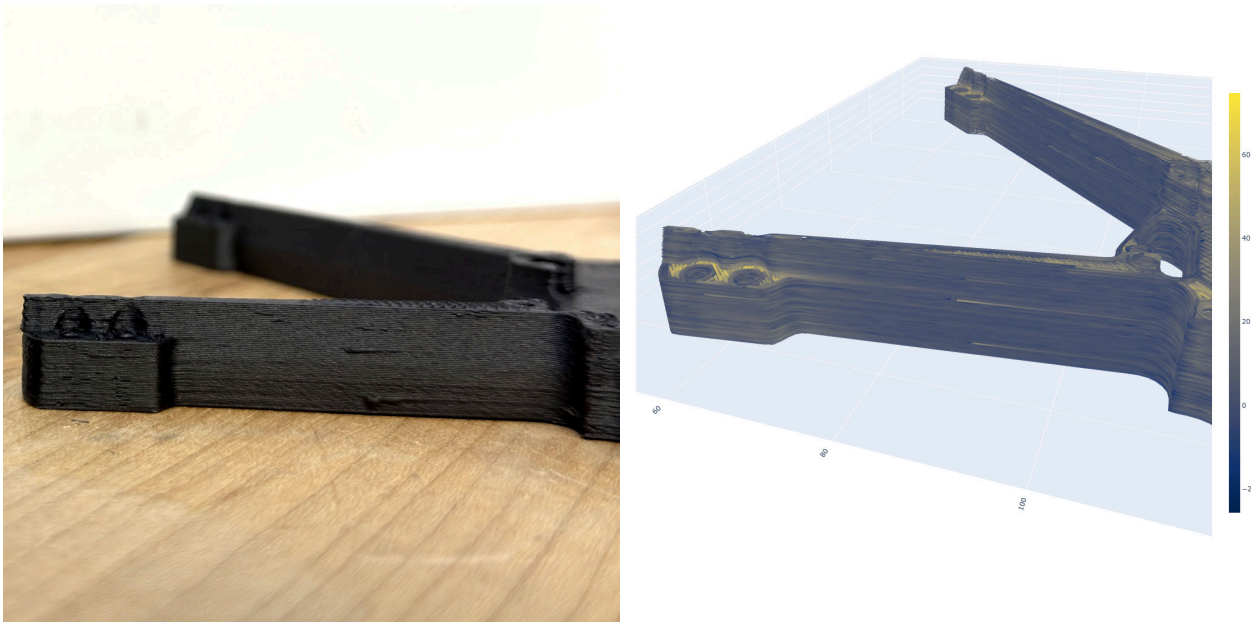


Figure 5.67: Our print of the Drone Arms in PCBlend with Carbon Fill using a 0.6mm High Flow Nozzle. Left: the print, showing over-extrusion errors. At right, deviations from the planner's estimate of nozzle pressure, $err = F_{meas} - F_{slvr}$, with the color map scaling from -3 to 70 Newtons, blue to yellow.

Mat.	Noz.	Geometry	T_{noz}	Q_{avg}	t	Err. XY Motion	Err. Pressure
			$^\circ$ C	mm^3/s	$min.$	A	N
PLA	0.6	3DBenchy	251	4.27	48.0	0.146	3.83
PLA	0.6	3DBenchy 1.50x	269	6.75	93.4	0.210	4.55
PETG	0.6	3DBenchy	277	5.80	35.3	0.169	5.90
PETG	0.6	Roadballs	289	11.03	49.3	0.176	6.02
PETG CF	0.4 HF	3DBenchy	213	4.61	44.4	0.141	24.45[a]
PETG CF	0.4 HF	Kleat	279	10.82	25.4	0.171	11.71
PETG CF	0.4 HF	Circuit Mount	257	9.04	64.7	0.167	6.03
ASA GF	0.4 HF	3DBenchy	219	4.57	44.8	0.135	8.92
ASA GF	0.4 HF	Brake Duct	279	11.74	142.3	0.310	14.52[b]
Nylon GF	0.6 HF	3DBenchy	230	5.03	40.8	0.141	5.61
Nylon GF	0.6 HF	Hood Latch	268	9.13	14.5	0.175	7.21
Nylon GF	0.6 HF	Motor Mount	238	6.40	38.7	0.148	5.41
PCBlend CF	0.6 HF	3DBenchy	289	5.36	38.3	0.164	7.10
PCBlend CF	0.6 HF	Drone Cover	289	6.34	15.4	0.179	6.41
PCBlend CF	0.6 HF	Drone Arms	289	9.25	60.4	0.197	8.54
PCBlend CF	0.6 HF	Drone Fuselage	289	10.24	65.1	0.175	11.39
Timberfill	0.4 HF	3DBenchy	166	4.67	43.9	0.141	13.40[a]
Timberfill	0.4 HF	Moai	184	9.06	87.1	0.162	8.83
Timberfill	0.4 HF	Knife Holder	248	13.68	90.3	0.256	5.87
AluFill	0.6 HF	3DBenchy	246	2.41	85.1	0.153	8.75
AluFill	0.6 HF	Circuit Heatsink	238	2.29	59.4	0.124	8.31
AluFill	0.6 HF	Circuit Heatsink 1.18x	246	2.88	83.8	0.160	8.19

Table 5.15: Parts printed, their nozzle temperatures, average flowrates, total print times t and errors between the planner's motion system predictions and measured values (in Amps, for the XY stage) and between pressure predictions and measured values (in Newtons).

[a] For both the PETG-CF 3DBenchy and the Timberfill 3DBenchy, the workflow chose flowrates close to the minimum flow temperature, which I believe is why these two prints show outsized modelling errors; see Section 5.11.7 for discussion on this.

[b] This part was printed *with a 0.6mm nozzle,* but the model used for flow was generated with the *0.4mm nozzle,* which is the likely source of error in this case.

Flow models are typically accurate only to 5% or so and can be particularly bad over certain intervals in a part. In Figure 5.68 I show two time-series slices of data from the part in Figure 5.66; models fit well in certain state spaces but poorly in others. We can see divergence in values for the extruder pressure and even divergence in phenomenology: the plot at right shows sharp pressure peaks as each new track begins whereas the model predicts square-topped pressure values. As I discussed in Section 5.10.5.1 I think that this emerges because of the lossy method that I use to

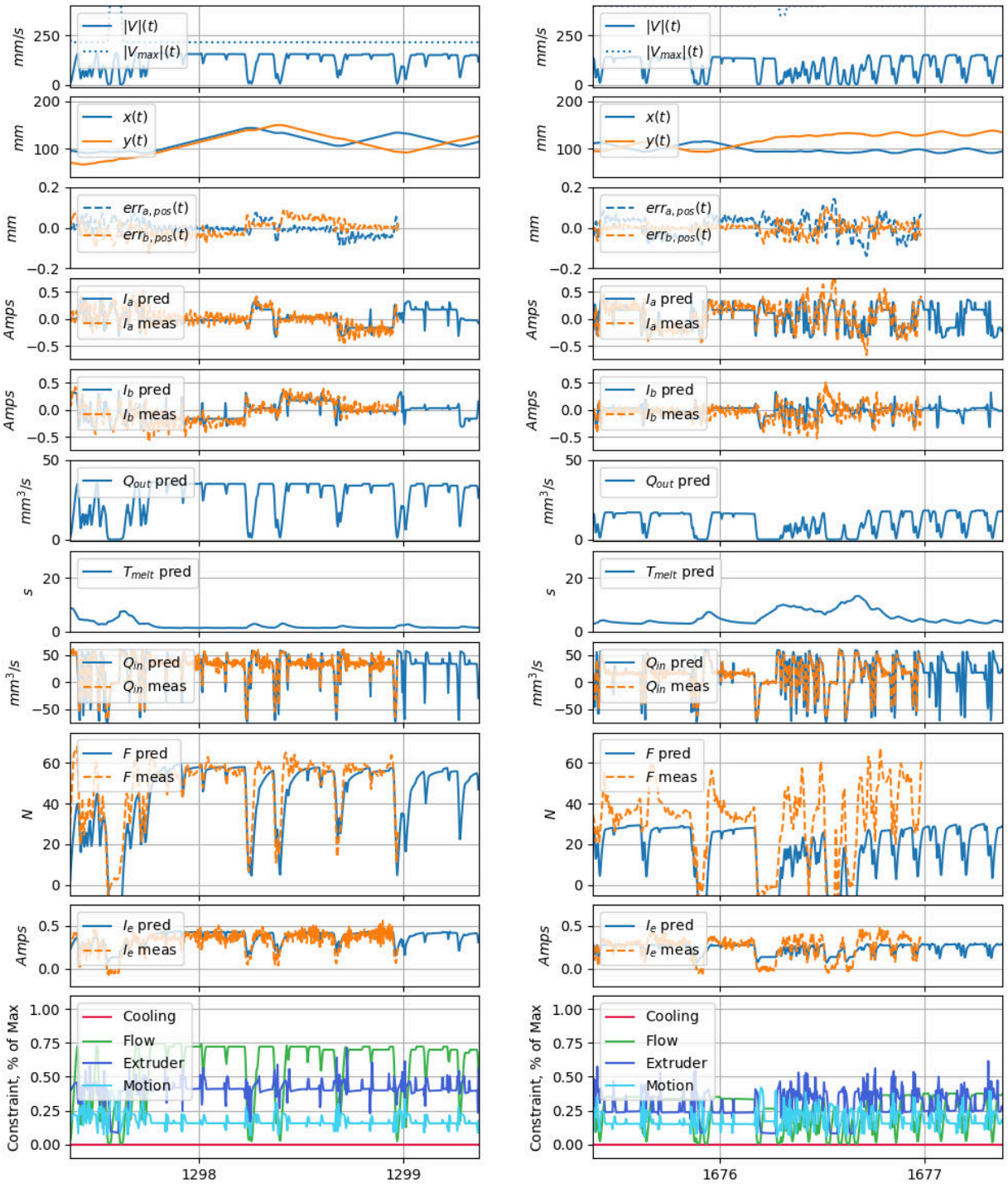


Figure 5.68: Time-series selections from the Kleat print in Figure 5.66, traces in blue are planner predictions and in orange are measurements. The plot at left is a two-second slice around $t = 1298$, an area of the print where models did well to predict measurements. At right is a two-second slice around $t = 1676$, an area of the print (highlighted in yellow in 5.66) where the models fit poorly. Data top-to-bottom: velocities selected by the planner, positions, positioning error, `a` motor currents, `b` motor currents, outflow predictions, melt flow dwell time T_{melt}, inflow, load cell readings, extruder current, and saturations from Section 5.11.1.

approximate time-varying melt flow behaviour; the right plot (where models fit poorly) is taken where the melt flow's dwell time (T_{melt}, measured in seconds and representing how much time the filament at the nozzle tip has spent in the melt zone) is on average much longer than it is in the left plot (where models fit well).

The datasets used to fit these models comprise only tens of seconds of dynamical operation at each temperature under pseudo-isothermal conditions and then effectively one data point at each point in each approximately steady-state condition. Real-world operation of the machine spans many more states: time-varying melt flow temperatures, an array of nozzle temperatures, and a full span of pressures and flows. The other confounding factor is that extruding a track on top of an existing surface requires slightly more pressure than extruding into free air, which is how the initial datasets are generated.

This means that the modelling required to make this method work really well needs to be done using time-series data collected from real operation. In Section 5.12.1 I discuss how we might properly merge the isothermal and steady-state models and use that development to enable "continuous improvement and deployment" (CI-CD, Section 5.12.2) of these models. That would allow us to bootstrap models with the simple tests that I have developed here (or perhaps even simpler versions of the same) and then update models as the printer is used.

One *result* of this lossiness was that I had to implement the retract heuristic, which properly would be handled by the flow model properly driving pressure to zero. In Section 5.12, I cover a range of topics that might lead to improved modelling and operation of the hardware.

Some other part errors emerge from the motion system, which does not model machine stiffness and vibration even though this turns out to be a major limit to overall precision. I discussed those limits in Section 4.8.5, other limits to this system are discussed in Section 5.11.7.

5.11 Discussion

In Section 5.1.1 I described how preset-based configuration of FFF 3D printers limits adoption of new materials because they must be developed for each new material, nozzle or printer configuration, and how state-of-the-art controller configuration separates low- and high-level process control parameters. Section 5.2.2 explains how state-of-the-art workflows manage printer parameters. I expanded on how motion and process physics are intimately coupled in Section 5.2.3, and why that prevents us from running machines near their real underlying limits.

In the state-of-the-art we tune machines through lossy representations that disconnect what we expect should happen when we change slicer parameters from what actually happens once piecewise configurations and optimizations are applied. To see how we can improve this, I tried to build an end-to-end workflow (Section 5.3) that uses motion, flow, and cooling models as a basis to configure and run a 3D printer. Other researchers have built models that describe these physics (Sections 5.2.4.1 and 5.2.4.3) and some have applied model-based control to parts of the process (Section 5.2.4.2), none have connected motion control directly to process control for FFF, or have connected model-based control to higher level planning. I extended the state-of-the-art in those regards in Sections 5.9 and 5.8 to develop a complete end-to-end workflow (Section 5.3). Most research on overall optimization of the printing workflow is done on top of existing systems (Section 5.2.4.4), i.e., using fuzzy logic and neural networks to try to estimate how slicer settings will impact part performance.

In Section 5.10.1 I tested this workflow end-to-end, showing in Section 5.10.3 that the total space of parameters to tune is reduced and in Section 5.10.4 that it improves time optimality of prints when measured against an equivalent state-of-the-art workflow. In Sections 5.10.5 and 5.10.5.1 we see that although parts printed with this workflow have more visually perceptible errors than the state-of-the-art, those errors correlate well with deviations from model predictions. This is promising because it means that if we can formulate models such that they can be re-fit on new data (which the machine produces copious amounts of), we will be able to sequentially improve models and prints over time (see Section 5.12).

To discuss all of this, I want to go back to the compiler analogy that I made with Figure 1.4. In this comparison, the state-of-the-art workflow amounts to writing assembly codes directly: parameters we set are implemented literally by the machine (subject only acceleration control). Our workflow is something like a compiler, combining high level and physically meaningful heuristics with models and optimization to produce the actual, low-level instructions; it is declarative rather than imperative machine programming. So, in Section 5.11.1 I explore how we can use outputs from this workflow to reflect on machine design and process design / limits, and Section 5.11.2 shows that heuristics that *are* exposed here are tuned *against* underlying limits: rather than throwing darts at the process map, we throw them *within* the process map. We also stand to learn from flow models alone (Section 5.11.3).

The methods that I developed enable some new capabilities. Declarative programming allows optimization-based tools to select parameters from a wider swathe of viable parameter space (Section 5.11.4) and the planner's inclusion of time-varying melt flow dynamics can exploit performance (momentarily) that extends beyond steady state limits (Section 5.11.5). The workflow is also resilient to some human errors (Section 5.11.6), but is not without its limits. I discuss those in Section 5.11.7.

5.11.1 Learning from Printer Data Outputs

Whereas models on their own can tell us about how components of our system behave, the model based controller outputs can teach us about the system as a whole. During any print job, we collect numerous data streams from the machine and the planner that can all be used for analysis later. With sufficiently accurate models, we can even do this with the velocity planner alone, as it represents a kind of digital twin.

Probably the most interesting way to process this data is to analyze which aspect of the machine was limiting overall performance, these are *constraint saturations* and I show an example in Figure 5.69. That includes a distribution of saturations for the whole part (bottom) and those saturations projected onto the trajectory such that we see which physics limited the printer velocity for each segment.

To calculate constraint saturations I compare time series data of each component's outputs state with the relevant limit, for example motor current utilization vs. the maximum current available at that time. Each segment has a maximum velocity set via flow models and related heuristics, when that velocity is saturated we know that nothing else is constraining speed. Where velocity has been scaled to meet a minimum layer cooling time, we know that the part cooling model has limited speed at that point. In the table below, I summed these saturations across *all* the prints that I evaluated using the FrankenPrusa.

These saturations are included in print Tables 5.12 and 5.13. I tally total limit saturation percentages across the complete set of prints in Table 5.16.

Generating these types of outputs is not possible in state-of-the-art workflows, first because low-level control states are not exposed computationally but also because other controllers are structurally different. Trapezoid planners use *motion parameters* rather than motion models and do very little in the way of considering process control, as discussed. Their outputs, if collected, could be used in conjunction with models to render similar traces (replaying motion states through machine models), but we would see that the trapezoidal planner's own limitations would be the main performance limit, as I show in Section 4.7.1. Other model predictive controllers could probably be formulated to generate similar outputs but so far none have been instrumented in this manner, and none that combine extrusion and motion dynamics exist.

Because the control framework in this thesis allows us to run the planner on large compute hardware, there is bandwidth to spare to serialize and save intermediate controller outputs *like* each system components' limits for each time-step in the solution (which are normally compressed

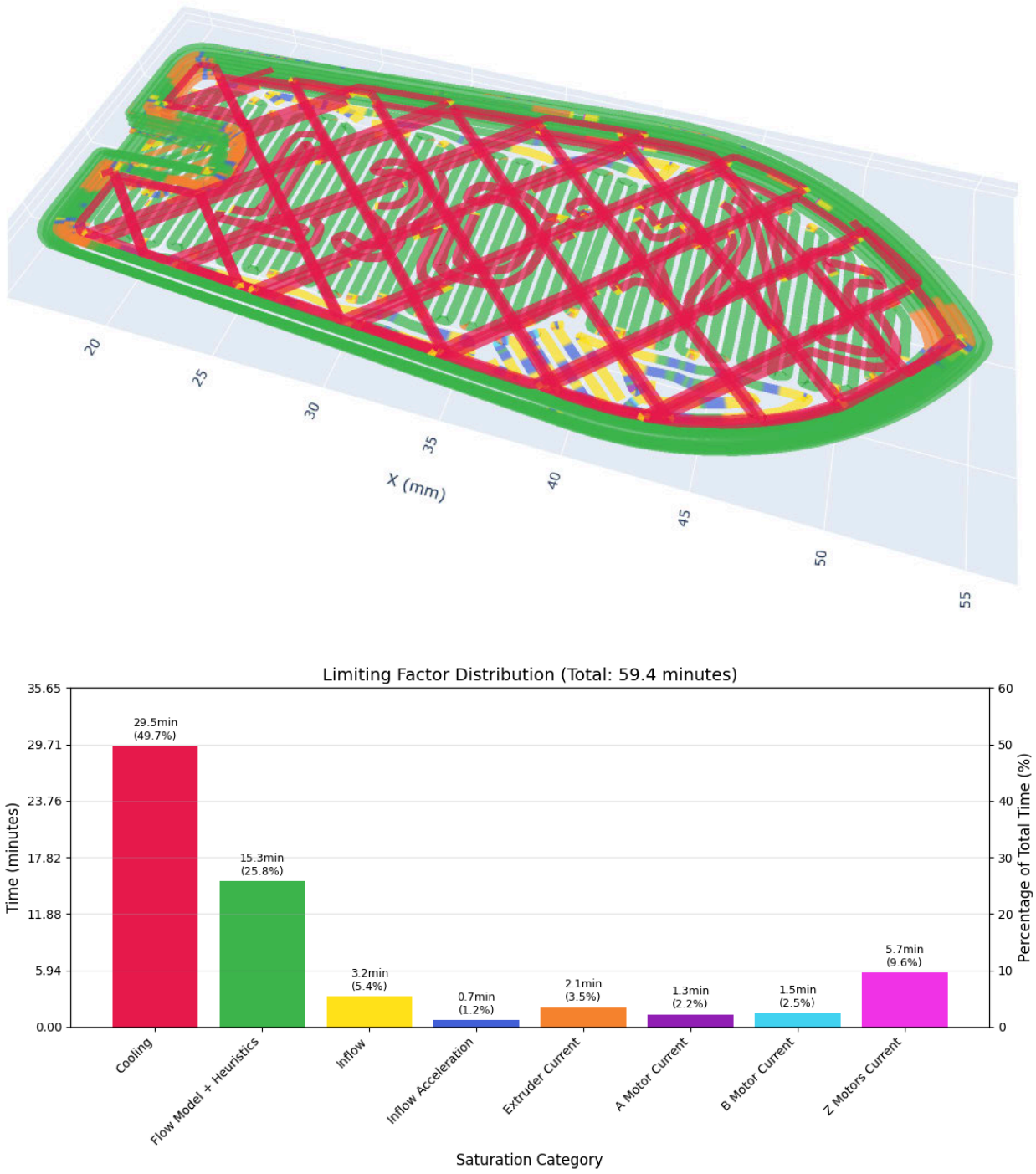


Figure 5.69: This figure renders a section of a printed part in 3D, with trajectory segments binned by whichever machine component (or model consideration) constrained the overall system speed at that segment. Distributions for the total print are below, showing total time that the system spent limited by each constraint. Above is a subsection of the print, where limits are rendered on the trajectory itself.

in the cost function to a single value). That means that this is the first example of a digital fabrication workflow that allows us to analyze a machine system in this manner, and we can learn something new here.

s_{cool}	s_{flow}	s_{extr}	s_{motn}
28.0%	46.0%	15.4%	10.7%

Table 5.16: Compressing saturation bins further into this subset, I tallied the total average saturation across all the prints in Section 5.10.1. We see that *flowrates* are the primary speed limit for that selection of geometries, followed by cooling, extruder dynamics, and *then* motion dynamics.

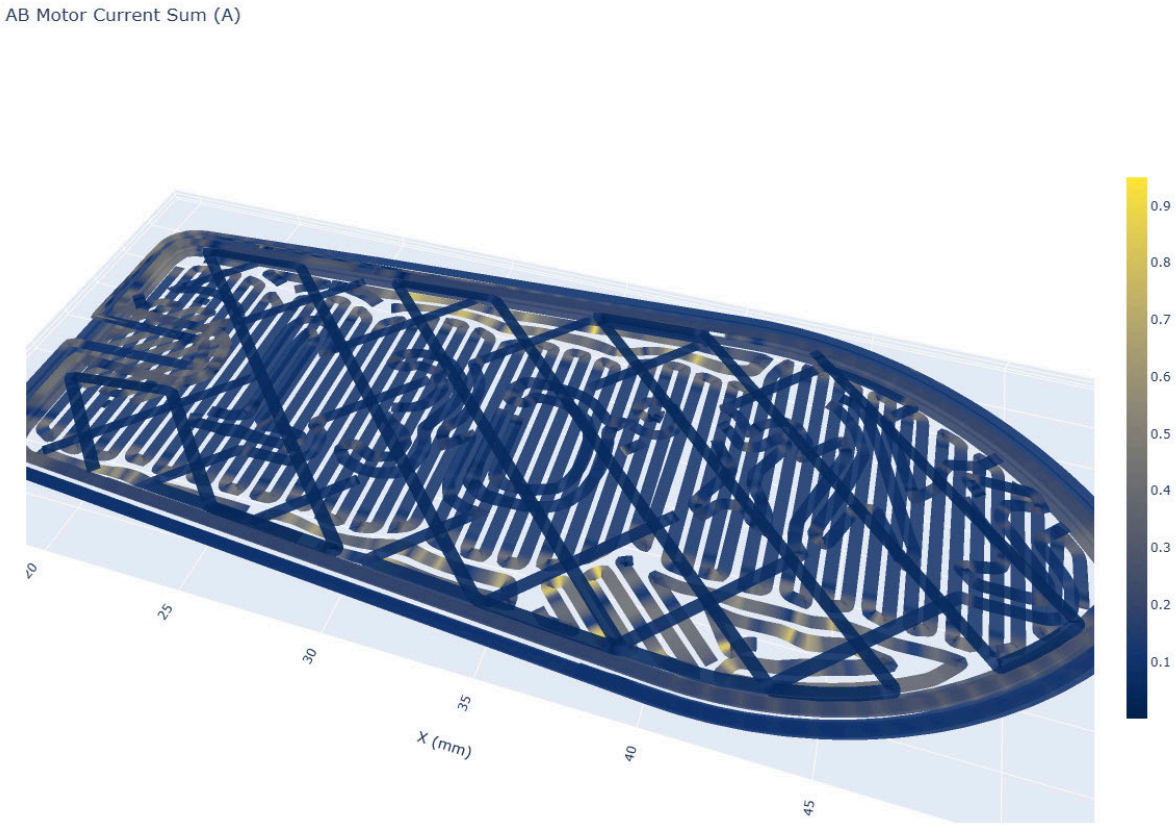


Figure 5.70: The vector sum of A and B motor currents deployed during this section of the toolpath. While we may expect these to peak mostly in corners, here we see that rapidly changing *track widths* also lead to motion system current peaks because the machine needs to rapidly accelerate and decelerate according to the flow system's limits.

The categories covered relate both to machine / motion models and flow models. The most dominant constraints overall (over the short set of parts) are the flow models themselves, which were combined with some heuristics (recall Section 5.8). In 5.12.4, I discuss how these heuristics could be supplanted with explicit descriptions of optimalities like precision and part strength, which would increase the fidelity of this "bin" of saturations. The second most dominant limit overall was cooling: tracks that have been slowed down to meet minimum layer time constraints. Then we have a series of machine related limits: inflow and inflow acceleration as well as extruder current limits are all related to the extruder *motor's* performance and power budget. AB and Z

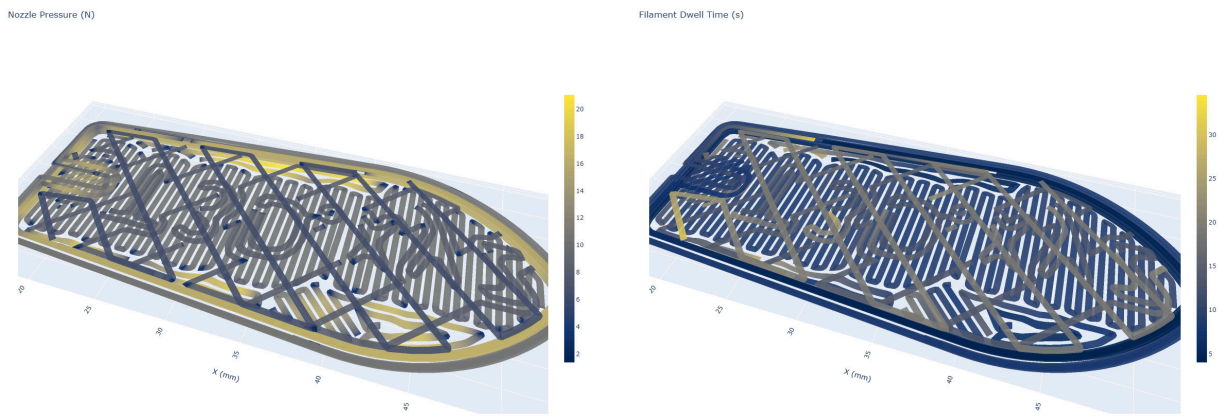


Figure 5.71: Left: data output for nozzle pressure during a print. Right: total filament "dwell time" during a print: how long the segment of filament currently being extruded has spent in the melt zone.

motor current limits relate to our actuator's performance, but also the machine's motion system as a whole: reducing the occurrence of these limits could involve increasing actuator size *or* decreasing friction or mass in the motion system.

Machine design is a game of trade-offs: if we improve overall flowrates by increasing the extruder motor size, we add mass that the motion system must heft around through tight corners. If we decrease the size of the part cooling fan to reduce the same weight, we remove cooling performance and need to add more time in each layer to counter it. Actually determining which of these trade-offs is the best to make is difficult to ascertain on pen and paper alone (or with a good spreadsheet) because constraint saturations are an emergent property of the system as a whole. These limits are also geometry dependent: small, intricate parts we are more likely to be limited by the machine's motion system and part cooling whereas large parts are more likely limited by overall flowrates.

Besides this more complex binning of output data to determine limiting components, we can also render individual traces to learn about our system. Figures 5.71 and 5.70 show nozzle pressure, filament dwell time and AB motor currents as extracted from the toolpath in Figure 5.69.

5.11.2 Model-Based Knobs

I want to talk briefly about connecting tacit knowledge to machine models. In the state-of-the-art, We have very many inputs (slicer parameters) and only a few easily measured outputs (total print time and quality, and observations of obvious errors). Those inputs relate directly to machine *operation,* but not to machine *physics.* This means that learning how and which parameters should be adjusted in order to combat different negative outcomes (or improve positive ones) requires that users develop an intuitive sense for machine physics and a tacit understanding of how parameters impact those physics through multiple layers of control.

For example, we might slow down linear traversal rates (Table 5.2) and see that part quality was improved. But this adjustment has three physical effects: it slows flowrates (reducing required

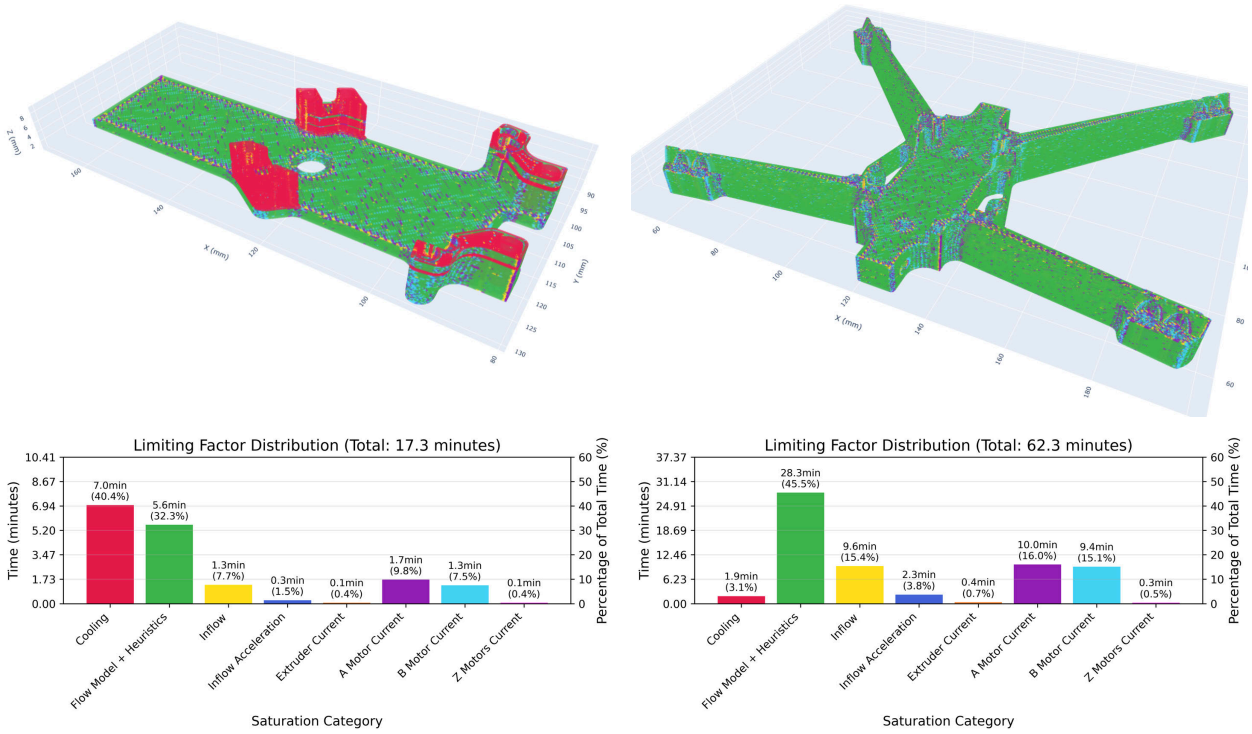


Figure 5.72: In smaller parts like the drone fuselage cap (at left), cooling tends to dominate total process time — but only in smaller layers. Larger parts like the drone's arms (at right) are dominated by *flow* and motion dynamics. In Section 5.12.1 I discuss how we could extend model-based control to use variable nozzle temperatures throughout a print.

extrusion pressure and extruder motor current), it calms the motion system (reducing actuator current requirements) *and* it causes layer time to increase, which changes the layer cooling system as a byproduct. It is difficult to tell which of these affects was responsible for the improvement.

This can happen across changes to geometry as well: big parts may print well using the same slicer parameters that work poorly for small parts, but we have little recourse to understand why. Using the new outputs generated by this workflow, we can see that smaller parts tend to saturate our motion and cooling models (left plots in Figure 5.72), whereas larger parts tend to saturate flow models (right plots).

In FFF, users often tune their machines by hand towards high speed printing. But hidden velocity optimizations can make this process confusing; in Figure 1.8 way back in the introduction I gave the example of two trajectories: one with a $100mm/s$ speed setting and the other with a $500mm/s$ setting. Both were run through the same velocity optimization with the same $1000mm/s^2$ acceleration limit. Despite the $5x$ speed *setting* increase, the realized time improvement was only 14% because the machine spent most of its time accelerating and decelerating. This can lead to tuning loops where we increase speed settings aggressively while printing a small test part (thinking that we are improving performance), and then when we run a larger part (where the machine can actually accomplish the target speeds), our flow constraints are suddenly limited and

Layer Height	Infill Pres.	Perim Pres.	Detail Pres.	XYZ Cur.	E Cur.
0.3	0.75	0.65	0.45	0.25	0.75

Layer Height	**Infill Pres.**	Perim Pres.	Detail Pres.	**XYZ Cur.**	**E Cur.**
0.3	**1.75**	0.65	0.45	**0.45**	**0.95**

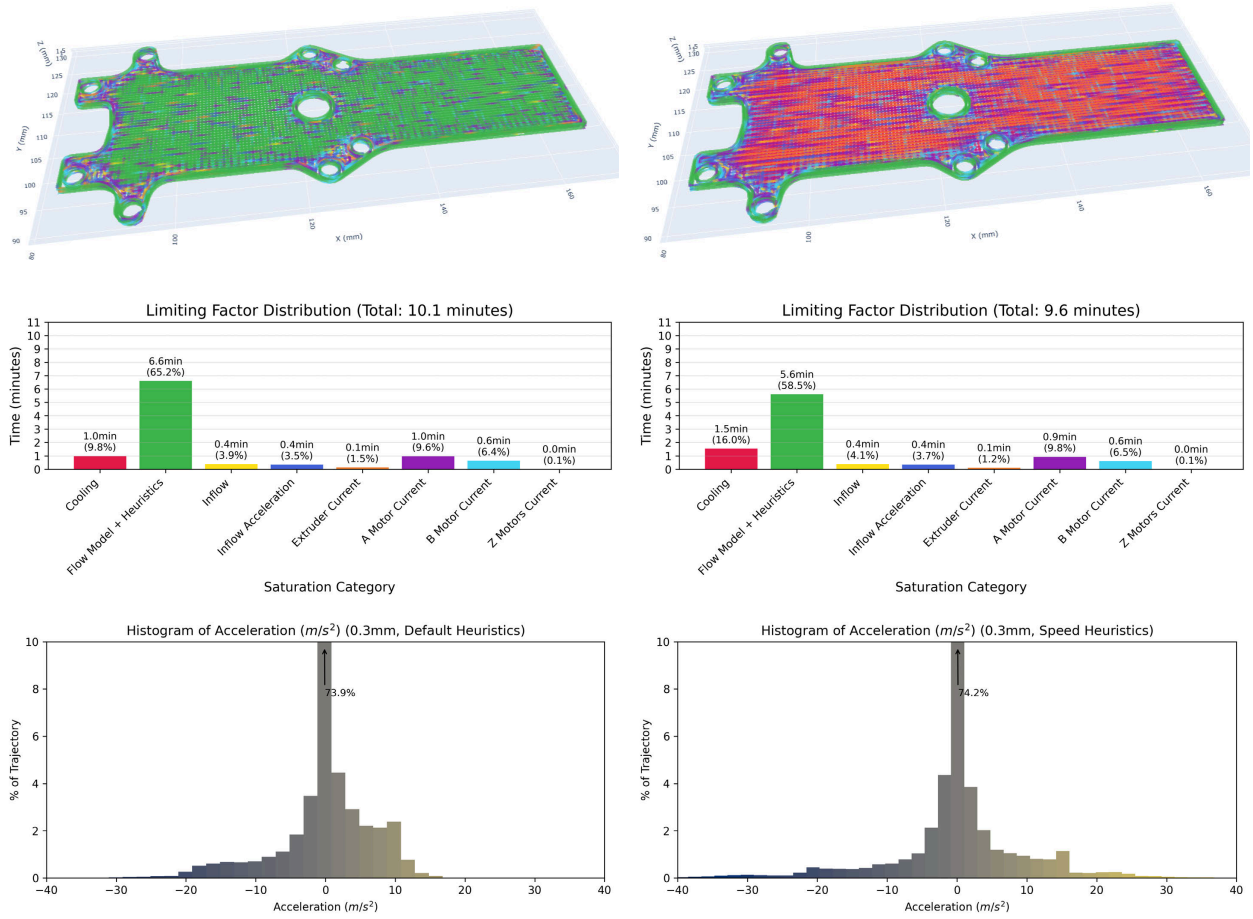


Figure 5.73: To express that we would like to maximize speed (at the potential cost of quality), we can increase the *scalars* that describe how much of the underlying maximum actuator motive force (in XY and in the Extruder motor) should be used and push the target extruder force during infill past its steady-state limit. At the top are heuristics / parameters applied to each part, and then we have saturations and bins, and the resulting change to the distribution of accelerations.

extrusion fails. I discuss this also in Section 6.3.1 in the machining chapter because visualizing these deviations is valuable to CNC programmers *before* they run toolpaths.

In Figure 5.73, I show how I express the same *tune* using the workflow that I developed in this chapter: at left is a part that uses conservative flow and motion heuristics that I tuned for quality, and at right is a part that uses aggressive heuristics tuned for speed. Rather than writing a new parameter that we might expect would result in more speed, we say at a high level that we want the printer to use more of its available capacity — as mediated by the flow and motion models via the realtime controller, increasing the *scalar* that describes how much of the maximum steady-state pressure should be targeted in the nozzle during infill moves (from $0.75 \rightarrow 1.75$) and also

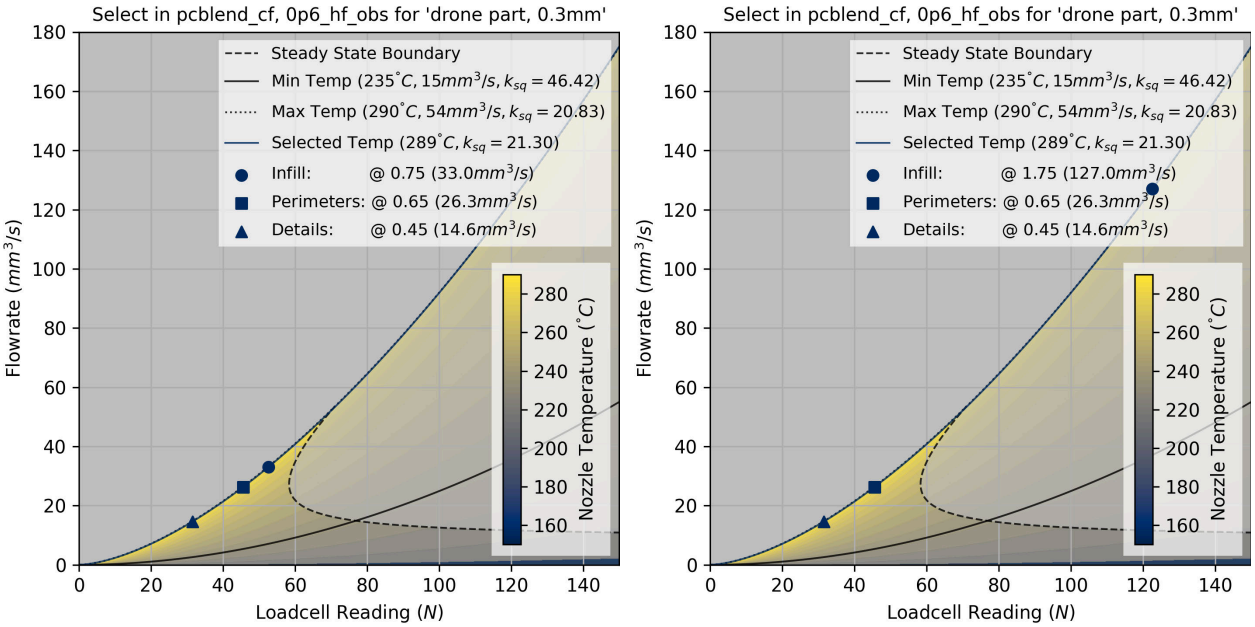


Figure 5.74: Pressure scalar selections from Figure 5.73; on the right hand we set the infill pressure scalar at $1.75x$ the maximum *steady state* load. The dynamic flowrate can be even higher when we can plan for time-varying melt flow temperatures.

increasing the proportion of the maximum available motor current that the system should use (from $0.25 \rightarrow 0.45$ for XY motion and from $0.75 \rightarrow 0.95$ for the extruder). I show parameters, resulting saturation maps and distributions, and histograms for the accelerations that result from the deployment of more or less actuator current.

Note that the saturation bars in Figures 5.73 and 5.75 all use the same *time* span across they y-axis. We can see that the total print time is decreased from 10.1 to 9.6 minutes and that the acceleration used changes: with more motor current deployed the distribution widens. The saturations change too; time against the motor current limits stays about the same (saturations are measured against the applied heuristics), but the cooling model starts to dominate more presumably because the speeds reached at the larger infill pressure are subsequently scaled back by the cooling model.

I want to offer one more example here, which I also called out in Section 5.2.2: if we change the *layer height* in our the state-of-the-art workflow, we effectively double the flowrates that our machine will try to operate with across all segment types. This is another result of writing direct parameters: geometric settings and flow settings are coupled in non-intuitive ways. In this workflow, geometry is decoupled from flow. In Figure 5.75 I show two of the same part, one printed at $0.2mm$ layer height and the other at $0.3mm$, both at the same nozzle temperature. Flowrate selections remain the same but the velocity distribution shifts. The machine is doing less "total motion" for the same print volume, so flow and cooling models saturate more often in the thicker part (together they are limiting for 71.8% of the $0.2mm$ print and 74.5% of the $0.3mm$ print), and motion dynamics saturate more often in the thinner part (19.7% of the $0.2mm$ print and 16.3% of the $0.3mm$ print).

I think that these examples show how machine operation can be re-framed using models. The methods that I developed in this chapter reduce the total number of parameters we need to tune in

Layer Height	Infill Pres.	Perim Pres.	Detail Pres.	XYZ Cur.	E Cur.
0.2	1.75	0.65	0.45	0.45	0.95

Layer Height	Infill Pres.	Perim Pres.	Detail Pres.	XYZ Cur.	E Cur.
0.3	1.75	0.65	0.45	0.45	0.95

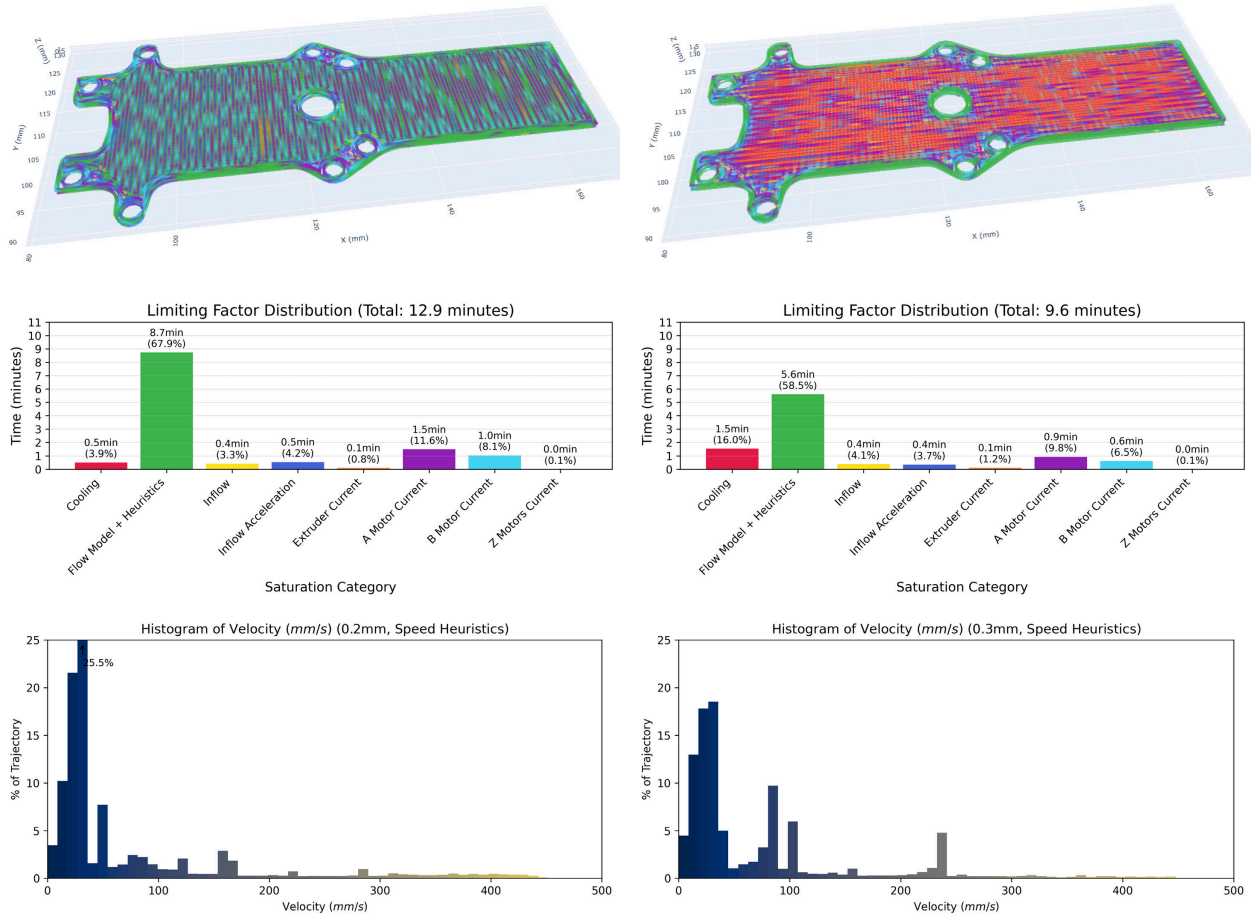


Figure 5.75: Updating the layer height (from $0.2 \rightarrow 0.3mm$) doesn't change the flowrate targets in this workflow. Instead, flowrates are selected using the pressure scalars. Changing layer height causes the machine to spend less time overall printing the part (from $12.9 \rightarrow 9.6$ min), and changes the distribution of limiting factors: less time is spent against the motion system's limits but a higher proportion is limited by flow and cooling.

order to print with new materials; they also connect the parameters that we *do* tune more directly to the real underlying physical constraints in our system. Rather than setting parameters without knowing if they will drive our printer beyond its underlying constraints, we tune them *with respect* to those constraints. In doing so, we can express what we want our printer to do at a higher level, for example "use more of the available motive power" rather than "try to print at exactly this speed," without any knowledge about whether that rate will actually be accomplished, or if it will cause the flow or motion systems to be over extended. In addition, we get new outputs that let us inspect the results of the *realtime* dynamics' application to those parts, identifying in minute detail how machine physics relate to performance. Because the velocity planner still applies an optimization on top of this, the results of these tuning updates can still have indirect consequences especially

in cases where the cooling adjustments are applied. But because that planner can run entirely in software using just a *virtual* model of the machine, we can experiment with these updates *before* running a part — and we can see *which* optimization or model has been applied and where.

I showed how model-based control can also help us understand errors from printed parts, in Section 5.10.5.1 we saw that visual artefacts tend to correlate to deviations between model predictions and measurements. That can happen when models are poorly fit or when unforeseen disturbances (or misconfigurations) occurred. In Section 5.12.4 I discuss how we might extend the workflow to express trade-offs that I currently manage using input heuristics as a weighted set of optimalities for speed, strength, and precision.

Finally, I want to stress that I do not see this as a replacement for our mental models of machines and processes. Instead, I think of these as a tool to help more of us develop those models in more detail, with less time, and in a manner that could be more easily shared with others.

5.11.3 Learning from Flow Models

5.11.3.1 Comparing Materials and Fills

I want to make the simple point that models encode information about our materials that could be used to develop intuitive understandings of those materials. In Figure 5.76 I show three materials, each developed on the same nozzle and printer.

We can also compare materials with different fillers, e.g., carbon or glass fibers. Before I started this project, I assumed that filaments with fiber fill were overall harder to print because the fills are stiffer than molten plastic and do not melt themselves. Looking at their flow models, I was

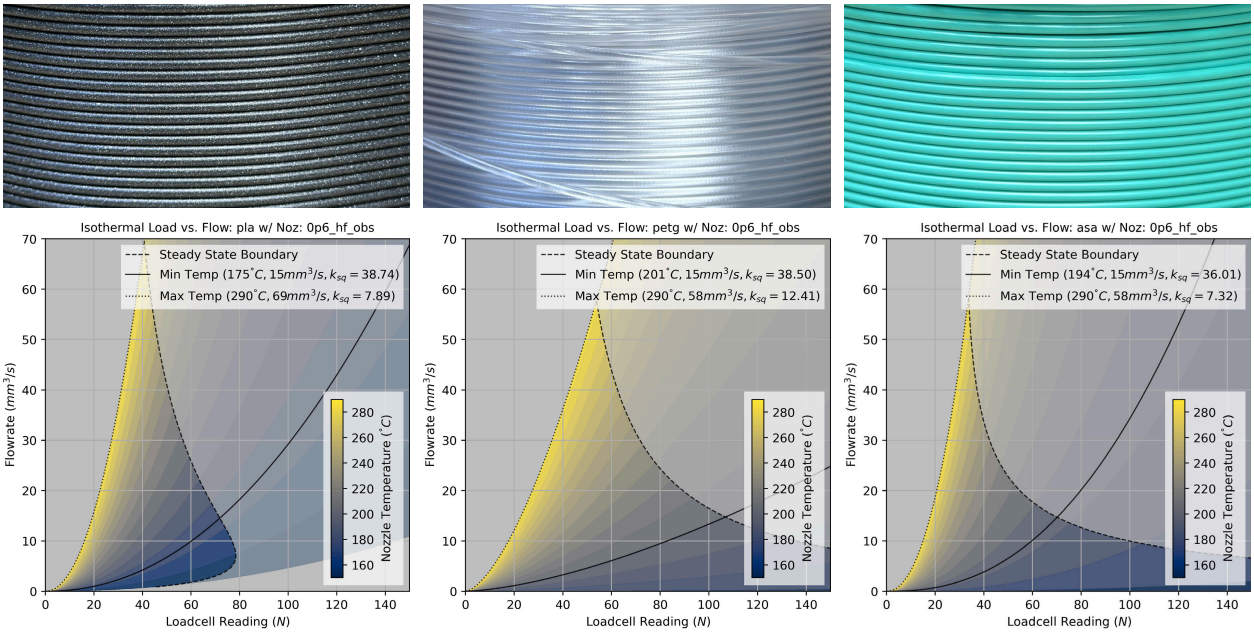


Figure 5.76: Isothermal models for PLA, PETG, and ASA (left to right), using the FrankenPrusa and the 0.6mm High Flow Nozzle.

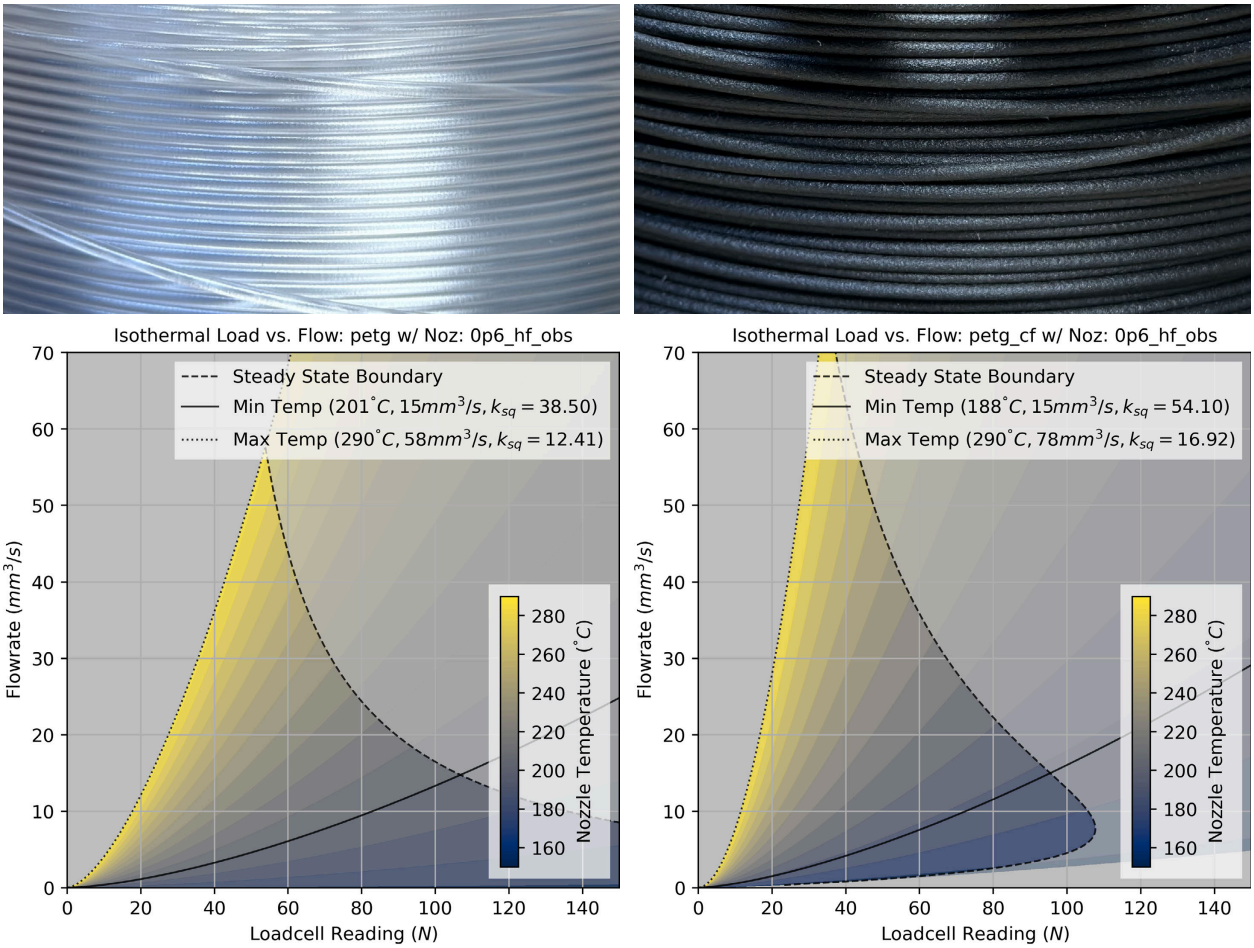


Figure 5.77: Isothermal Models for PETG (left) and PETG with Carbon Fill (right), using the 0.6mm High Flow Obxidian Nozzle.

surprised to see that filled materials actually tend to have favourable flow properties. Models for PETG (without, and with) carbon fill are shown in Figure 5.77, both using the same nozzle; we can see that the filled material moves the steady-state boundary away from the origin, and also substantially *decreases* the minimum flow temperature.

I think the phenomenology is that the fill — while increasing the viscosity of the material — also substantially increases its thermal conductivity. This leads to better flows overall because the melt flow's internal temperature is maintained at high flowrates. It may also be the case that the base PETG used in either filament is actually from a different batch or slightly different formulation, and carbon fibers may already be aligned in the filament in which case it would make sense that their inclusion doesn't decrease the base material's viscosity substantially.

If you read the spring rate (k_{sq}) for each material, you will see that the fill has a higher spring rate (16.9 vs $12.41 N/mm^3$) at the same temperature (290° C). We would expect this from the carbon fill, and it shows that it may be possible to extract some metric for material stiffness using this hardware, i.e., simply using the extruder and load cell and extruder as an in-situ Universal Testing

Machine.[94]

5.11.3.2 Comparing Nozzles

Recently, new FFF nozzles have been developed that include some internal geometry that improves heat conduction into the filament. Models for flow can provide a basis for comparing these parts. Below, I show flow models for PLA and PETG, using a traditional 0.6mm diameter nozzle on the left and a 0.6mm High Flow nozzle on the right. We can see that these nozzles *do* increase overall flowrates (as claimed) and also expand the size of the viable parameter space, moving the steady-state boundary curve away from the origin.

In Section 7.2.1, I discuss how figures or models like these could be used by nozzle manufacturers and customers to better articulate performance improvements or characteristics. For example a standard datasheet for a motor includes a torque curve, but in the state-of-the-art nozzle manufacturers only include scalar values for "maximum flow" for their nozzles, which are not even normalized across materials.

5.11.4 Using a Full Range of Possible Nozzle Temperatures

It is difficult to tune state-of-the-art systems across temperatures because springrates, maximum flowrates and cooling times all change along with nozzle temperature. Because here we can extract all of these values *from* models, we can use temperature as a tuning value without having to update all of those other direct parameters. For example Figure 5.79 shows this applied to the Nylon-GF filament; using the same input parameters overall the time-optimal nozzle temperature selects a colder temperature for the 3DBenchy (which is tall and skinny) than for the hood latch (which is relatively flat, and wider), based on the layer cooling models.

It would be difficult to apply this optimization if we could not automatically update other parameters against that temperature, i.e., we need a way to know *how much* flow can be increased with increased temperatures, how much extra cooling is needed, and we would need to manually interpolate springrates across temperatures so that the filament compressibility value is updated. However, the benefit is clear: the second row of Figure 5.79 shows that increasing nozzle temperature from $230 \rightarrow 272^\circ$ C drops the print time from $11 \rightarrow 8.5$ min, a 22% speed-up. Other examples of this e.g., the Kleat print in Figure 5.31 (from Section 5.8.4) is decreased from $31 \rightarrow 20$ min. Minimum temperatures here are set using a *minimum flowrate* heuristic that I describe in Section 5.8.1.

5.11.5 Exploiting Time-Varying Printing Dynamics

In some examples above I set the pressure scalar for infill moves to be 1.75 x the maximum *steady state* pressure at that time. This is viable because the velocity planner still optimizes against

[94] A Universal Testing Machine (UTM) — often known by the brand name Instron — is a machine used to generate stress-strain curves for material samples. They stretch or compress materials (normally until they break) while measuring elongation or squish and recording load cell measurements.

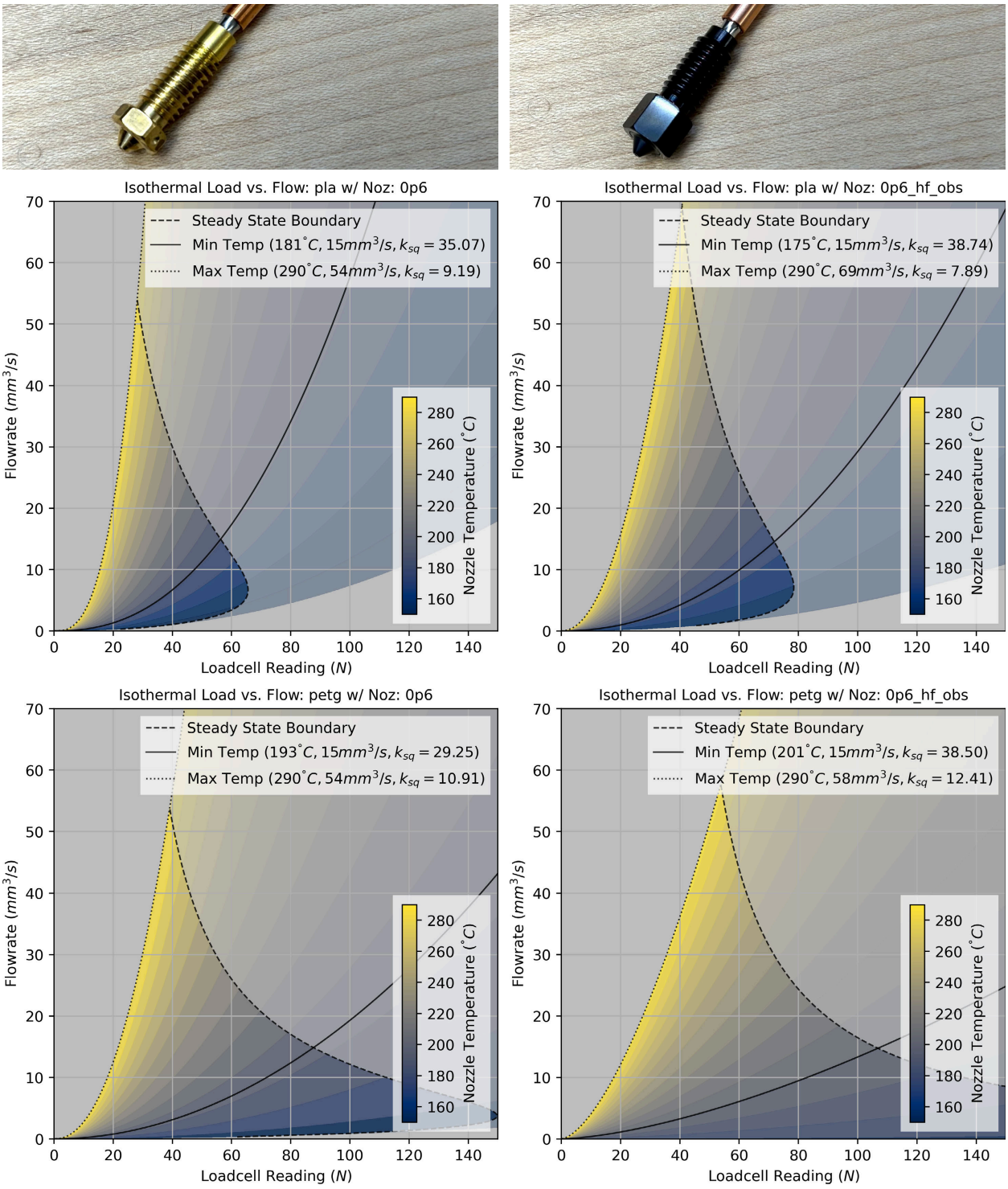


Figure 5.78: Isothermal models for PLA (middle) and PETG (bottom), using 0.6mm Brass (left column) and 0.6mm high flow (right column) nozzles. The high flow nozzles have a longer melt zone and some internal geometry designed to improve conduction into the filament. Flow models quantitatively capture the size of the improvement across temperatures and flowrates; in both materials here the steady state boundary (dashed line) is expanded.

the complete flow models; the dynamical limit to flow is much larger than the steady-state limit

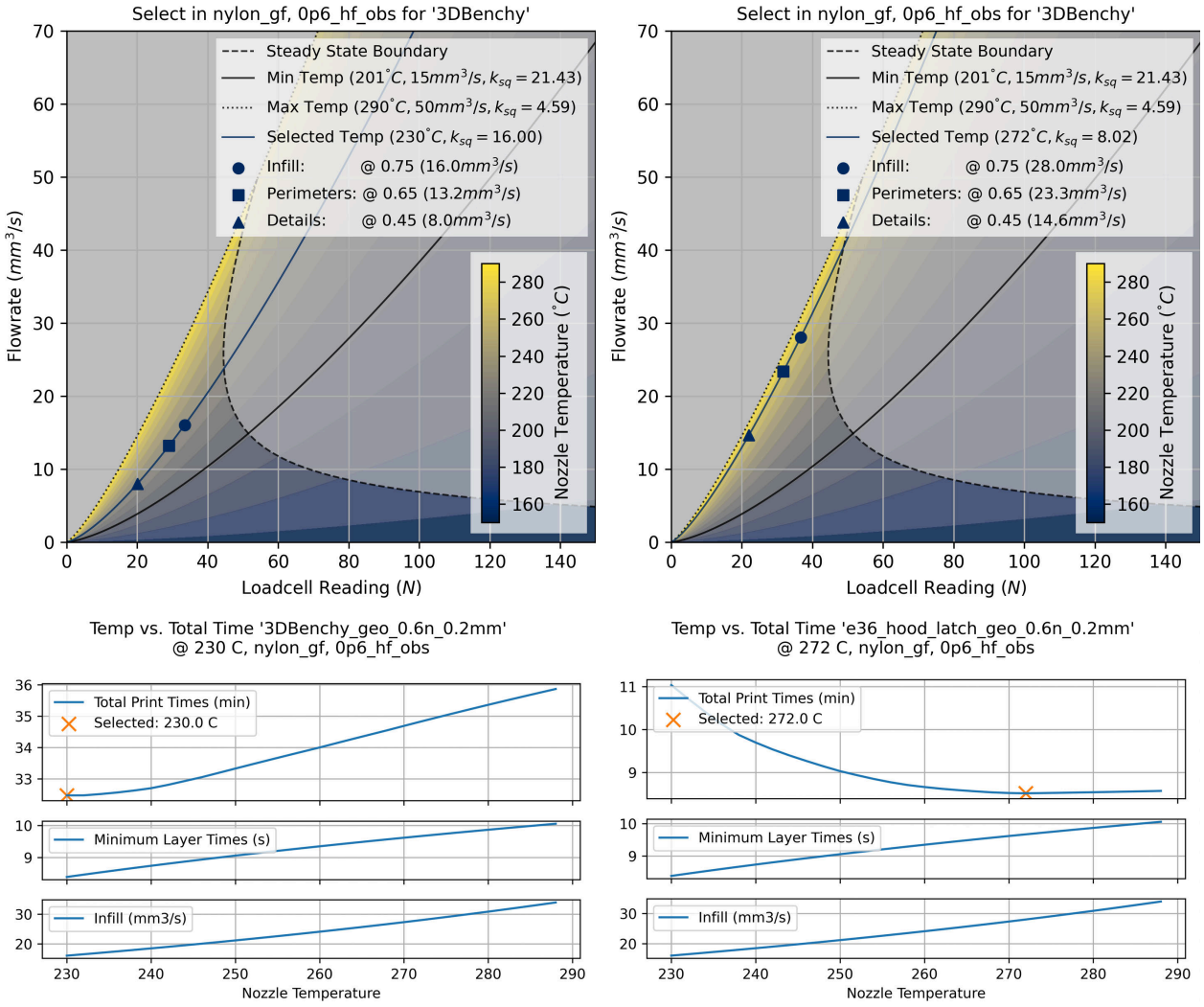


Figure 5.79: Flowrates and springrates extracted from the same flow model (for Nylon-GF) at two different temperatures: 230° C for the 3DBenchy (left) and 272° C for the hood latch (right). Plots in the second row estimate total print time across temperatures according to the layer cooling optimization.

because over short time spans maximum flow is based on the underlying rheology, not the time-varying melt flow temperature / nozzle thermodynamics (see Section 5.5.4).

For example in Figure 5.80 the third row from the top is the motion planner's estimate of the internal melt flow temperature; this is colder when the average velocity of the material in the melt zone is fast, but after slow parts of the print (e.g., detailed outer perimeters) the melt flow temperature gets close to the nozzle temperature. So, when we transition out of these regions there is an opportunity to print much faster than the steady-state limit alone encodes *until* the thermodynamic limit is reached.

5.80 shows how these dynamics evolve in the highlighted segment. When it starts the melt flow is close to the nozzle temperature (around 240° C in this case). As the machine accelerates the outflow briefly exceeds the steady-state limit, but as the melt flow temperature decreases (towards

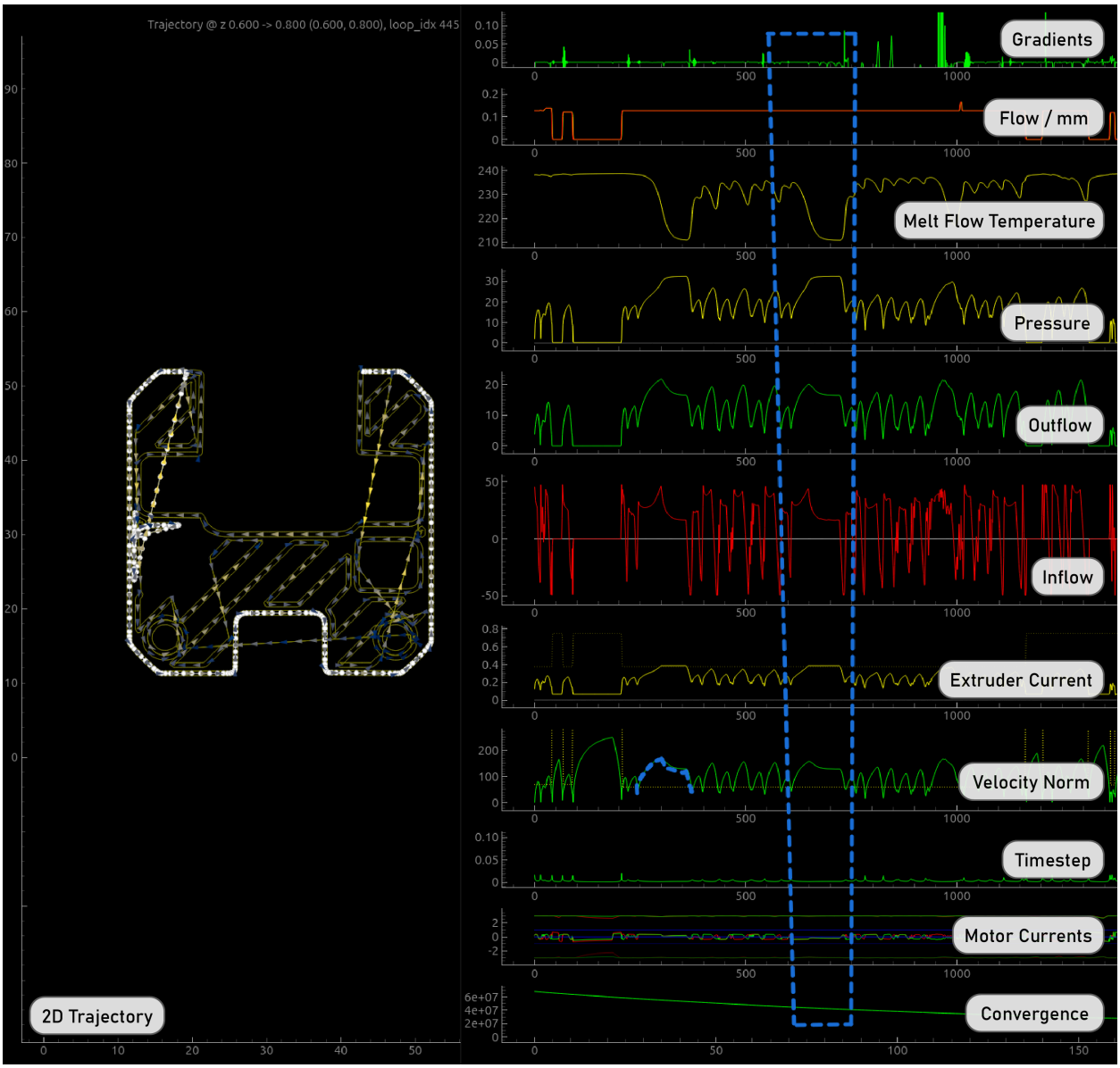


Figure 5.80: A highlighted segment from the model-based velocity planner for FFF, where we see that the planner can leverage some of the flow system's time varying dynamics; a complete explanation is in Section 5.11.5.

210° C) the pressure required to extrude at this rate increases and the system reaches the extruder motor's current limit (5th row from the bottom). To maintain track width, the machine slows down overall, which is visible in the velocity norm plot (fourth row from the bottom).

This is another good example of how motion and flow are coupled and how different physics limit operation under various conditions, as I discussed in Section 5.2.3. It is also a good example of how this planner enables us to optimize over time-varying dynamics; motors themselves have similar thermal limits and as I discussed in Section 4.9.2 applying this strategy for motion systems themselves may have real benefit.

5.11.6 Resilience to Human Errors

A number of prints in Section 5.10.1 have a note: *geometry was mistakenly generated for a 0.6mm nozzle.* In these cases, I made the human error of labelling a flow model with the nozzle that I *thought* I had put into the machine (a 0.6mm High Flow Obxidian nozzle), whereas I had actually installed a **0.4mm** version of the same. The Obxidian nozzle is black, making it difficult to catch the nozzle diameter visually.

Some of these prints had more extrusion artefacts than usual, especially on top surfaces. I assumed that these were just modelling errors, until I realized that the nozzle was actually smaller than I had labelled it. However, they all *worked* and on later inspection the model errors are not very far out of the distribution when compared to the rest of the prints.

In these cases, the models used to process the files *was* still consistent with the nozzle that was used to print the parts: I swapped it before making models, not between model-making and printing (except for in one case). This means that the only thing amiss is that the *track widths* were configured for a 0.6mm nozzle, while the nozzle was actually 0.4mm. The system is resilient to this error since track widths are not dependent on nozzle diameter, only on total flowrate per unit of spatial displacement (mm^3/mm), which can be forced out of a nozzle of any size.

It does mean that a key heuristic was missing, which is that track widths should not be printed that are substantially larger than the nozzle diameter. In Section 5.2.2 I briefly explained this: track widths are normally calculated over nozzle diameter, e.g., up to $1.25 \cdot d_{noz}$ In these cases, track widths are about $1.7 \cdot d_{noz}$.

As an experiment, I tried printing using the state-of-the-art workflow, now intentionally making the error: I used a preset for each material in the two figures below for a 0.6mm nozzle, but installed a 0.4mm nozzle in the machine. Surprisingly these prints also worked without significant errors, as we can see in Figures 5.82 and 5.81. That also indicates that presets and state-of-the-art printer controllers do not drive machines very close to their real underlying limits; if they were close, shrinking the nozzle size by 50% would surely cause failures. This is likely because operating close to physical limits during normal operation would lead to more visual artefacts and problems, and because the state-of-the-art tuning workflow does not allow preset developers to directly see those limits and tune against them.

5.11.7 Key Limits

I've shown that we can fit and then use models for FFF flow dynamics in-situ, and then how we can organize a feedback-based *workflow* around these models that simplifies much of the direct parameter tuning from the state-of-the-art workflow. But there are a few notable limitations to what I have done, which I will discuss here.

5.11.7.1 Missing Piecewise Evaluations

This connected a long list of components: isothermal and steady-state models, an optimization-based planner, a new set of tools for systems integration and motion control (from Chapter 2), a



Figure 5.81: 3DBenchy printed in Timberfill, with a 0.4mm High Flow nozzle installed, but systems configured under the conditions where the system is mislabelled as if it had a 0.6mm High Flow Nozzle installed. Left: our workflow, right: SOTA.

time-optimal nozzle temperature selection using models for cooling, and a new method to apply heuristics *alongside* models to e.g. scale relative pressure and current deployment during operation.

Each of these components could be unpacked and tested in more detail: for example understanding if the method that I developed in Section 5.5.3 to model *outflow* from *inlet flow* and the internal

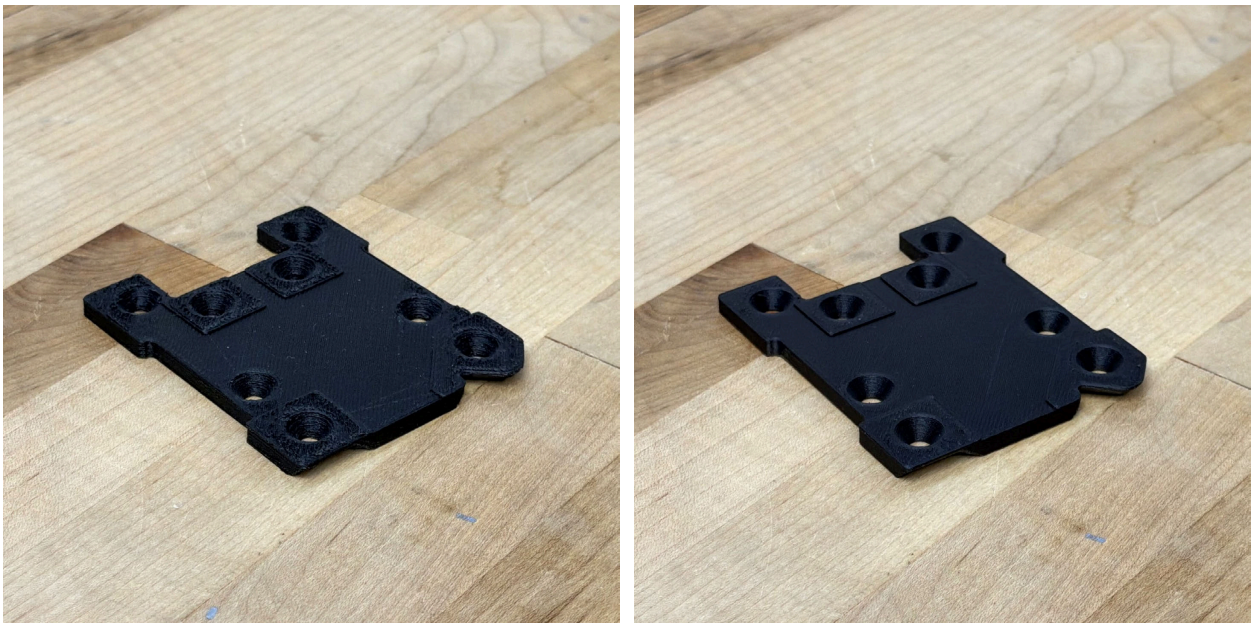


Figure 5.82: The Kleat printed in PETG-CF, with a 0.4mm High Flow nozzle installed, but systems configured under the conditions where the system is mislabelled as if it had a 0.6mm High Flow Nozzle installed. Left: our workflow, right: SOTA.

pressure measurement is only implicitly evaluated here in the end-to-end workflow; although the prints that I showed are not visually exceptional they do not have obvious under- or over-extrusion errors in corners as would be the case if the flow models were egregiously under-fit. Evaluating this method on its own would be possible using the methods developed in [272] and in the Rubedo system [299]. I had even previously developed a workflow for these direct measurements (see Figure 5.8 in Section 5.2.3.3), but didn't apply that measurement to verify the new method. The interpolation between steady-state and isothermal models from Section 5.5.4 could be verified against FEA models like those developed in [309], [310], [311] (although those primarily cover only steady-state operation, see Section 5.2.4.1), or against measurements taken using a thermal camera using the methods developed by [318] (see Section 5.2.4.3), and the layer cooling models could be verified using thermal imaging.

Not having done these evaluations makes it more difficult to tell *which* of these contributions is most productive and which may add errors rather than help to improve prints overall. There is already excellent research on most of these topics in isolation; but they have not been combined with control nor in end-to-end workflows, which is primarily where this work advances the state-of-the-art (although the flow measurement using the internal pressure state and the combination of steady-state and isothermal dynamics are both novel, as is the particular set of model fitting routines). I wanted to focus on exploring how models can be integrated across many aspects of machine control, organizing previously disparate representations using physics directly. In *"Wait, it's all Models?"* (Section 7.2) I explore this in more detail.

5.11.7.2 The Isothermal to Steady-State Interpolation is Lossy and Prevents Refitting

In Sections 5.10.5.1 and 5.10.6 I began discussing why I believe the steady-state to isothermal interpolation (5.5.4) is a main source of loss in this work; the printer covers a much broader state-space during real-world operation than it does in the small test routine that I run to fit these models. In particular, the tests are designed to approximate *either* steady-state *or* isothermal operation. Neither condition is actually reached even in the test routine, which is obvious when we see how the models overlap in Figure 5.83: according to my interpretation of these models, all points in the steady-state contour should be *above* the same point in the isothermal contour because the nozzle thermodynamics is such that for the same flowrate and pressure, we should need a larger nozzle temperature for continuous flow than we would need for instantaneous flow. However, as formulated the models overlap near the origin.

I made the connection between these two models only late in the development of this system; earlier versions of the velocity planner used the isothermal model alone but that led to obviously wrong model fits in places where average flowrates were high. Besides limiting the quality of the velocity planner's pressure predictions (which has ramifications also for retraction), describing this system as two separate models that are interpolated limits my ability to update the models on new data. That would be an invaluable capability because real-world operation obviously spans the state spaces that are most relevant to fit models within. In Section 5.12.1 I discuss how these could be properly connected from the start by developing a coherent energy model for the system and fitting some additional free parameters for heat flux, heat capacity, etc.

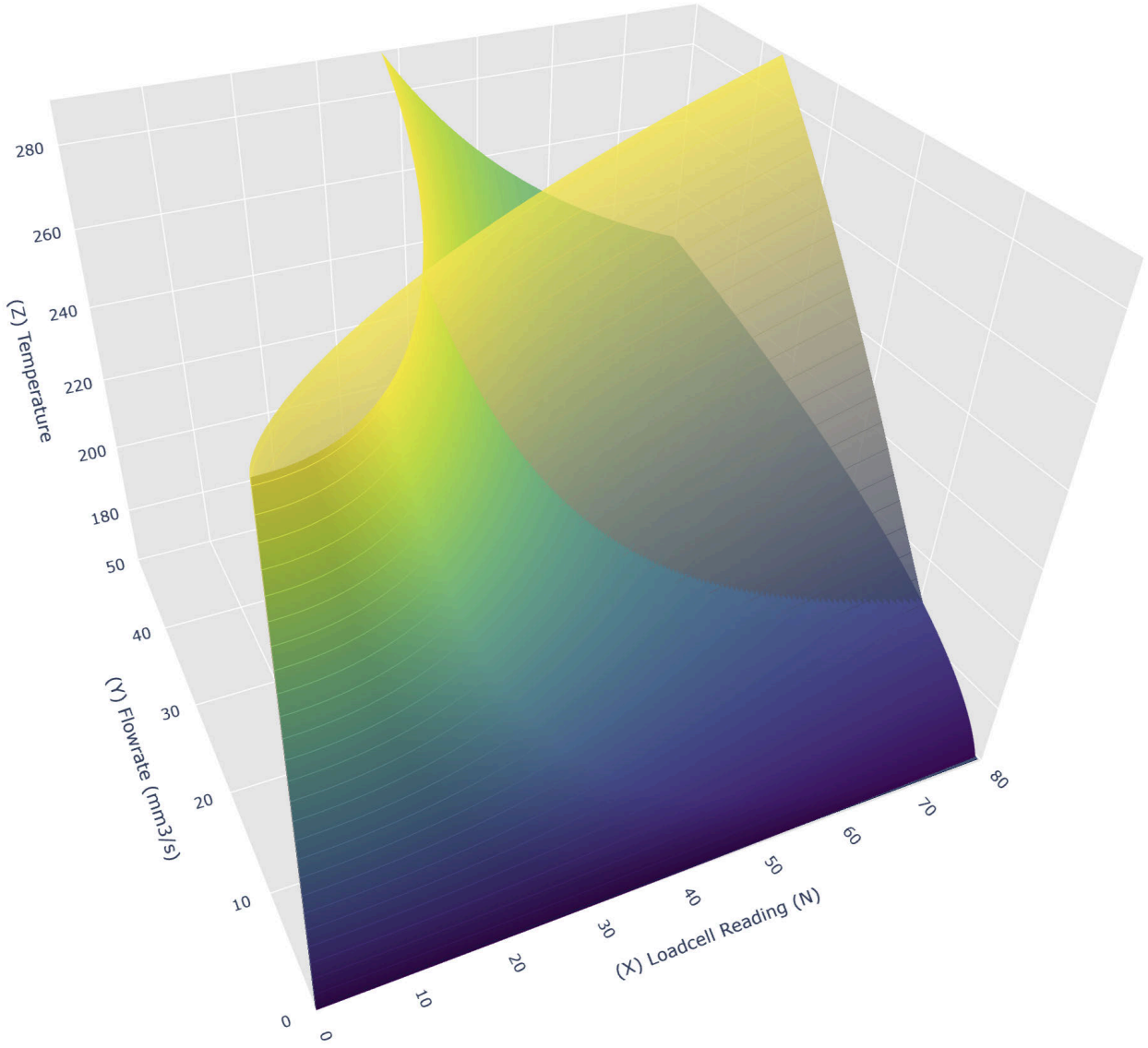


Figure 5.83: The steady-state model (cividis, faded) and the isothermal model (ciridis, opaque) overlapped in the space of nozzle pressures, flowrates, and temperatures (XY and Z axes).

5.11.7.3 Bridging, Ringing, and Other Missing Models

In Section 4.8.5 I mentioned that the models I use for motion control do *not* include machine stiffness and vibration, which turns out to be a major limit to overall performance. I apply the motion system heuristics (limiting deployable actuator current) partially to counter these effects; using less motive force naturally shrinks the amplitudes at which machine structures and transmissions are excited. Vibrations appear in 3D printing as "ringing;" vertical artefacts that occur near corners — sharp deceleration into the corner loads up the machine structure, which then oscillates as it relaxes while the machine prints the subsequent line.

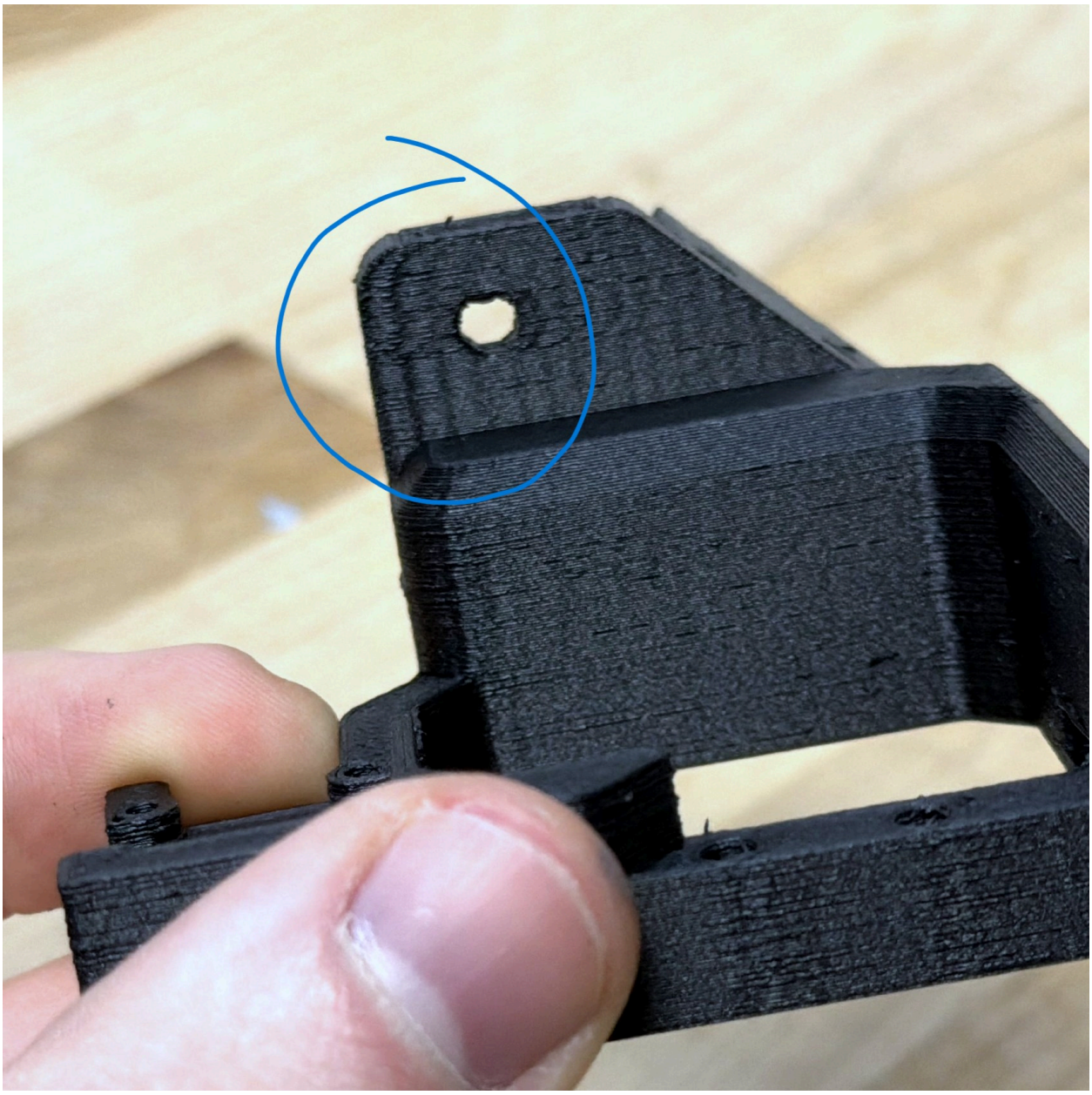


Figure 5.84: Ringing (highlighted here) happens when machines vibrate after making sharp accelerations through corners.

I show one example of ringing here in Figure 5.84. In Figure 4.32 from Section 4.8.5 another example of ringing is shown, where we can also see ringing in the dataset generated by the machine (as servo positioning errors).

Bridges in a 3D print are unsupported spans between two supported sections. These can be drastically improved by printing them slowly with maximum cooling while also slightly under-extruding, so that tracks *stretch* across the gap and are rapidly solidified. In the state-of-the-art these heuristics are applied to bridges that the slicer identifies geometrically.

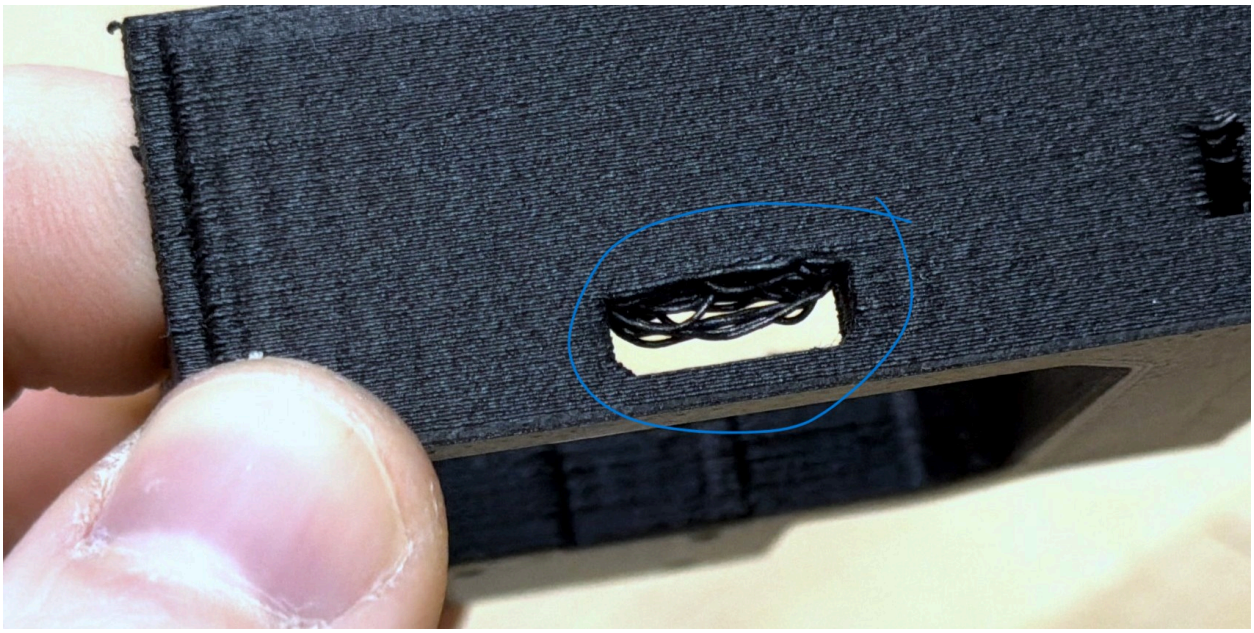


Figure 5.85: Bridges are unsupported print layers that span two supported sections.

There are many other heuristics that are applied by slicers, for example support materials are generated according to overhang angles, infill tracks overlap slightly with perimeters to improve bonding between the two, etc. These types of heuristic are related both to part geometry and to flow, but the workflow that I developed here misses many of them.

5.11.7.4 The Planner's Phase Space Representation is Poorly Suited for Flow Models

In Section 4.8.2 I discussed why solving the velocity planning problem for motion systems alone works well in phase space, where the problem is represented in discrete spatial steps that have varying time steps based on the chosen velocities.

However, in the current version of the 3D printer velocity planner, calculations that involve the (varying) Δt appear in multiple locations. In particular the melt flow temperature calculation is history dependent and the dynamics that describe filament compressibility and flow are based on time steps, not spatial steps. This means that the extruder system has the same difficulty that the motion system *would* have if it were formulated using fixed time intervals; its targets are effectively moving around after each update to $v(s)$. As a result, the allowable gradient descent step size is much lower in the printer system, which limits overall performance and can lead to "shaky" outputs when the planner fails to converge on a good solution in time.

It may be possible to reformulate parts of the extruder system to avoid this conflict — after all the amount of filament extruded in each spatial step should ultimately be the same even as the time scaling over those steps change. The thermal history is another story; although the energy *lost* in each step is equivalent to the volume of the filament extruded times the temperature of that filament, the energy added to the melt flow is certainly time-dependent. It may also be that properly articulating the whole thing as a classical direct transcription solver is best, as I discussed in Section 4.8.5.

5.11.7.5 Low-Level PID / PI Errors

Finally, I suspect that the motor controllers' low-level controllers (from Section 4.4.0.2, which are tuned by hand) contribute to the visual artefacts that can be seen in the prints from this chapter. It is obvious why a poorly tuned low-level position tracking controller would lead to some artefacts, and I discussed in Section 5.9.1 how the motor's velocity controller bandwidth limit requires that we impose additional constraints on the extruder motor's velocity and acceleration.

5.12 Future Work in FFF

5.12.1 Properly Combining Flow and Thermodynamic Models

In Section 5.5.4, I showed how I interpolate between steady-state and isothermal flow models. This step aids greatly in pressure prediction because FFF melt flows are, in practice, constantly changing temperature.

However, the modelling strategy that I use is somewhat lossy (see Section 5.11.7.2) and the models that I use on either end can't be fit on "generic" machine data; e.g. because the motion models that I use are not time-varying they can be fit against data from any set of time-series data that the machine produces, which means that they can be improved over time as the machine operates (see Section 4.7.3). For the flow models, we need to isolate data that approximates either steady-state or isothermal behaviour; the tests from Section 5.5 are designed to do so.

After building the velocity planner's flow temperature calculation step in Section 5.9.1.1, I can now see that modelling the system directly as a thermodynamic system *plus* a temperature varying model for *isothermal* flow is viable. Effectively we should develop a model that describes the melt flow temperature as a function of flow history and then use that internal estimate to fit the isothermal flow model. This is similar in practice to what I do already, but there I use the steady-state flow model (which is abstract) to approximate an estimate for temperature drop at any given average flowrate. Now I am proposing that instead of the interpolation, we fit a few additional free parameters to estimate the temperature drop: the filament's volumetric heat capacity (which I have already showed we can estimate, in Section 5.7.1), conductivity between the melt zone wall and the filament, and heat capacity and delay in the heater block itself.

I described models built by other researchers that are similar to what I am imagining in Section 5.2.4.1. [309] and [310] use FEA for this, but Pigeonneau's work [311] is probably the best place to start from because resolves into a more concise analytical description of the system whose parameters would be simpler to fit against data.

Modelling the thermodynamic system directly may have knock-on benefits. Its free parameters are values that we are interested in finding anyway; real thermal measurements for the heat block are an excellent basis for tuning the controllers that regulate them and both the filament's heat capacity *and* conductivity are important in the layer cooling model.

Because this introduces many new internal states, adding additional data sources may be valuable; that could involve modifying extruder designs to add a second thermocouple closer to the nozzle to better estimate the heater block properties, as is done by [308] to more accurately model flows. [274] does the same and also adds thermal imaging. I implemented a small microbolometer on the FrankenPrusa's hotend but didn't use it in any capacity beyond just looking at the videos it produced, [274] and [318] both use thermal imaging to estimate internal flow temperatures.

Improved modelling of heaters themselves is also critical for the development of feed-forward terms in thermal controllers that would enable predictive control of hotend thermodynamics. In the state-of-the-art we *rely* on the large heat capacity of the heater block to make flows temperatures more predictable, but there is ongoing work in developing *inductive* nozzles for FFF. These have much less thermal mass, and so can change temperature much faster. I expect that they will become the new state-of-the-art over the next few years and will require better thermal control.

Given that *cooling* is a major driver of overall print time and that *nozzle temperature* has the largest overall effect on flow dynamics, the reduced thermal mass of an inductive nozzle (and their higher power and more *even* heating) also presents a major opportunity. Fast thermal slew rates could be used to optimize nozzle temperatures over the course of a print; larger layers printed at warmer temperatures, smaller layers with colder temperatures. If thermal slew-rates are fast enough, we could even ramp nozzle temperature *along with acceleration* to produce prints with roughly equal extrusion pressure throughout or e.g. use hot and fast tracks during infill and more precise extrusion on perimeters, etc. In Section 5.11.5 I showed that we can connect these types of time-varying dynamics to motion control, so connecting these two systems seems immanently possible and would be invaluable if we are to move towards *"Pareto frontier printing"* (Section 5.12.4).

5.12.2 Continuous Improvement and Deployment

One of the things that makes this system unique is that the measurement equipment and the process equipment are the same. In most materials modeling paradigms, we use offline hardware to characterize samples, develop process parameters for feed-forward controllers, and then deploy them. In this case, we use much lower fidelity instrumentation but can use it continuously to generate "an embarrassment" of data [333], about one gigabyte per hour.

Section 4.7.3 shows that we can successfully re-fit motion models from data generated during machine operation. This is particularly valuable for machines whose damping and inertial terms may change over time as e.g., grease dries out, belts are tightened or loosen, or (in a milling machine) when masses change even over short time spans. We should be able to do the same for our process models, continually refining flow, cooling, and extruder models as we amass more data. This is limited mostly by the flow models themselves as I just described in Section 5.12.1, but there are a few more things to consider.

The first is that much of the data generated will be redundant; if we print the same part tens of times, we need methods to apply updates to models without endlessly filling hard drives. Models are made for this (they are effectively compressed representations of observable phenomenology…), but when we re-fit we would want to keep some older samples to avoid over-fitting only in the states represented by new data. This is basically a normalization problem, for which there is well established practice.

The other consideration is about how to separate measurements that represent model *updates* from data that represents *errors* in the system; both are cases where new data fit poorly to old models, but mis-fits here could either represent unknown behaviour *or* external disturbances like

part failures, material changes, or machine changes. This could be done by separating deltas over various time scales: changes that happen over hours and days may indicate machine maintenance issues, whereas changes that happen in milliseconds or seconds may indicate disturbances. Changes that happen when we load a new *batch* of material could indicate that something in the material supply chain has been modified (even if that has not been made clear).

5.12.3 Smaller Improvements to the System

5.12.3.1 Real-Time Closed-Loop Pressure

I mentioned in Section 5.9.1 that (with good enough flow models) a good avenue to control new FFF machines would involve closing the loop over the melt flow pressure. In this case we would eschew driving the extruder motor directly and instead send the extruder system a target pressure, building a lower level controller to update extruder motor torques such that the target pressure is reached. Pinyi Wu made good progress in servo control for an extruder motor in [312], but he doesn't connect the extruder dynamics back into the velocity planner or use pressure-based control (his extruder tracks a velocity signal from an existing planner and modifies it according to filament compression dynamics alone).

Doing this properly would still require predictive control such that the pressure reference signal does not exceed the system's physical bandwidth, and would require that the complete pressure-to-flow system is accurately modelled. As we have seen that is highly dependent on hidden internal states like the melt flow temperature; if our models estimate the flow temperature to be 200° C but it is really 240° C the same pressure target will generate much more flow.

So, this leaves us with a real question: how do we combine control modes or e.g. use open-loop control as our models are developed and then transition towards closed-loop control. Another exciting approach would involve measuring track width in real-time to control extrusion against that signal directly, etc. Pressure-based control may be particularly well suited for screw-type extruders that are used in larger extrusion systems [334] which are more difficult to tune in open-loop because they are not direct displacement pumps, i.e. a single turn of the screw does not correlate linearly to a single unit of flow from the nozzle.

5.12.3.2 The Filament Sensor

I added a filament sensor to the RheoPrinter that measure inlet flowrate alongside the filament's real width using a passive idler wheel (connected to an encoder) with a preloading wheel whose displacement can be measured precisely; see Figure 5.11. The design is based on Thomas Sanladerer's [325]. I originally used this for steady-state flow tests in [305] where the extruder motor was run entirely open-loop and its teeth would shear material when flow stopped. With the closed-loop motor I can instead limit motor torque to values that won't shear materials.

But this sensor provides an opportunity to control flow based on the measured width of the incoming material rather than relying on upstream manufacturing to produce high precision filaments, which increases costs. In the state-of-the-art filament manufacturing must be done very

consistently, e.g. each roll of Prusa filament is inspected with a laser width sensor at the factory [290].

5.12.3.3 Die Swell and Cooling Models

In Section 5.2.4.3 I mentioned some ongoing research efforts to model filament behaviour *after* it leaves the nozzle and in Sections 5.2.1.2 and 5.2.3.4 I described the importance of layer cooling overall and how it is managed in the state-of-the-art. In this work, time-optimal nozzle temperature selection which is *based* on cooling models (Section 5.8.4) provides some of the biggest benefits (Section 5.11.4).

So, connecting these two sides of the problem (dynamics in the nozzle to part formation outside the nozzle) is an obviously interesting area for future work, and there is a deep connection between how melt flows are "disentangled" within the nozzle and their adhesion to one another outside of the nozzle, as noted by [319]. In [316] we see that extrusion pressure / shear rate is also correlated with die swell, which is a widening of extruded tracks that occurs mostly when we extrude polymers when they're not completely molten or are extruded at extremely high pressures.

These aspects of extrusion may indicate that there is a *second* upper limit to flowrates and melt flow temperatures, besides those that we can already see that are based on pressure generation and heating of the filament. Some materials also degrade when they are heated past certain temperatures, representing another important boundary.

Here again the microbolometer is likely a valuable instrument, e.g. [316], [318] and [274] all use thermal imaging to improve both heating and cooling of individual tracks of filament.

Thermal data has an obvious utility in estimating how layers cool; the methods that I develop in Section 5.7.2 use estimates for heat transfer to air and into the layer below. Those could be supplanted with data fit from the microbolometer. In fact because its frames are time-synchronized to the motion system, we should be able to use them to recreate a 3D thermal history of the printed

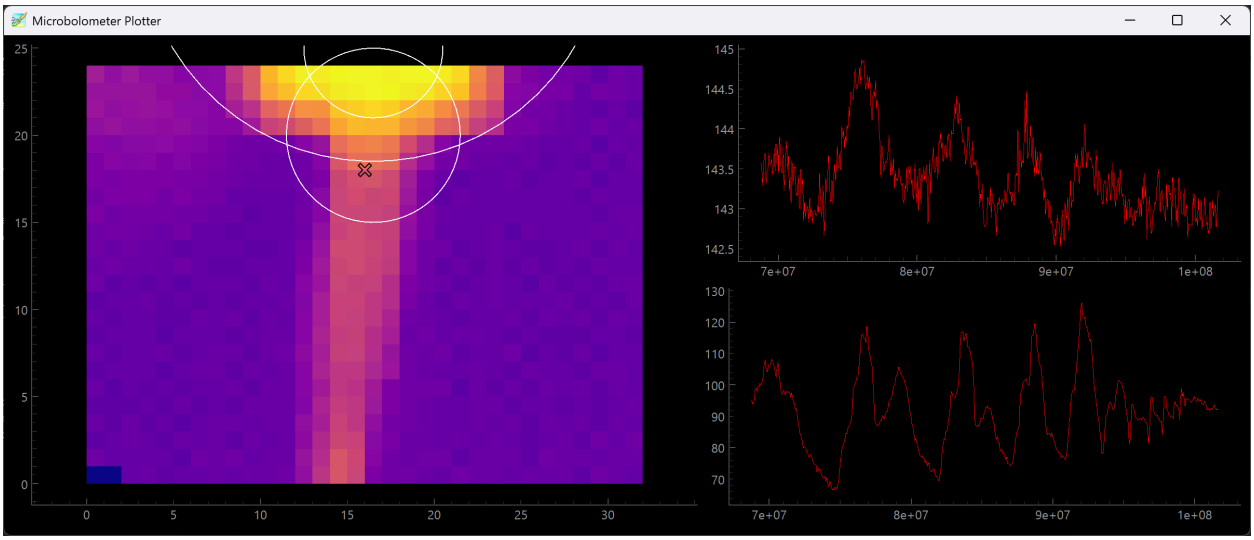


Figure 5.86: I made an attempt (with Quentin Bolsée) to measure die swell using the FrankenPrusa's microbolometer, but its limited resolution made this difficult.

part. It may also be easier to develop simpler strategies like measuring the average temperature of a part surface under variable fan parameters and geometries.

5.12.4 Pareto Frontier Printing

Finally, and the most exciting way to improve the process overall, is the prospect of formulating the FFF workflow on the basis of Pareto optimality across underlying trade-offs between precision, strength, and speed.

For example the key *tuning* knob in the optimum nozzle temperature step here is picking our target layer temperature T_{targ} — if we favor "hot and fast" prints that may be somewhat slumped (but stronger due to improved layer adhesion), we can dial this up: the minimum-time optimization will be penalized less for warmer flows (as minimum layer times will be shorter). If we want precision above all, we can turn this down and we will see the opposite result. This makes up one axis in a theoretical Pareto frontier, which trades strength (at the cost of precision) for speed. If we want strength *and* precision, we would print layers hot but also slow them down to avoid slumping. Similar trade-offs extend into the motion system; using more motive force there increases acceleration and cornering velocities but excites mechanical vibrations in the machine and requires that motors track higher bandwidth signals.

This is a classic "pick two" scenario: we can have precision and strength but not with speed, or speed and strength but not precision. By "Pareto optimality" we mean that, given two or more objectives, we must decrease the optimal outcome of one in order to improve the outcome of the other. In between these trade-offs there is still a "Pareto frontier" that is the most optimal outcome overall, given the constraints of the underlying problem. Originally developed in economics [335] it often appears in engineering and mathematics [336] to describe these types of trade-offs.

In FFF printing we could formalize trade-offs for precision, speed and layer strength by defining optimal results for each. Optimal speed is obvious: we measure total time. Overall precision is a strong function of our cooling target temperature, motion system and also flow system. Layer adhesion / part strength is a function of the layer interfaces' thermal history [292]. What is missing from the current work is a characterization of *how much* layer strength is gained from a given increase in interface temperature / time, and *how much* precision is gained from slowing down and over-cooling parts. These are all dependent on physical models that change from machine to machine and across materials.

What is perhaps more difficult than modelling how these trade-offs interact physically is choosing how to articulate them to machine operators. For one example, rendering systems in CGI sometimes let users say e.g. "I have five hours maximum to wait for this render to complete, render at as high of a resolution is possible in that timeframe;" we can imagine the utility of making the same formulation for FFF 3D printing. For precision users could articulate how much deviation they would allow between the printed part and their CAD model, but interlayer adhesion may be the most difficult to express and to model. Rather than specifying target values directly, a simpler interface may simply allow users to pick weights within $0 \rightarrow 1$ for each trade-off.

Overall, I think the future of FFF printing is bright. I know that there are many others working in this domain, and (luckily for the public good) many of the critical patents have expired[95]. Despite the onslaught of new strategies being developed for additive (laser bed fusion, binder jetting, etc.), FFF remains extremely useful, relatively easy, cheap, and capable of producing excellent parts. In my research group at the Center for Bits and Atoms we have access to a long list of advanced equipment, some of which are worth hundreds of thousands of dollars each. We also have a wall of twelve FFF 3D printers, each worth around one thousand dollars. Measured in total output of new ideas tested, they are the most productive pieces of equipment in the lab.

[95]For example Stratasys held a patent on *enclosed build chambers* and so for about 15 years most FFF companies couldn't put their machine in a box (which greatly improves part warping, and makes the printer less susceptible to i.e. a drafty room). Some readers may be wondering why I keep calling it *FFF* - well, the other acronym is trademarked by the same company. This is all fair game in today's patent law ecosystem, but a clear case where our old laws (developed in an era where technological advancement was more costly and took place in longer cycles) have become outdated. Not to mention a changing global order, especially in manufacturing, where some of the main participants are developing equipment within completely different legal frameworks - given the difference in the two worlds' approaches to IP law it is no surprise to me that they are able to innovate at such an outstanding pace. Notes should be taken…

6 Model-Based Development and Use of a CNC Milling Machine

6.1 Introduction

CNC milling[96] machines are at the heart of our manufacturing economy. In fact, GCode was originally developed for their use before being extended to other processes like turning, laser cutting, 3D printing, etc. We use them to make aircraft parts, car parts, molds, fixtures and tooling, surgery robots, spacecraft, gearboxes in wind turbines, scientific instruments, the list is endless. Perhaps most relevant for readers of this thesis: CNC Mills are often called *mother machines* because they are where we start when we want to build more machines.

In the introduction I mentioned that there are many unfilled manufacturing jobs in the west. Many of those roles are in CNC programming. In the industry, "CNC programming" doesn't mean writing machine controllers, it means writing GCodes to run on CNC machines (using CAM software). Training new GCode programmers takes a long time because they must absorb the tacit knowledge required for the task. Understanding what will and won't work in a CNC toolpath is especially hampered by the *hidden optimization* (Section 1.3.3): cutting physics is particularly sensitive to changes in translational rates due to machine acceleration. Designing CNC milling machines also remains an art; actual machine performance is the result of interactions between motors, machine physics, and controllers themselves. This might limit the rate at which new CNC firms enter the market, or the rate at which existing firms develop new models.

Many parts of this thesis revolve around the idea that we can formulate machine control as a *constrained optimization* problem. Model building is a prerequisite for machine controllers that are explicitly framed as constrained optimization solvers, and I demonstrated how these can be developed for motion control (Chapter 4) and for process and motion control as a coupled task in Chapter 5. I also discussed how models can be used as tools for machine users, enabling them to more easily operate and understand their machines (Section 5.10.3 and Section 5.11.1) and their materials (Section 5.11.3).

Here I want to do a few things. The first is to show that we can rapidly deploy the controllers that I developed in Chapter 2 on new hardware. The second is to show and then demonstrate one simple way that visualizing velocity plans can be valuable, and how models and machine data can be combined to make new measurements. Here I organize that around the idea that models can be used as a part of human-in-the-loop workflows for CNC programming and machine design.

[96]Note that while "CNC" stands just for Computer Numeric Control and technically i.e. a 3D printer is a CNC Tool, in practice we often simply use "CNC" to mean CNC machining.

First, I will cover the basics of machining physics in Section 6.1.1, and explain the state-of-the-art workflow in Section 6.1.2. I will also discuss existing work in the development of models for CNC in Section 6.1.4. I will then explain my methods for model-based design of a CNC machine in Section 6.2, and model-based operation of the same (6.3).

In the results section, I will show how motor models were used to evaluate trade-offs in motor selection for a machine (Section 6.4.2), and look at machined components and data from this machine's operation (Section 6.4.3). I will show how models and our unique control architecture can be used to expose the *hidden optimization* to machine programmers in Section 6.3.1, and how those models can be combined with real-world measurements to estimate cutting force data from motor data (6.3.2).

To close the chapter, I will share and discuss the results in Section 6.4, and look ahead to the possibility of taking on CNC machining as an explicit constrained optimization problem (Section 6.5).

6.1.1 CNC Physics

The physics of machining are arguably more intense than their counterparts in FFF, involving extreme shear rates, thermodynamics, tool life and geometry. The *core* of it is about chip load, which depends on the ratio between spindle RPM and translational velocity. Cutting an oversized chip will generate too much cutter force (leading to broken tools, a stalled spindle or actuator

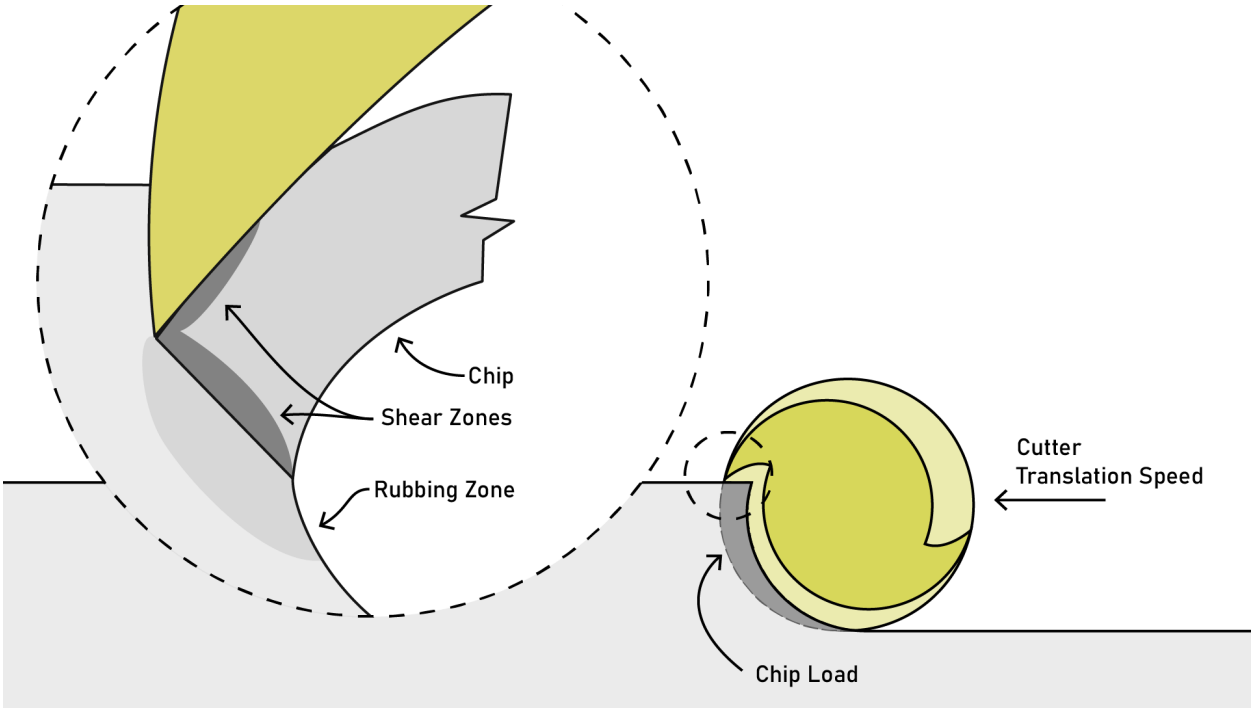


Figure 6.1: This figure diagrams the basic physics happening when a CNC machine cuts through metal. The cutter (yellow) has a number of edges (called *flutes,* two in this case) and spins at some user-determined RPM. It is moved through the material at a *translational rate,* such that each revolving flute engages with a chip of a set width (called the *chip load*).

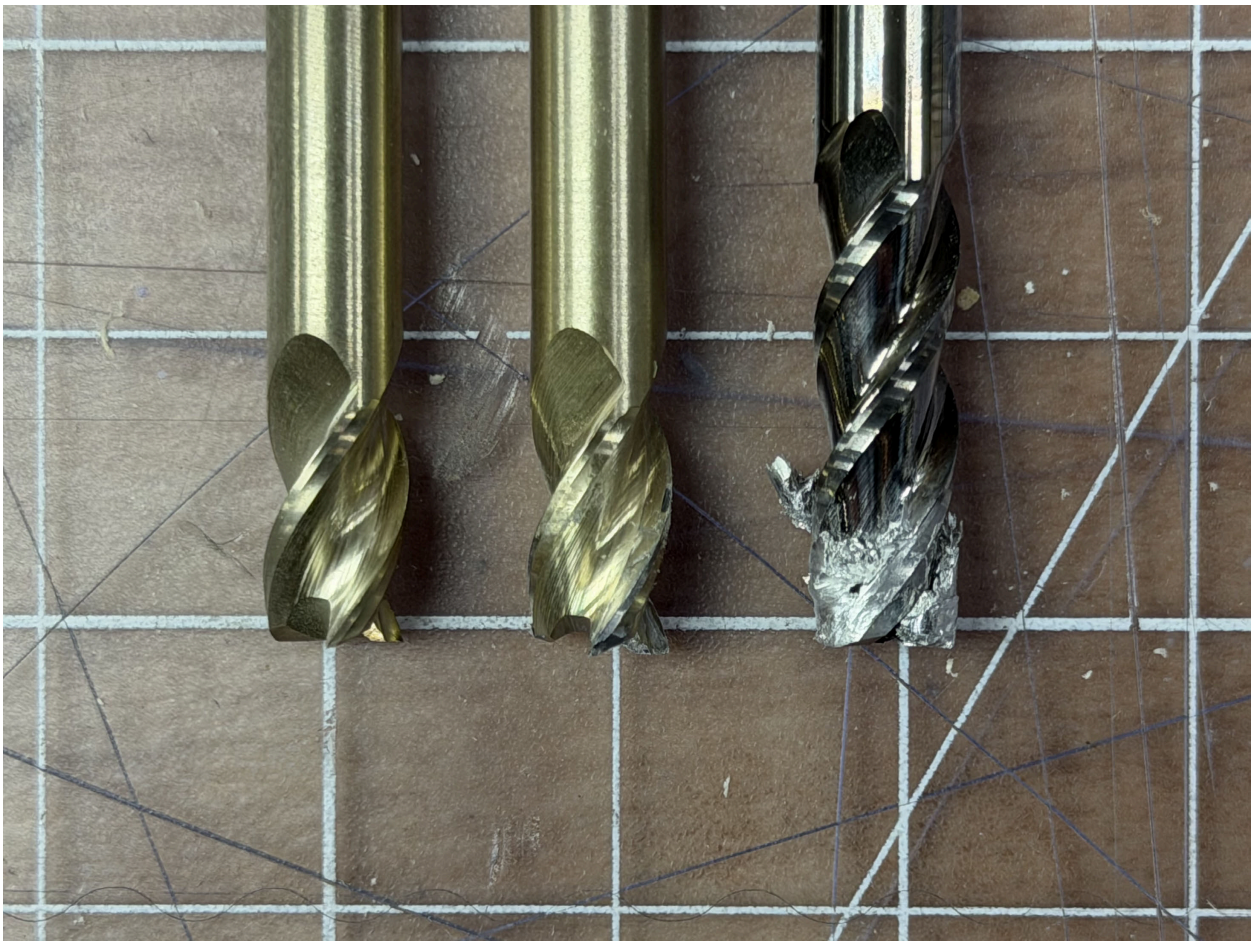


Figure 6.2: CNC endmills from left to right: fresh, used with small built-up edge formation, and crashed due to undersized chip loading (which leads to melting, rather than cutting).

motor, or egregious excitation of the machine or part's resonance). Cutting an undersized chip (smaller than or close to size of the flute's tip radius) means that the *cutting* physics become *rubbing* physics: this generates an exceptional amount of heat, dulls tools or leads to material buildup on the flute edge, and in some cases can weld the cutter and the part together which almost always leads to a machine crash.

Another important parameter is *surface speed,* which is the speed which the flute edge passes by the material being cut. It is a function of tool diameter and spindle RPM. Physically, this is related to the *shear rate* in the material as it is being separated from the work piece. Changes in shear rate affect chip formation: low shear rates tend to make larger chips (which can be difficult to evacuate from the work piece and can lead to "chip re-cutting"). Larger shear rates exert more total stress in the chip, leading it to break up into smaller chips that are more easily evacuated. The ideal shear rate for a cut is mostly a function of the material being cut *and* it's temperature, but also of the tool geometry, sharpness and surface finish.

These physics all exert loads on the machine itself, it's actuators, and the workpiece. Actuators must produce enough force to push the cutter through the piece, and spindle motors must produce enough torque to keep the tool rotating at the desired speed.

Most complex of all in CNC machining is the issue of resonance: any machine has a given natural frequency and if chip formation happens to excite the machine near that frequency, the whole system vibrates, causing inaccuracies or even tool breakage.

6.1.2 The State-of-the-Art CNC Workflow

CAM for 3D printing (aka 'slicing,' Section 5.2.2) is much simpler than CNC machining because each job is effectively one operation: slice into layers, produce perimeters and then infill for each layer. That is, same basic geometric algorithms can be applied to just about every different part. A machining job can include many (tens or hundreds) of different path planning strategies for different geometric features, each selected manually. I show a selection of these strategies in Figure 6.3.

This also applies to tool selection. For example there is an obvious trade-off between large removal rates using big tools and detailed operations using smaller tools. There are also special cases; a tightly dimensioned hole may require the machinist to select a purpose-made reaming tool, threaded features require threading tools, blended or chamfered corners require fillet or chamfer end-mills, etc.

CNC CAM *is* similar to FFF Slicing where parameter selection is concerned. Once geometry and tools are selected, users must tune a litany of speed and cutting parameters. These define how fast the tool should move through the part, how much material should be cut with each pass (step over and step down), and how fast the spindle should run. These are known colloquially as *speeds and feeds* and affect cutting physics in the ways mentioned above. Starter values for feeds and speeds can normally be calculated from tabulated values for *surface speed* and *chip load,* but even tabulated windows can be large and need to be heuristically tuned.

Milling parameters, then, are similar to FFF parameters: a user sets geometric and speed-related parameters that are only indirectly related to the underlying process physics that govern the system. Instead, they are directly related to the low-level instructions (GCodes) that we send to the machine.

In all other ways, the state-of-the-art CNC workflow is similar to FFF. Users import part geometry into a CAM package, configure geometric and low-level parameters, export GCodes, and run them on their machines. They watch the machine and listen to it, iterate on their parameters, and make incremental improvements to their selections using their own intuitions for the process and machine physics.

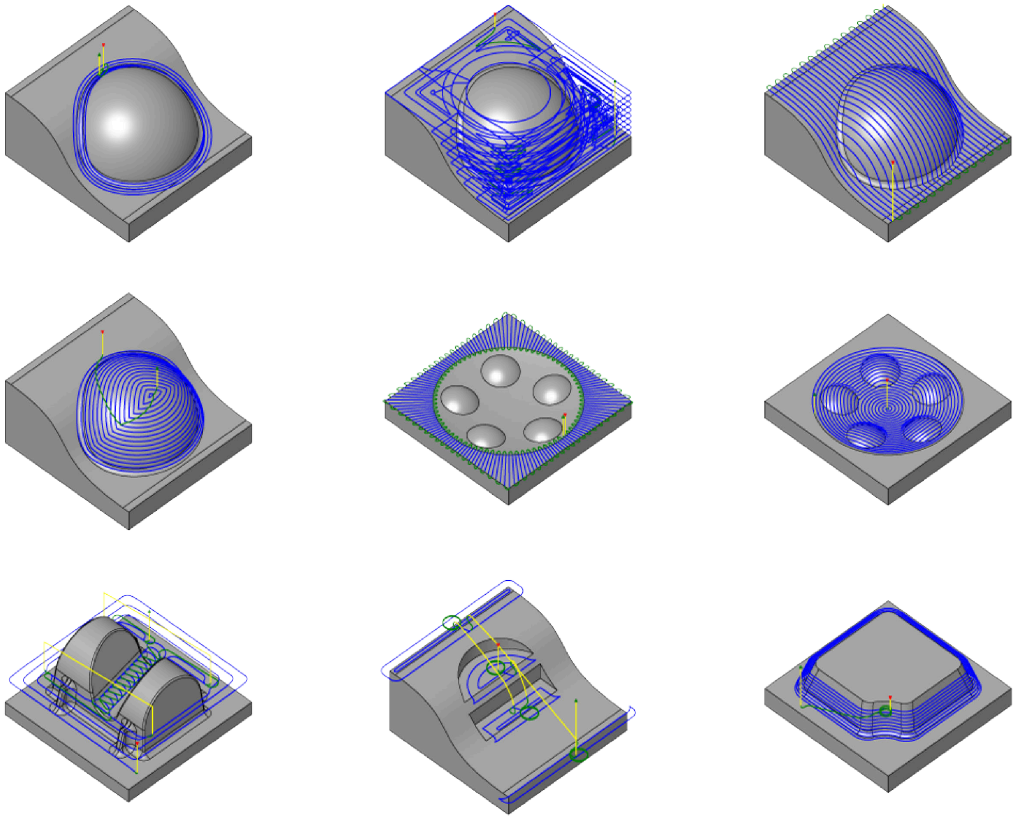


Figure 6.3: These are just a few of the many CNC machining *strategies* available in one modern CAM tool (Autodesk Fusion). Each strategy has some unique parameters to tune, though most re-use a similar subset.

6.1.3 Interaction of Cutting Parameters and Acceleration Control

A key focus in this chapter will be the interaction between machine controllers and cutter parameters, namely chip load.

With CNC milling, decelerating into a corner can change chip size dramatically. Machines need to slow down through corners to avoid violating their actuator constraints, but they manage to decelerate and accelerate at rates quite quickly. Spindles have very large rotational inertias, so we cannot rapidly slow them down to maintain chip load as the machine's translational velocity changes. Instead, we simply have a lossy layer here. We may set up a job to run at a particular chip size, but tight corners in the job will be done at much lower speeds with a much smaller chip

[97]Many readers will be familiar with High-Speed-Machining, a series of milling strategies developed somewhat recently. These generate tool-paths that effectively minimize cornering radii for the majority of the work that they do. These were a kind of magic when they were introduced, and were realizable only when modern machine controllers could readily handle the complexity of the instructions produced. It is my intuition that this insight was not so much about maintaining high feed rates, but that the resulting geometries mean that *more of the actual*

size[97]. CNC Controllers don't expose this acceleration control layer to users, and it is not specified in GCodes.

6.1.4 The State-of-the-Art in Model-Based CNC

Because most research in model-based motion control is related to the optimization of milling machines, I have already covered most of the relevant research background for machining in Section 4.2. Specifically Sections 4.2.4, 4.2.5 and 4.2.6 for the industrial state-of-the-art and then Section 4.2.8 for the state-of-the-art in research. Section 2.6.2 compares control systems architectures used by other CNC machining researchers to those that I developed in Chapter 2. Here there are just two small additions.

6.1.4.1 Estimating Cutting Force

While the physics that generate cutting forces are intense (high shear rates, plastic deformation, heat and mass flow, etc.), we only need to model those physics if we want to look at cutting forces in small length scales: i.e. understand chip formation and evacuation, tool life, etc. In our case, we just want to understand the loads seen by the machine. A former CBA undergraduate researcher Chetan Sharma wrote his thesis on the topic [337], I was also able to find another masters' dissertation: [338]. Both outline relatively simple models that output radial and tangential cutting forces as a function of straightforward geometric parameters (cutter size, cut depth, spindle rpm and linear feed rate), with four coefficients to fit for cutting force and edge force coefficients, each with tangential and radial coefficient respectively.

$$
\begin{aligned}
F_t &= K_{tc} a h(\theta) + K_{te} a \\
F_r &= K_{rc} a h(\theta) + K_{re} a \\
\\
a &= \text{cut area} \\
h &= \text{cut height} \\
\theta &= \text{chip thickness} \\
K_{tc}, K_{rc} &= \text{cut force coefficients} \\
K_{te}, K_{re} &= \text{edge force coefficients}
\end{aligned}
\tag{6.1}
$$

[337] shows that coefficients can be estimated using data generated on-machine, using measured spindle power and force exerted on the workpiece (a load cell). There are a number of other efforts that replicate similar work, some of which are in Section 4.2.8.1 as I mentioned and a complete review is in [339]. In particular I would note that [191] and [192] both tackle the problem of identifying the a, h, θ parameters from Equation (6.1) from part geometry, tool geometry, and toolpath interpolation.

So, estimating cut forces is not itself a new contribution, but other authors went to some length to build systems specifically for this purpose. In this case I extend data that the machine controllers

milling time is spent *at the target feed rate* - i.e. tight corners, which produce feed rates lower than those set in parameters, were minimized.

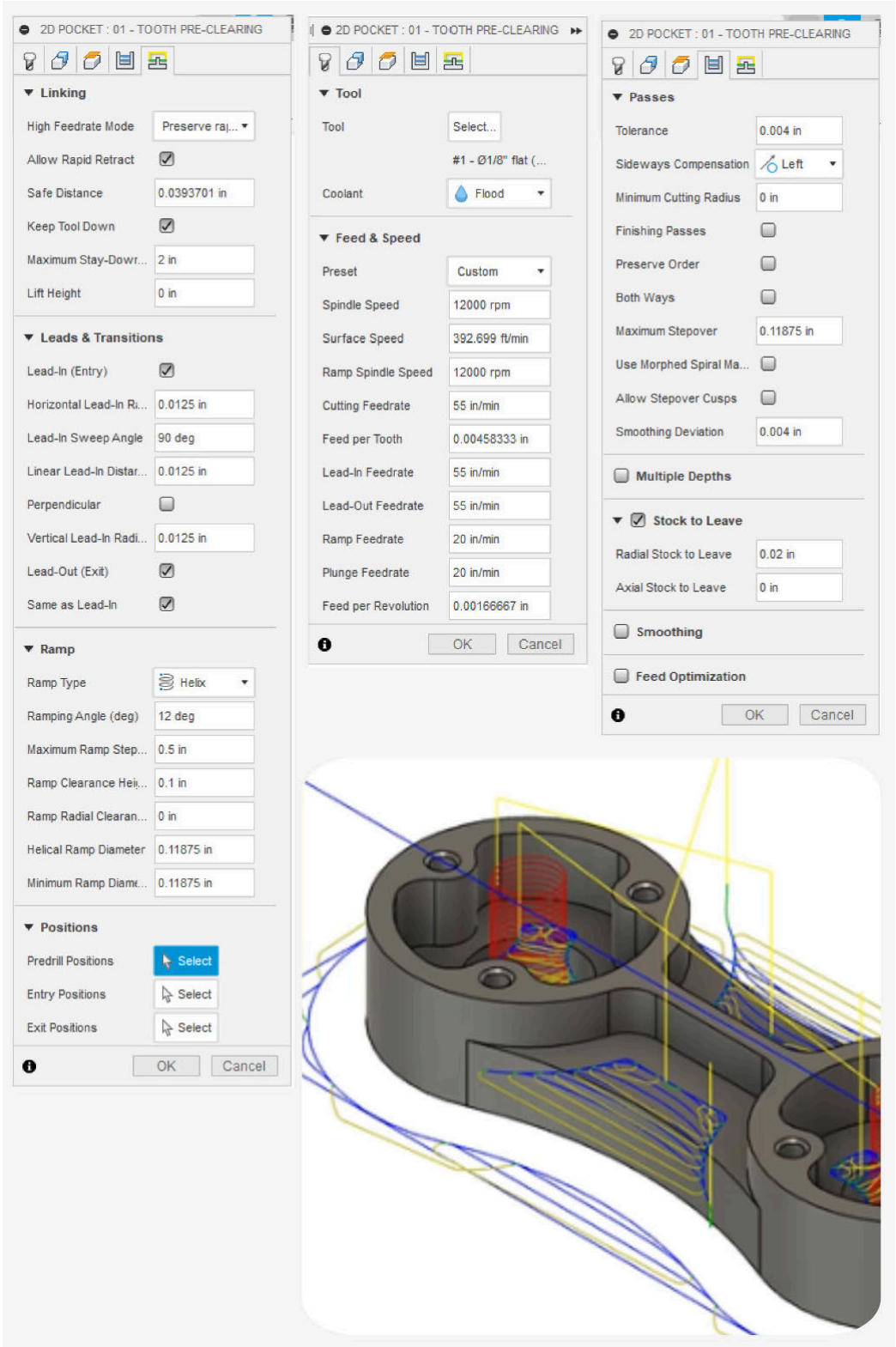


Figure 6.4: This is a screenshot of the parameters available to configure one cutting strategy (a 2D Pocket) in Autodesk Fusion.

from Chapter 2 already generate, and combine those with models from Chapter 4 to infer cutting forces.

6.1.4.2 Measuring Machine Stiffness and Resonance

In practice *chatter* causes more displacement than static loading. [338] adds resonance measurements: he uses a solenoid hammer to generate a step response of the tool, measuring the tools 'ring' using an inductive probe as a displacement sensor. Using this system, he generates a stability plot across spindle RPMs and depths of cut. In [340], the authors develop a system that samples from a Heidenhain machine controller at 10kHz to detect chatter in real-time using motor currents, and [277] develop a model-based adaptive controller to mitigate chatter. [191] implements a "machining digital twin" to optimize feed rates of a milling center in real-time to predict and automatically prevent chatter. Finally, for a broader overview of machining dynamics, we also have [341] and [342].

In my progress so far I have only used machine data to estimate cutting forces, but I think these works are worth mentioning here again because they show that the types of data I can produce using the systems developed here could be extended to model these physics.

6.2 Machine Design Tools

When we develop machines, we must select from discrete families of off-the-shelf parts with limited customization (especially if we are a small firm or individual with small order quantities). Of particular interest is motor selection, where we must consider characteristics of our motors *and* our machine's kinematics since both affect the final performance.

For example, motor physics makes for natural trade-offs between availability of low- and high-end torque (Section 4.8.4). Motor parameters that help us understand these trade-offs are often available from manufacturers, *but not always,* and as I discuss in 4.8.4 even two identical motors (same make and model) may have slightly different parameters. The most useful motor selection tool that is normally available from manufacturers is the motor torque curve (where we see how much force the motor can generate across a range of speeds), but torque curves can change depending on motor drive voltage and controller type.

It is also sometimes difficult to understand how motor torque curves might couple into a machine's dynamics, through its drive system (belts, leadscrews, linkages, masses, and frictions etc.). At MIT, spreadsheets are the state-of-the-art design tool of choice for this step [289]. Spreadsheets are excellent design tools, but they tend to require that we iteratively step through dynamical states (acceleration, velocity) to understand performance, and require that we use multiple spreadsheets to understand each component of our machine. They also require that we input model parameters manually, and they are of course not complete simulation tools that can be subjected to virtual operation.

In this section, I develop tools that allow us to virtually swap motors onto simulated hardware to analyze machine performance across a range of parameters. We do this using both abstract parameter spaces and using real toolpaths. I discuss the use of these tools in 6.4.2.

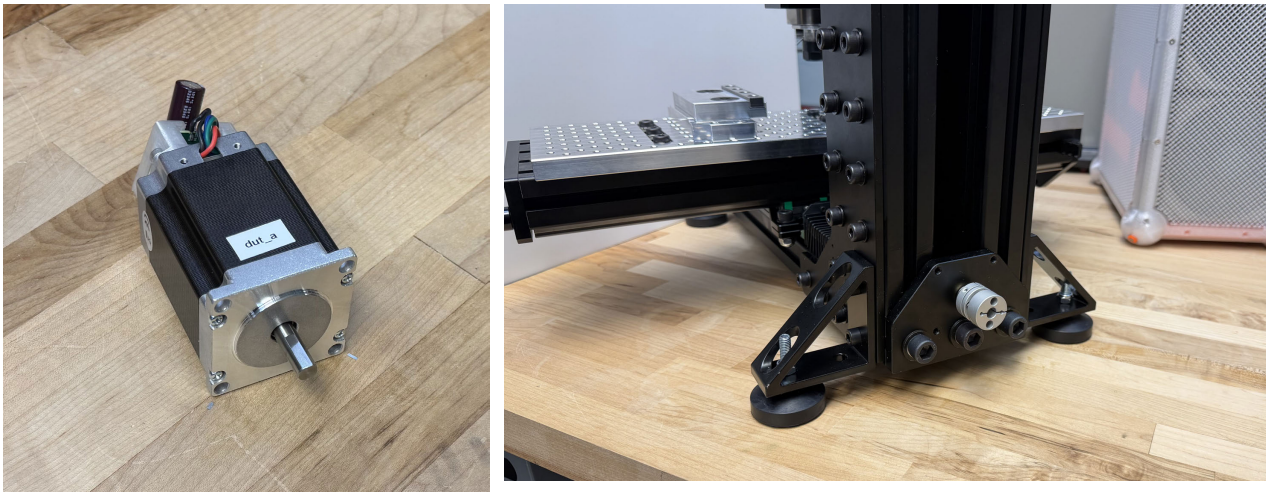


Figure 6.5: Here we combine motor models (from left) and kinematic models (right), to generate **force margin** plots.

6.2.1 Virtually Comparing Motor and Machine Dynamics

In Section 4.3.1, I showed how we can develop motor models using a closed-loop motor controller and simple test inertia. In Section 4.3.2, I showed how we can use those models to develop kinematic models of machines that include mass and damping terms.

Both of those models remain separable, which means that given a set of motor models and a machine's kinematic model, we can virtually compare performance across motor options.

CNC Machines need to move around *while* providing cutting forces. To compare motor selections in this context, I developed a *force margin* analysis tool. Here, motor models define how much current can be generated in the motor stator as a function of the rotor speed. Using this model, and the motors' k_t value (A/Nm), we can estimate the amount of torque generated across these speeds. We can then combine this estimate with the kinematic model's estimate for force required to accelerate the mass (just $F = MA$) and use the model's damping terms to estimate forces lost to friction.

The tool takes a maximum current saturation parameter, and generates a *force margin* (maximum available force from the motor, less forces taken up by motion) across a grid of velocities and accelerations.

This plot can help us to select motors for our machine, and also forms the basis for the next tool, which can help us select operating parameters.

6.2.2 Selecting Motion Control Parameters using Models

Using force bandwidth plots (above), we can select (max accel, max velocity) parameter for our motion controller such that we guarantee a force bandwidth overhead in Newtons.

This is for use in a trapezoidal motion controller, which cannot use the entirety of the motor's dynamic range (Section 4.7.1). That leaves us with a trade-off: high-end speed or low-end force.

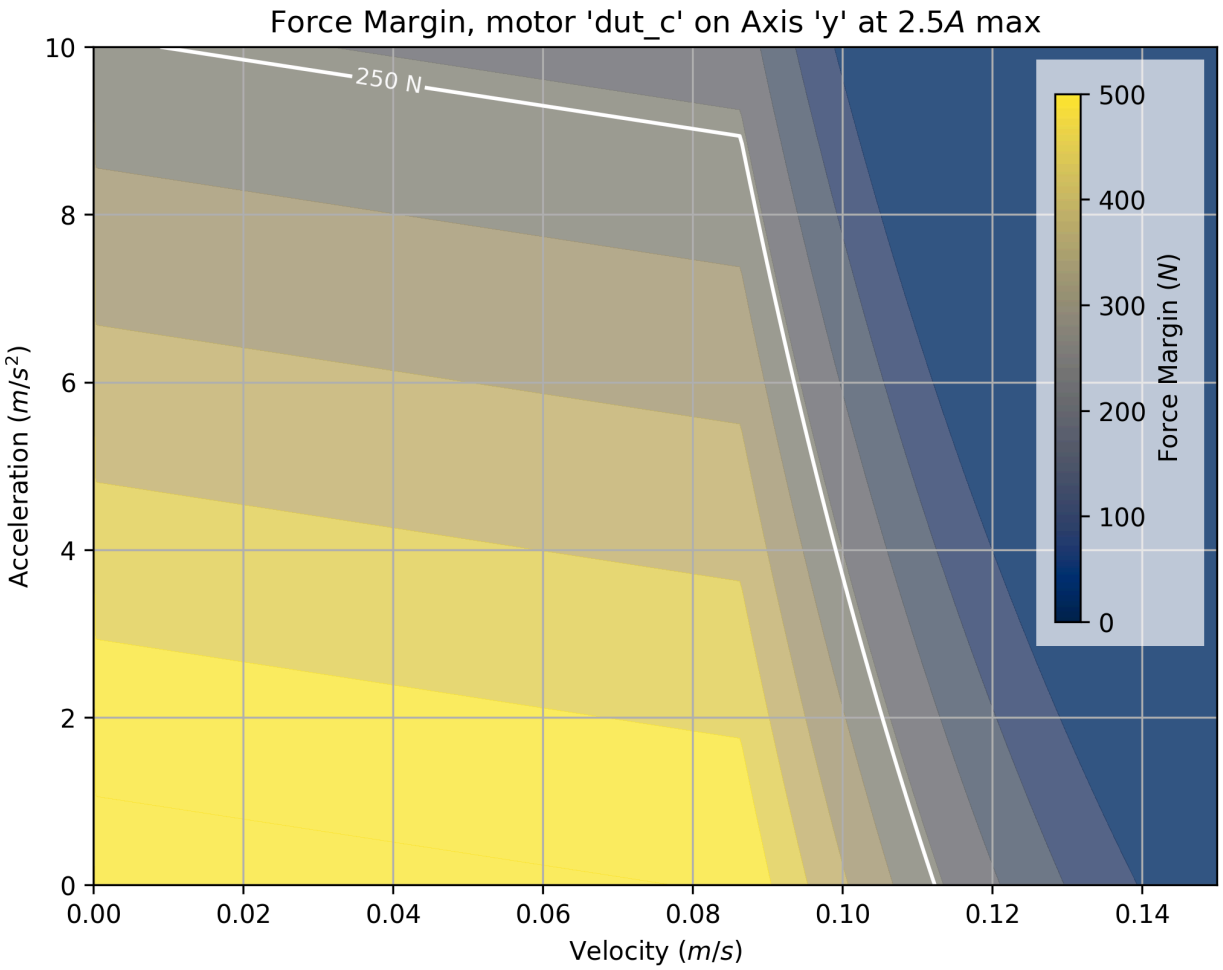


Figure 6.6: A one-axis force margin plot, which is mapped over axis velocities $(x, m/s)$ and accelerations $(y, m/s^2)$ with a contour (color map, N) rendering the force available from our motion system under those accelerations and velocities.

These are the core methods for selecting motors and motion parameters using models. I show how I used these methods together for our milling machine in Section 6.4.2.

6.3 Toolpath Inspection Tools

6.3.1 Visualizing Real Chipload

In this thesis I have discussed the idea of *constrained optimization* at length. Machines are limited by the physics of their motion systems and those are coupled into process physics. In particular I mentioned a *hidden optimization* in Section 1.3.3, where the *velocity* of a path plan is modified in real-time by machine controllers in order to avoid violating actuator and kinematic constraints.

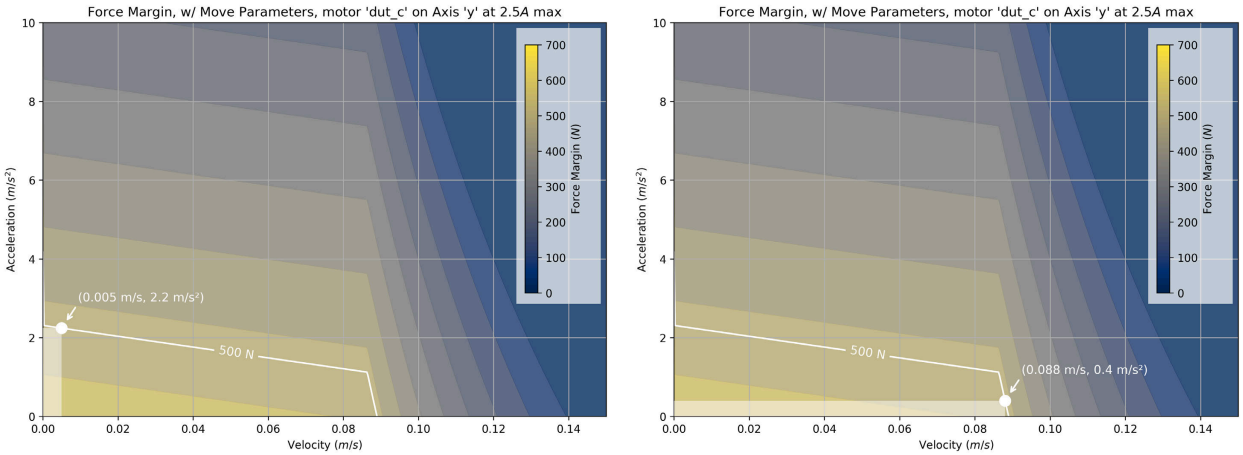


Figure 6.7: Selecting motion parameters (maximum velocity, maximum acceleration) using motor models, kinematic models, and a force margin. At left, a selection that maximizes acceleration ($2200mm/s^2$) but only allows for a maximum speed of $5mm/s$ and at right, we make the opposite trade-off and select for high speed ($88mm/s$) with only $400mm/s^2$ of acceleration.

Because our real-time controller is a software module, we can easily run path plans virtually to render the results of this optimization for machine operators before they run a job. In practice this just means that we run the machine controller in a configuration where it is disconnected from hardware, which is done using a change in the PIPES graph that runs the system. The motion controller's outputs can then be rendered in 3D along particular points in the path, or as histograms that show statistics across the entire path.

In the case of this CNC machine, I use a trapezoid-based motion controller for acceleration planning (see Section 4.2.4). It's outputs include the time-series basis spline that is normally sent to motors (and it's velocity, acceleration, and jerk states), as well as an output of the current segment's *target* velocity (as specified in the path plan), as well as the *type* of motion segment (a G0 "Rapid" (traverse) move, or a G1 "cutting" move).

To render these outputs, I use the same 3D Plotting software as in Section 5.11.1. I measure deviation of the trajectory's actual feed rate from the target feed rates, normalizing it into the range $0.0 - 1.0$ where 0.0 indicates no deviation from the target feed rate, and 1.0 indicates a complete stop.

$$V_{dev} = (V_{targ} - V_{norm})/V_{targ}$$

3D Plots allow careful inspection of toolpath components. For a broader look at the toolpath's overall deviation from target rates, we can also produce histograms of this data.

I use this tool as part of a comparison between the pocket and adaptive toolpath in 6.4.4.

6.3.2 Cutting Force Estimates

It is possible also to make cutting force estimates using a combination of real-time data from the machine and motion controller and our motor and kinematic models.

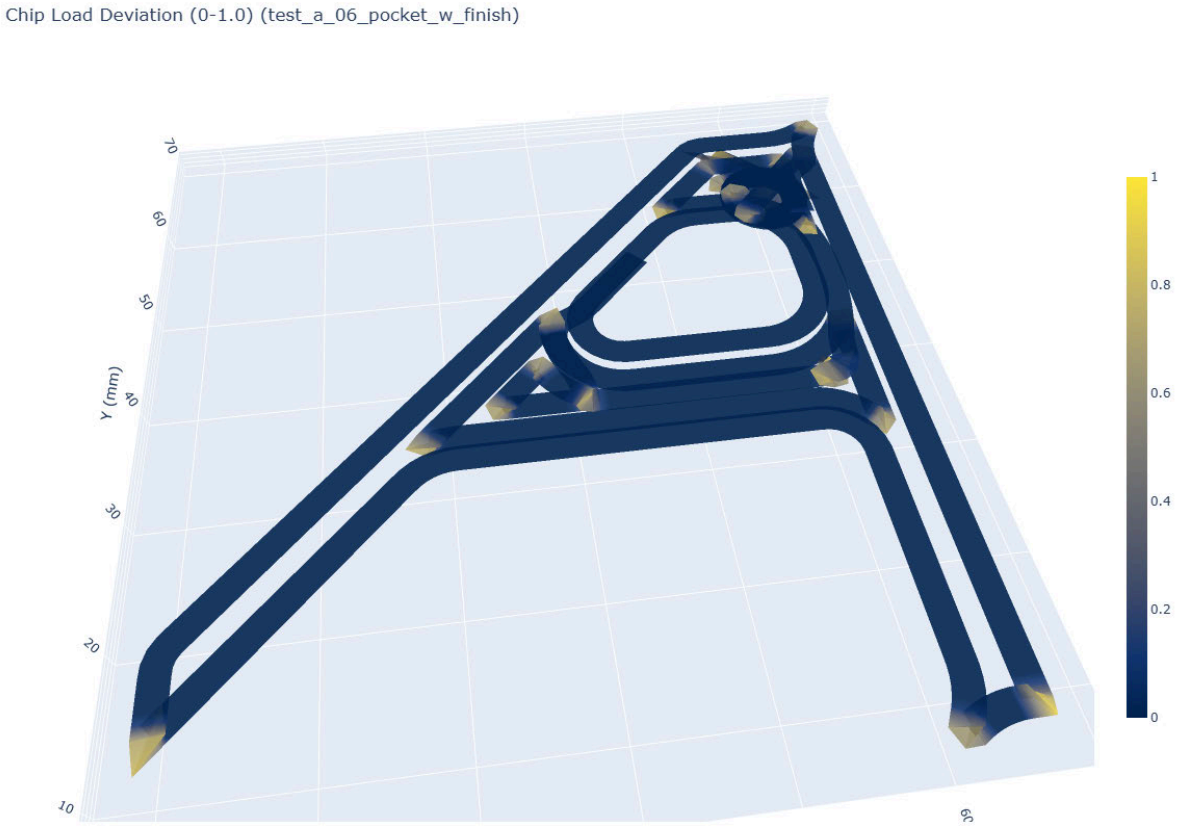


Figure 6.8: 3D output of the chip load deviation analysis tool, in a selected region of the pocket toolpath. This highlights segments of the trajectory where deviations are large.

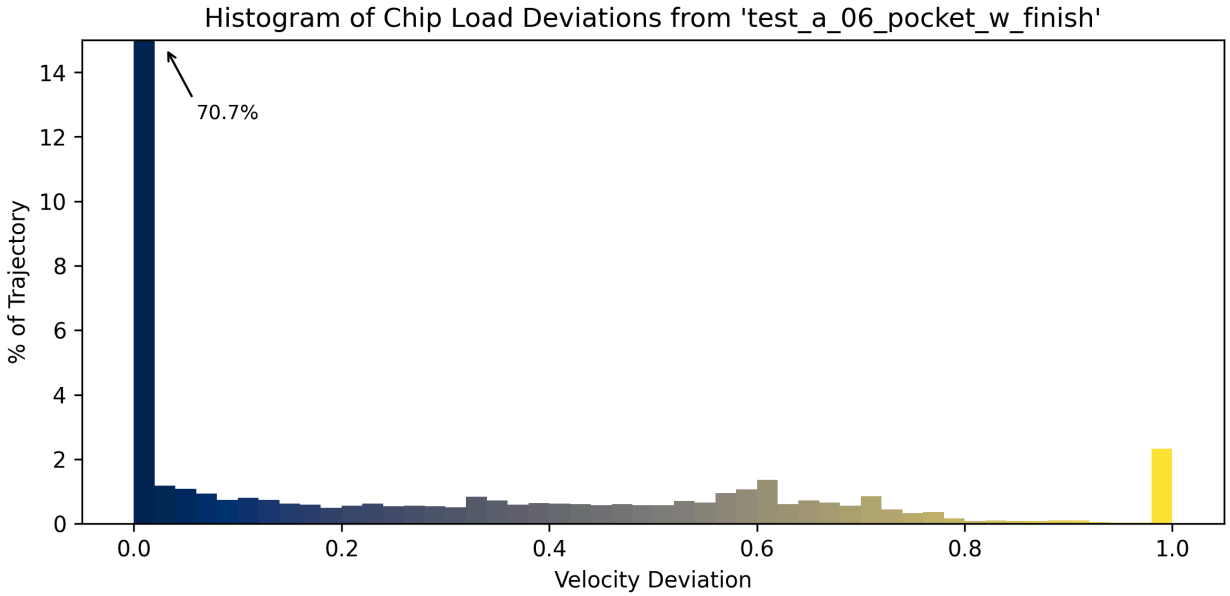


Figure 6.9: An histogram across chip load deviation in the entirety of the pocket toolpath.

While the machine operates, we collect time-series data from the motors on their positions, velocities, accelerations, and their measurements for motor current. A complete table of data collected from motors during operation is available in Table 4.3. With the motor model (Section 4.3.1), we know that current is directly related to motor torque. Our machine's drive ratios relate these torques to linear forces per axes, meaning that we can make a time-series of force vectors that were exerted into the machine's axes during operation.

Much of that force is used only to accelerate and move the machine around. However, our kinematic models can estimate those currents for any velocity and acceleration. That means that we can develop a time series that combines our total force (a motor-model based measurement) with *motive* force (a kinematic-model based estimate) to calculate residual forces.

$$F_{meas} = I_m \cdot k_t$$
$$F_{est} = F_f(vel, accel)$$
$$F_{cut} = F_{meas} - F_{est}$$

In this step, I additionally filter both force estimates before combining them using a second order butterworth filter [343] with $\omega = 20.0Hz$ passed through scipy's `filtfilt` function (to eliminate changes in the measurement's phase).

I use this tool as part of a comparison between the pocket and adaptive toolpath in 6.4.4.

6.4 Results and Discussion

6.4.1 Hardware Developed and Tested

I used the tools and methods described in this chapter to outfit a small milling machine. The machine was assembled from a Rat Rig Mill kit [344] and a GPenny Spindle which is rated to 1.5kW (about two horsepower) and runs up to 24000 RPM [345]. The spindle has an ER16 size collet and is 65mm in diameter, and is controlled by a generic variable-frequency drive powered on single phase 110V.

I added a Saunders Machine Works fixture plate and vise [346], and outfit the machine with my own motor controllers (from Section 4.4), picking motors using the process described below in 6.4.2. The motion system runs on a 48V switched-mode power supply, to which I added 10000uF of capacitance and two 47 Ohm bleed resistors (in series) to help linearize it.

6.4.2 Model-Based Selection of Motors and Motion Parameters for CNC

In this subsection, I explain how I used the tools developed in 6.2.1 (alongside heuristics!) to select motors for use in the CNC Mill above. I was able to assess the hypothetical performance of each

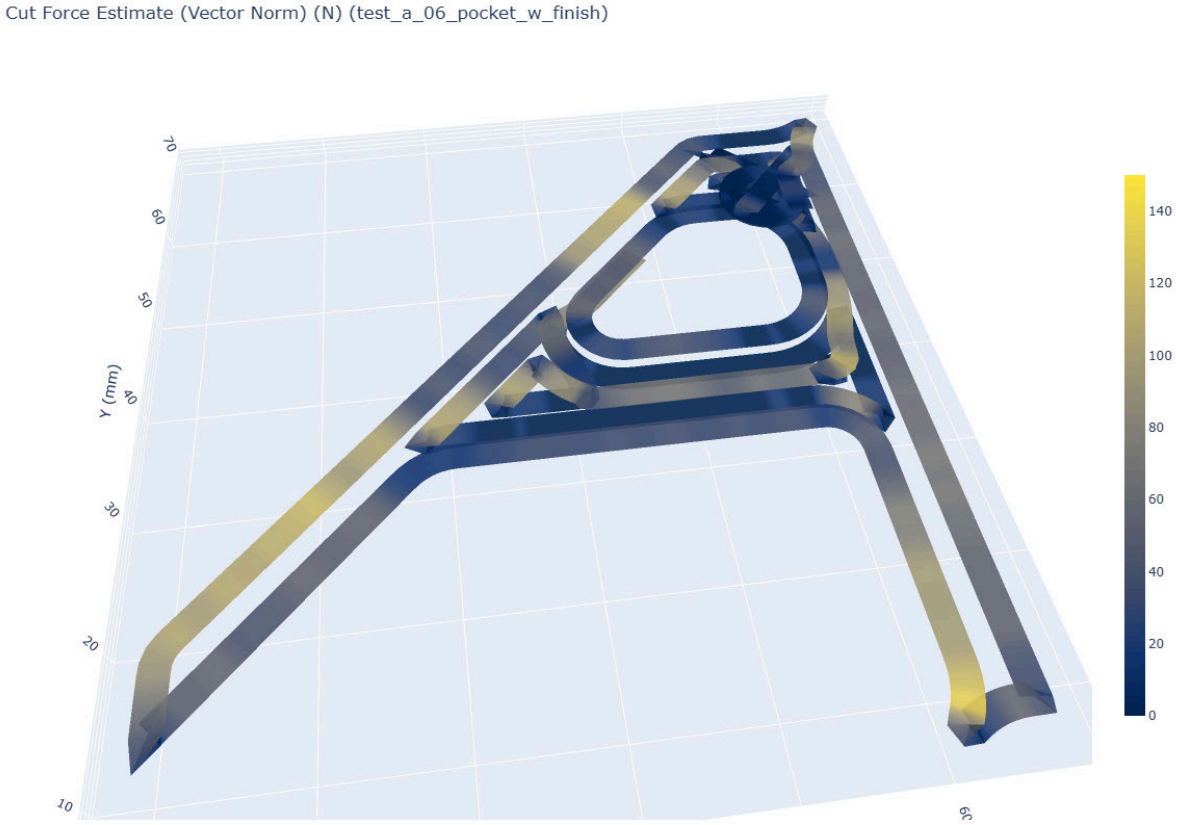


Figure 6.10: 3D output of cutting force estimates, mapped onto the trajectory. This sample is from one stepdown pass in the pocket toolpath.

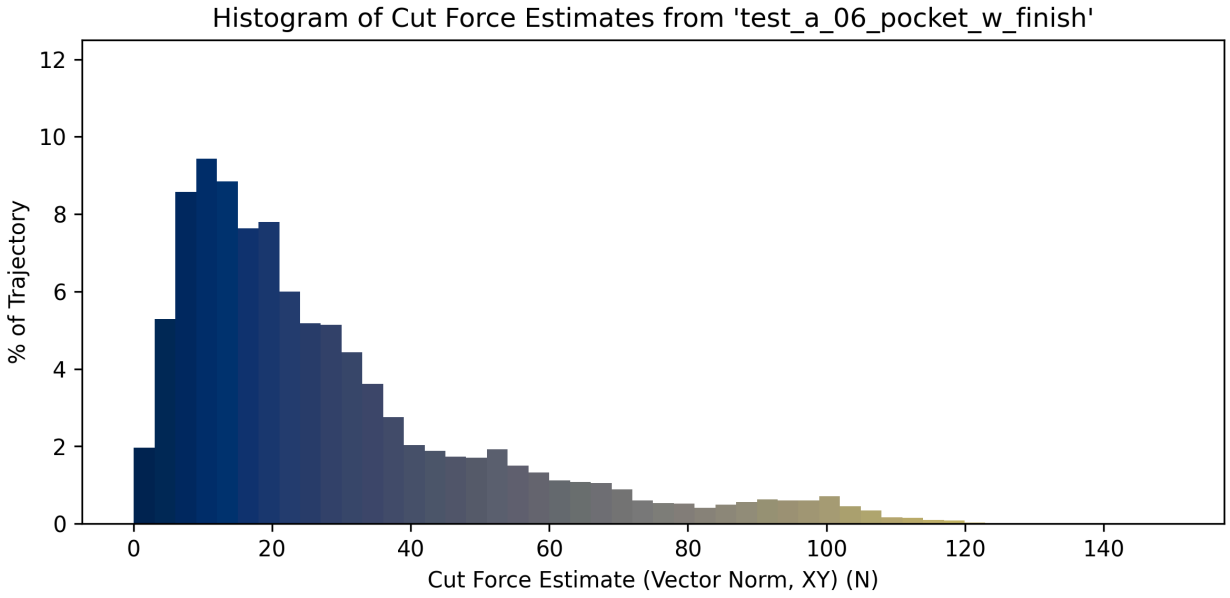


Figure 6.11: Histogram of cutting forces sampled across the entire pocket toolpath.

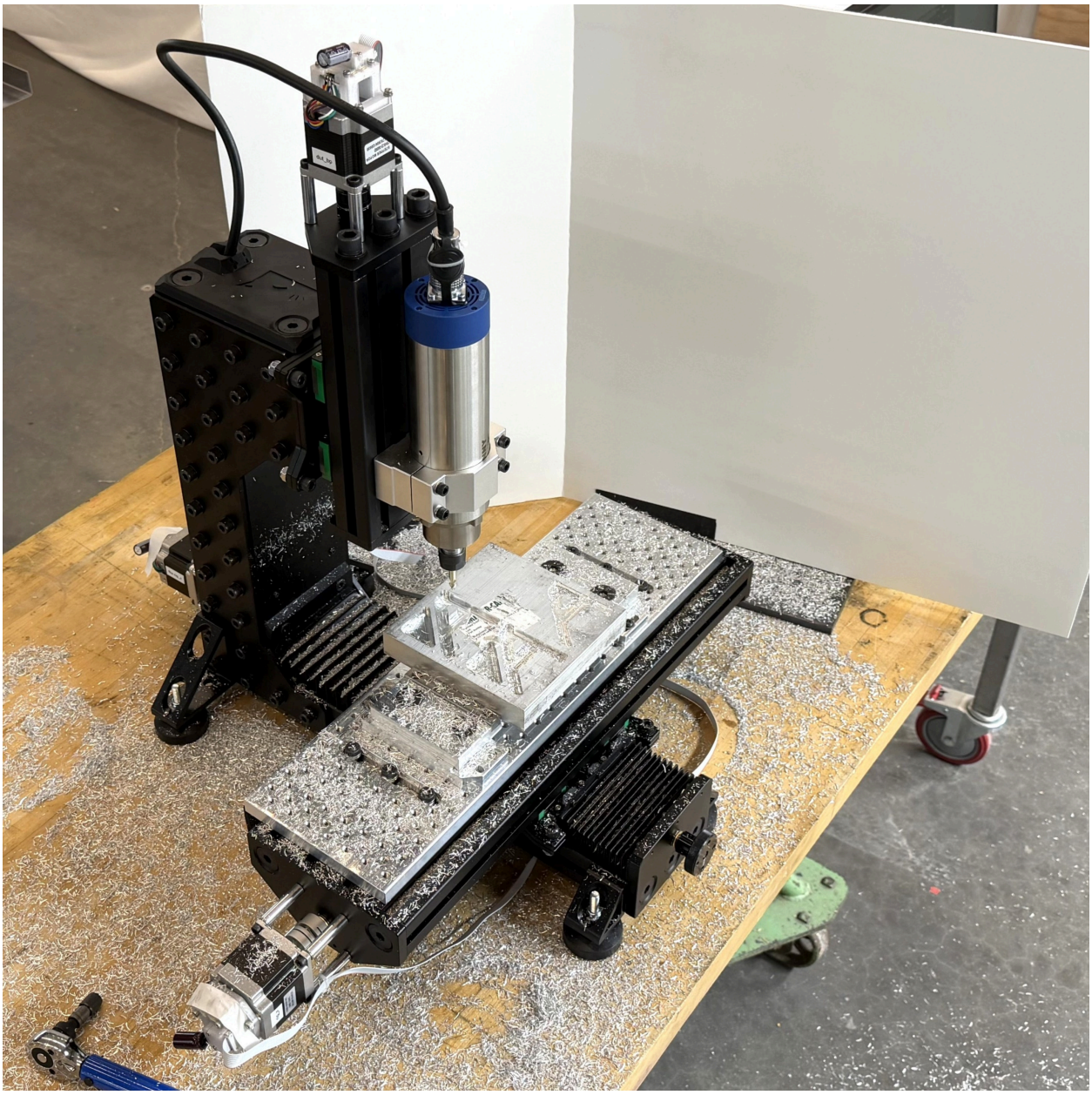


Figure 6.12: Pictured: the Rat Rig Mill used in this work, outfit with a Saunder's Machine Works fixture plate and vise, stock aluminum, GPenny Spindle (1.5kW ER16, 24000RPM, 65mm Diameter), and fit with NEMA23 stepper motors, and controllers as developed in this thesis.

axis using each motor (a total of twelve combinations) to select the motors that would provide the desired force bandwidth in the range of speeds that I expected to drive the machine.

I selected four candidate motors from our stock bins at the CBA. Each is a NEMA23 size (a faceplate dimension standard, and a requirement of our machine's frame), but have varying electrical properties and lengths. I used the method from 4.3.1 to generate an encoder calibration and model for each (this takes about 10 minutes per motor). The resulting torque curves are

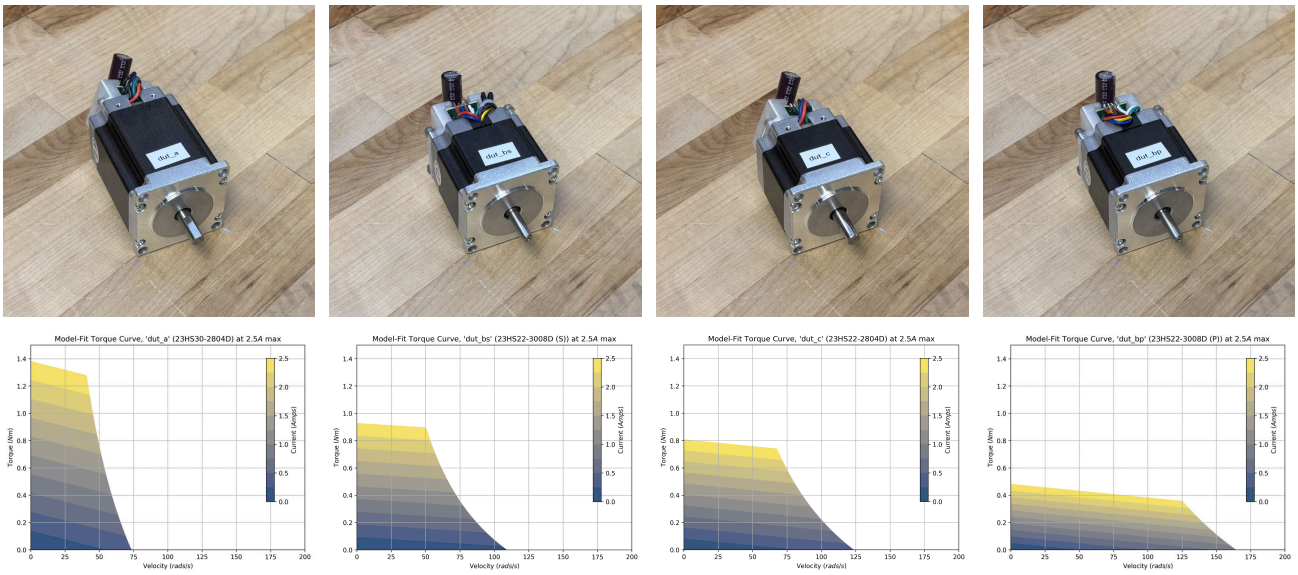


Figure 6.13: All possible motors (top) and their torque curves (bottom). Each was generated using measured parameters for the motor's resistance, inductance, and rotor inertia, as well as model-fit terms for the motors' k_t, k_e and rotor damping terms k_d, using the method described in Section 4.3.1.

plotted below, and parameters are tabulated in 6.1, where I include values also for the motor that Rat Rig specifies for the CNC kit.

To generate kinematic models, I used the methods described in 4.3.2. In brief, this involved mounting one motor on each axis (I made a preliminary selection of *dut_c* for this, as it sits in the middle of the range of available performance), and having the motor follow a chirp signal using a set of preliminary PID gains. Parameters from those model fits are below, showing the machine's relatively large damping terms ($k_{d,offset}, k_d$) that are a result of it's leadscrew transmissions.

I then combined motor models and kinematic models to produce a force margin plot for the expected performance of each. For each axis, I generated four hypothetical combinations (one with each motor). I show each of those combinations in 6.14.

Name	dut_a	dut_bs	dut_c	dut_bp	MFG
Size	NEMA23x76	NEMA23x56	NEMA23x56	NEMA23x56	NEMA23x86
τ_0	1.85	1.26	1.20	1.26	2.45
$A_{\max}$	2.80	2.12	2.80	4.24	3.00
$R(ohms)$	1.13	1.50	0.95	0.38	1.2
$L(mH)$	4.80	4.40	3.00	1.10	4.0
$J_{rot}(g \cdot cm^2)$	480	300	300	300	670
k_t	0.553	0.371	0.323	0.193	n/a
k_e	0.572	0.399	0.331	0.245	n/a
k_d	0.0025	0.00064	0.00093	0.0010	n/a
$v_{\max}(rads/s)$	75	104	125	165	n/a

Table 6.1: Motors for comparison, and their parameters.

Axis	X	Y	Z
Drive Ratio (mm/rad)	1.27324	1.27324	0.63662
Moving Mass (kg)	9.932	10.818	4.392
J motor ($kg \cdot m^2$)	0.00003	0.00003	0.00003
J coupler ($kg \cdot m^2$)	0.000002873	0.000002873	0.000003180
J leadscrew ($kg \cdot m^2$)	0.000001419	0.000001469	0.000000706
$k_{d,offset}(N)$	58.95	60.27	202.6
$k_d(N/(m/s))$	375.0	439.6	1602.4

Table 6.2: Kinematic Parameters from our milling machines' axes.

Using these models, we can see that the Z axis' increased drive ratio (it uses a 4mm lead screw, whereas X and Y use 8mm lead screws) leads to — unsurprisingly — smaller achievable feed rates overall. It also has punishing damping terms. Heuristically, we know that maintaining high feed rates is important for machining with a low-rigidity machine, to keep overall removal rates up without exerting large forces. We can turn this heuristic into a force margin, supposing we only plan to exert 200N (about 20kgf) on the machine during operation in X and Y, and half of that in Z where cutting forces are lower (we tend to *plunge* more gently). Then we can select towards higher feed rates, so long as we maintain high acceleration. Since machining involves many retracts in the Z axis, we favor acceleration over maximum speed on this axis.

In the table above, I show the final motion parameters and force margins selected using this process. I also include parameters from the CNC machine kit manufacturer's recommended setup.

In summary, this process is preferable over the state-of-the-art because it enables us to rapidly compare more possible design outcomes using real data from *our* hardware, rather than estimates. The simple metric here is that we were able to use four motor models and three axis models (each of which is fit in tens of minutes) to evaluate twelve combinations of each. Done manually (swapping each motor onto each axis) may take a day or longer. In addition, the models we use allow us to simulate operation across a wider range of parameters than those where we test them, and our controller allows us to virtually test the entire machine in a number of configurations.

This process also led to the selection of dramatically improved motion parameters over the manufacturer's recommended settings: 2x (xy axes) and 3x (z axis) increase in top speeds and 8x (xy axes) and 16x (z axis) increases in acceleration. The manufacturer selected motors have much higher rated holding torque over my selections, but a torque curve for their motor seems to be unavailable, meaning that they may have selected motors based on holding torque alone without analyzing performance at high speeds. Their selections surely also encode many of *their* heuristics, so we may simply differ in that regard. They also deploy open-loop steppers, where loss of a single step can be extremely troublesome, whereas our closed-loop motors allow us to push closer to the machine's limits without worrying about occasional hiccups. That is to say, their margins to machine performance are probably much larger.

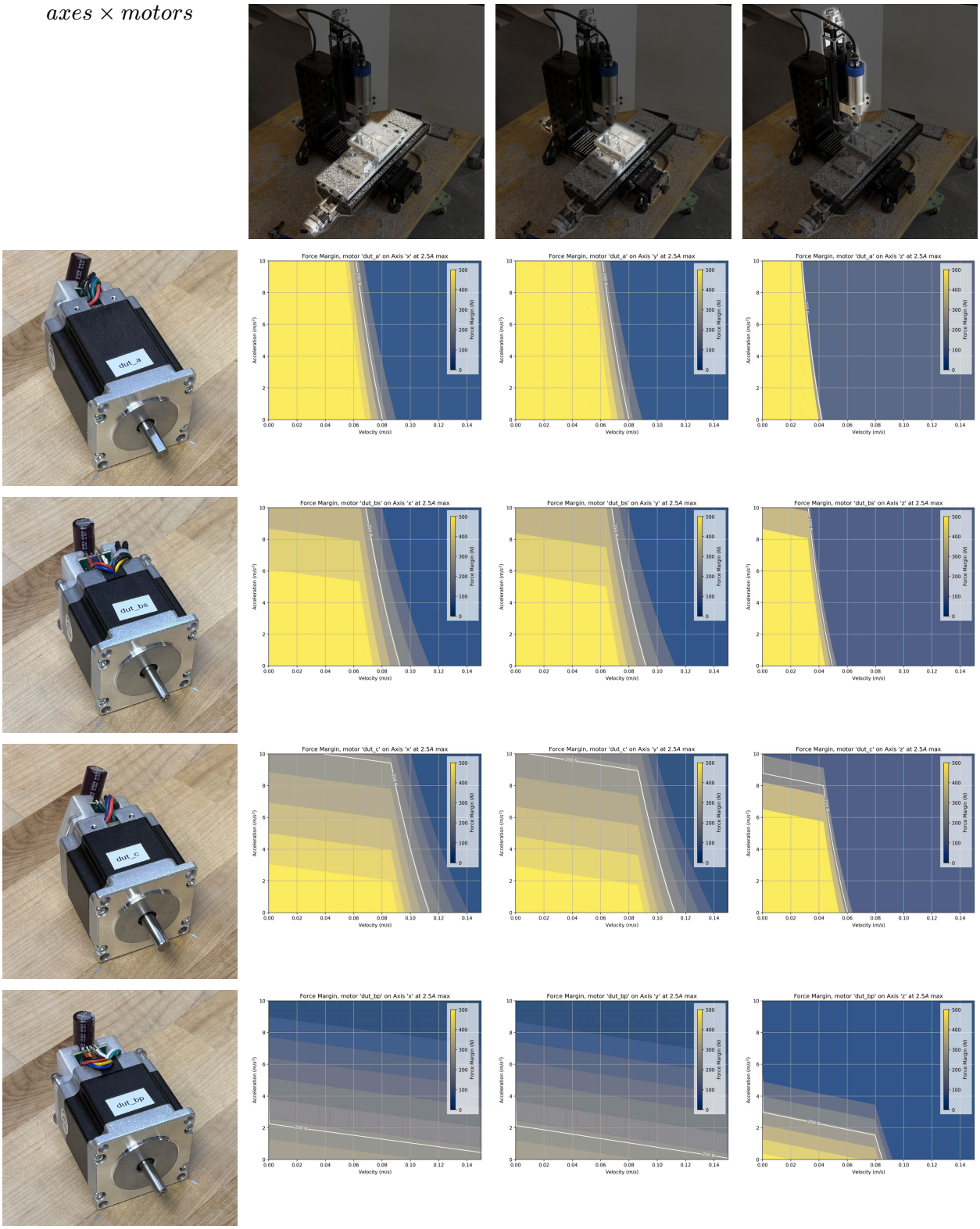


Figure 6.14: Motors (left column) and machine axes (top row) can be combined virtually to produce force margin plots of each combination's expected performance across a range of velocities and accelerations.

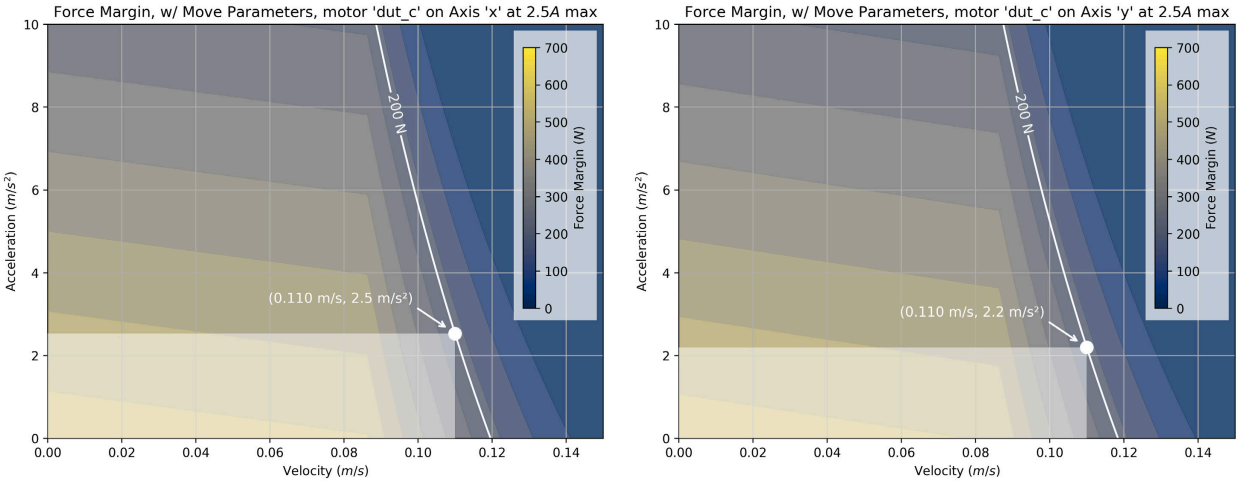


Figure 6.15: Selection for maximum velocity and acceleration parameters using force margin. Left: x-axis, right: y-axis.

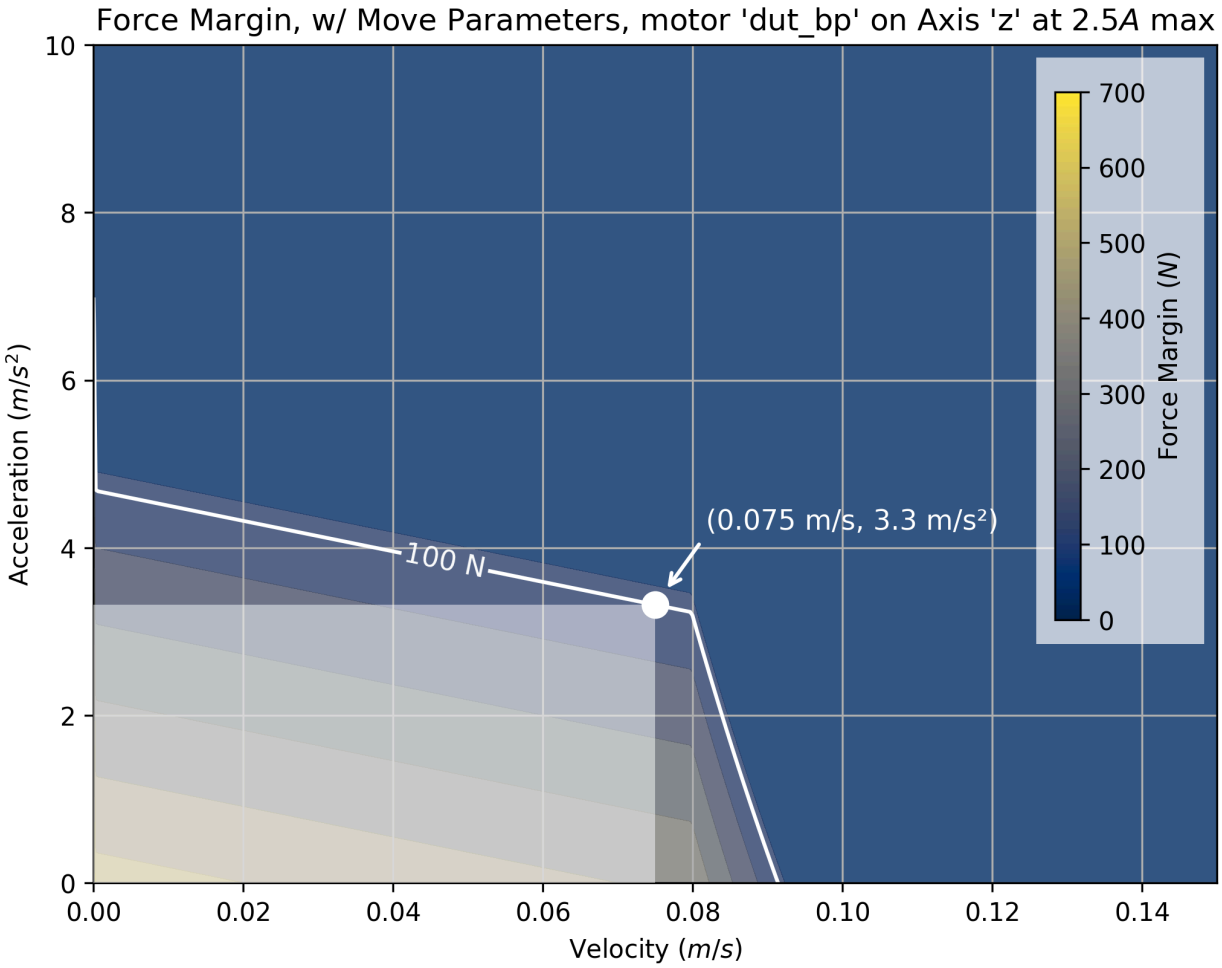


Figure 6.16: Selection for maximum velocity and acceleration, for the z-axis, using force margin.

Axis	X	Y	Z
Motor Selection (ours)	dut_c	dut_c	dut_bp
Motor Selection (model)	23HS22-2804D	23HS22-2804D	23HS22-3008DP
Motor Selection (mfg)	LDO-57STH86	LDO-57STH86	LDO-57STH86
Force Margin (ours) (N)	100	200	200
Force Margin (mfg) (N)	n/a	n/a	n/a
Max Velocity (ours) (mm/s)	110	110	75
Max Velocity (mfg) (mm/s)	50	50	25
Max Acceleration (ours) (mm/s^2)	2500	2200	3300
Max Acceleration (mfg) (mm/s^2)	300	300	200

Table 6.3: Selected motors, force margins, velocities, and accelerations for our machine's axes. Includes a comparison to the machine manufacturer's selections.

6.4.3 Machining with Models

I modeled a small test piece, *test_a,* with two bosses at 1/4" depth each, to mill out of aluminum. The part is small, but I tried to include some common geometric features: small islands, both rounded and sharp corners, and *dogbones,* which are relieved sharp corners of roughly the same diameter as the cutting tool (bottom left and far right of each figure below).

I generated three toolpaths for this part using Autodesk Fusion version 2701.1.18. I selected a three-flute helical endmill at 1/4" diameter, which I thought would approximately saturate the spindle power during high speed machining. The toolpaths generated (see Figure 6.17 and parameters in Table 6.4 below) include a conventional *pocket* where voids are milled using concentrically increasing paths that are offset from the pocket edge, and an *adaptive clearing* toolpath, which is a "high speed machining" strategy that clears pockets using a series of smaller arcs. The third toolpath is another pocket, set up as a finishing pass: it clears a small amount of material that is intentionally left over during higher speed, higher load operations (to produce a smoother finish).

All three toolpaths are configured to generate the same chip load, using the spindle's maximum RPM and relatively high linear feed rates. The chip load value was selected heuristically, using mine and our shop director Dan Gilbert's own knowledge in machining. However, the machining strategies have different cutter engagement areas (cut depth x cut width). The adaptive path strategy is developed to maintain a uniform engagement area, avoiding slotting (where the tool cuts across its full width) by generating a series of small arcs to traverse through narrow sections. The pocket toolpath does not do this, and so its engagement area is variable - again, see Table 6.4 for numerical values.

6.4.3.1 Discussion of the Model-Based Machining Experiments

In this subsection I am going to describe the process that *I* went through — using a few of my tools and a heavy helping of heuristics — to develop the parameters in Table 6.4 above. While it does not constitute a rigorous evaluation of the process, I think that it is useful to include to describe the types of workflows that modelling can aid. For reference, I am familiar with machining as a

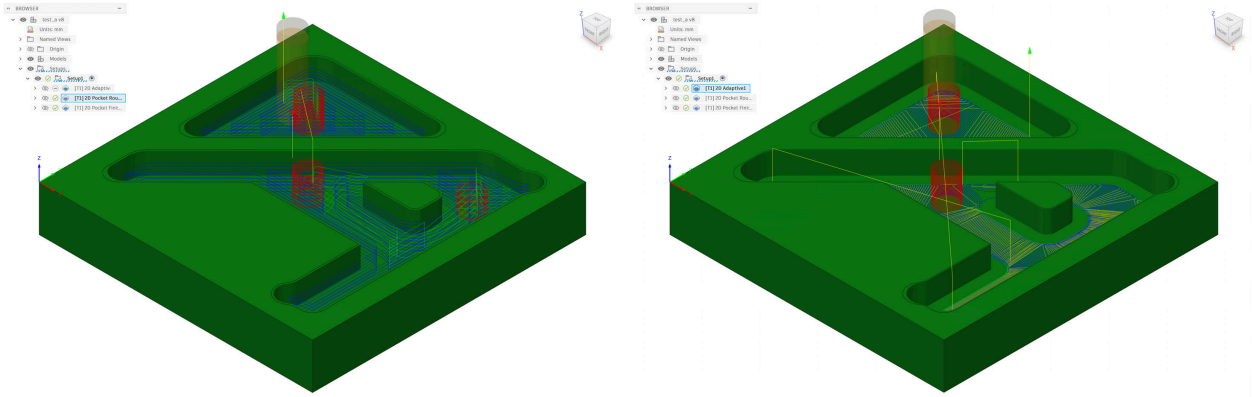


Figure 6.17: The two toolpaths that I tested, a 2D Pocket (above) and 2D Adaptive Clearing path (below). Both were generated in Autodesk Fusion 360, with parameters listed below in Table 6.4.

process but have not accumulated more than ten hours of total time programming CNC machines. Of course, I developed an understanding for their physics during the course of this project, but my *tacit* knowledge in the domain is limited when compared to a "seasoned" machinist.

Table 6.4: Cutting parameters for the machining experiments.

Final Parameters Used	Value	Meaning
All Toolpaths		
Spindle RPM	$24000 RPM$	Spindle's rotational speed.
Surface Speed	$478 m/min$	Speed at cutting edge.
Cutting Feed rate	$78 mm/s$	Translational rate.
Chip Load	$0.065 mm$	Linear translation per endmill tooth appearance.
Ramp Angle	$0.6°$	Helical angle for plunges.
Ramp Diameter	$6 mm$	Helical diameter for plunges.
Pocket Toolpath		
Stepover	$2.5 mm$	Depth of cut.
Stepdown	$0.75 mm$	Width of cut.
Cut Engagement Area	$1.875 - 4.763 mm^2$	Area of cut $(w \cdot d)$
Adaptive Toolpath		
Stepover	$0.5 mm$	
Stepdown	$6.35 mm$	
Cut Engagement Area	$3.175 mm^2$	
Finishing Pass		
Stepover	$0.4 mm$	
Stepdown	$6.35 mm$	
Cut Engagement Area	$2.54 mm^2$	

I developed these parameters primarily guided by chip load and cutter engagement heuristics. I tuned parameters first for the adaptive path, using the machine simulation (in particular, histograms of chip load deviation were most useful (6.9)) to select a feed rate where the motion systems's acceleration limits (selected in 6.4.2) would not cause egregious deviations from the target chip load during cornering.

I ran this machining job a total of six times, making small adjustments to feed rates and cutter engagement areas as I proceeded. I started with the adaptive toolpath and initially selected a smaller spindle speed and only half of the final feed rate and cut depth in Table 6.4, to maintain the target chipload but minimize cutting forces on the machine as I had no sense of what those forces may be. The very first run stalled the spindle during the initial ramp in (the red helical moves in Figure 6.17), after which I halved the ramp angle. The second run (an adaptive path) was successful.

I then used the cut force estimation plots (Figure 6.10) to assess my use of the machine's available force under real cutting conditions. Seeing that my initial parameters used only tens of newtons of (estimated) cut force during the heaviest sections of the operation, I doubled the feed rates, step-down, and spindle RPM. This maintains the target chip load, but increases the total cutting force. The limit at this point was the spindle RPM.

Having access to a thermal camera was an additional aid; I used this to assess whether the endmill was generating too much heat, i.e. melting metal rather than cutting it. An heuristic here is that we want to see the heat energy going into the *chip* rather than the *part.*

During the third test I had parameters set to the rates in the table above, *except* for the ramping feed rate, which I had lowered out of fear. This turned out to be the wrong intuition: the ramping rate, in combination with the higher spindle RPM, decreased the chip load such that the ramp failed again, melting the material rather than cutting it. The sacrificial endmill for this run is at right in Figure 6.2. I picked up on this for two reasons: (1) the endmill had obviously welded itself to the part and (2) the chips that were emerging at this point were much smaller than those that were thrown out of the part during the last successful cuts.

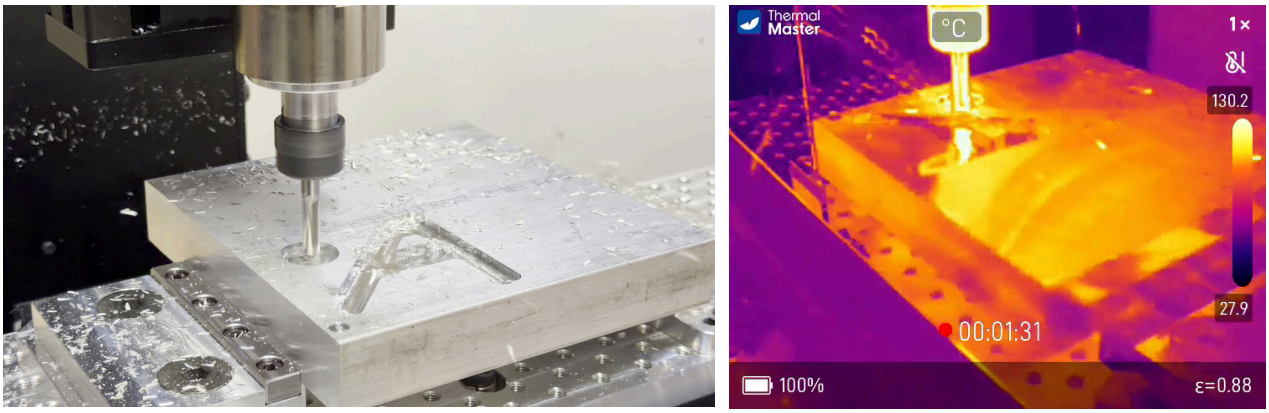


Figure 6.18: First Chips on the CNC Machine, cutting at 1/2 of the final depth, feed rate, and spindle speed. Image at right is a thermal camera frame around the same time.



Figure 6.19: Chips ejected during cutting. Looking at these provides valuable feedback; it is not uncommon for a machinist to measure chip thickness to ascertain what *realized* chip loads are during their toolpaths.

For the final test of the adaptive path, I increased the ramp feed rate to match the other paths in the toolpath. Ramping can be difficult to figure out, since cutting edges at the bottom tip of an endmill have more variable geometry than their side edges - the center of the tool also spins at "zero" RPM which presents its own set of problems that I won't discuss at length, but you can imagine how steep ramp angles may exacerbate those issues.

With that last adjustment, I had a good recipe for the adaptive path. One more note here is that I swapped the uncoated endmill for a similar tool with an *ANF* coating (Aluminum Non Ferrous). This is a proprietary PVD (Physical Vapor Deposition) coating [347] that is designed to decrease friction and increase hardness at the tool surface, which in turn is said to reduce heat flux into the tool itself. Looking at the thermal images from Figure 6.18 and 6.20 above, (130 vs. 71 degrees at the hottest point), we can see that this holds some water.

I copied those parameters into the adaptive path, but had to find new stepover and stepdown parameters. I initially estimated that I could get away with a 2mm stepdown, to finish the part in only three depth passes. This test failed during the second step-down, stalling the spindle. I

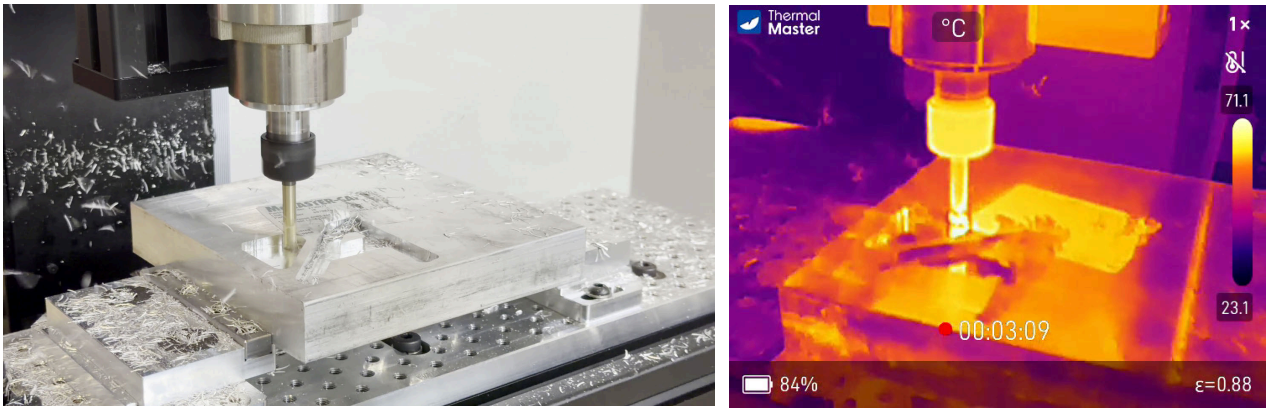


Figure 6.20: Machining the adaptive path with final parameters. Image at right is a thermal camera frame around the same time.

used the cutting force analysis tool to check the force exerted by the machine during that pass, and decreased the stepdown to 0.75mm, producing a total of eight depth passes. The second run of the pocket toolpath was successful.

In all, the tools I developed were useful to understand the machine's feed and force limits, and anticipate slowdowns due to acceleration before the parts were run. These types of feedback are not available in state-of-the-art CNC workflows where motion control and process parameter selection are separated and machinists tend to rely purely on tacit knowledge and intuitions e.g. about how a machine should *sound* when it is operating. Here, we can add quantitative data for these analysis where it was previously unavailable.

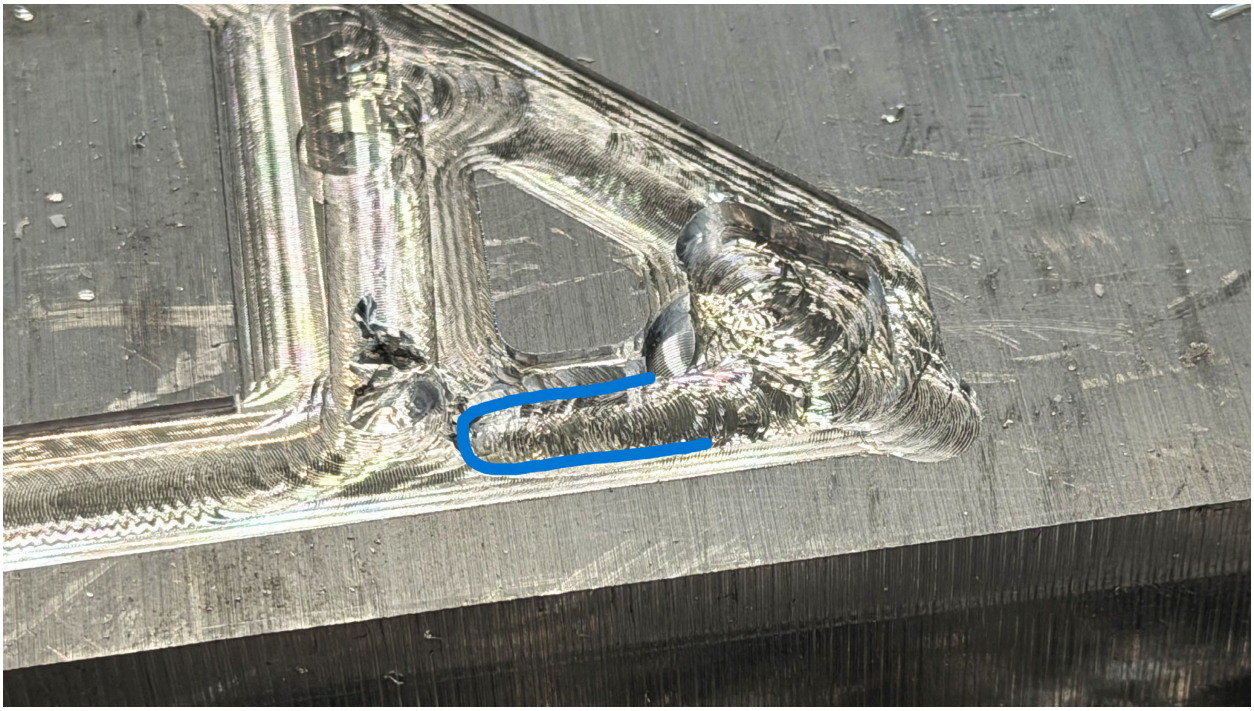


Figure 6.21: The site of failure in the first pocket toolpath.

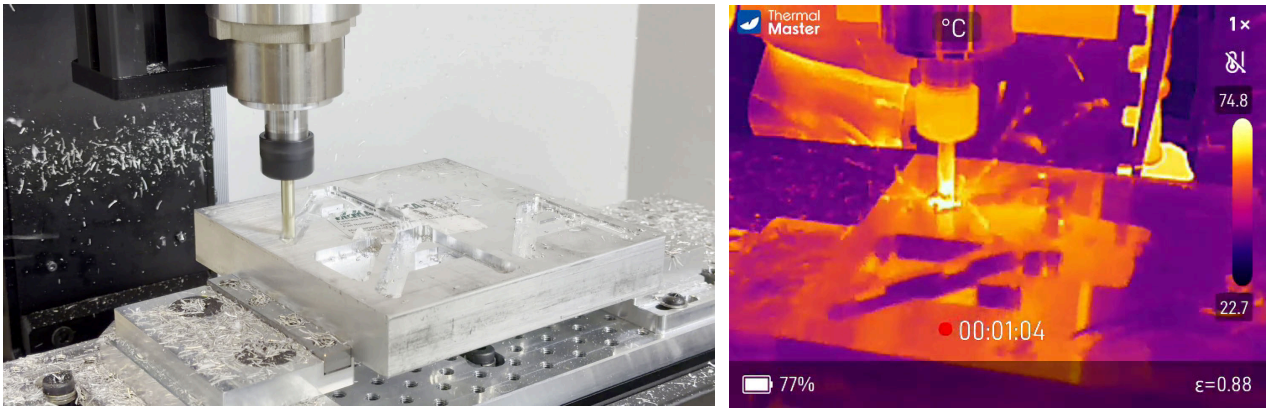


Figure 6.22: Machining the pocket path with final parameters. Image at right is a thermal camera frame around the same time.

However, I had to rely heavily on heuristics to successfully machine parts. Clearly missing from this work is a cutting force model, a spindle power model, and properly articulated heuristics or calculation of chip loads (two failures came from spindle stalls, and one from chip melting). I discuss this, and the possibility of producing real-time user (and controller) process feedback in Section 6.5.

6.4.4 Chip Load Deviation and Cutting Force Estimates

"High Speed Machining" strategies like the adaptive toolpath tested here are a (relatively) new phenomenon in the CNC industry. They became possible when machine controllers became fast enough to handle the very-many small arcs and segments that they are made up of. Their operation is slightly counter-intuitive; in an adaptive path our machines spend about half of their time *not* cutting (circling around to the next cut), whereas traditional toolpaths engage the tool more frequently.

We can use our analysis tools to look at the differences in real machine operation across these two paths, reason about interactions between the motion controller and toolpath, and discuss limits to the current set of tools I have developed.

The first thing to notice is in the chip load deviation comparison above. We can see in the histograms that the pocket toolpath (left) actually deviates *less* overall from targets than the adaptive path (right). There is a clear concentration of deviation in the adaptive path around 0.4, my estimate is that this corresponds to the cornering velocity through the chosen *radius* of the path's lead-in and lead-out segments (where it *turns around* to start a new cut). We can see that in the 3D plot as well. These radii are also tuneable parameters in our CAM tools, and so it may be possible to use these virtual motion controllers to more intelligently pick those. In fact, ascertaining the maximum speed for a kinematic system through an arc is quite simple (so long as axes accelerations are uniform or close to it) ($V_{max,corner} = \sqrt{A_{\max} r}$). In my tests, I used default radii from Autodesk Fusion, which are likely tuned for *industrial* grade machines, which obviously do not match our small CNC mill in terms of motion performance.

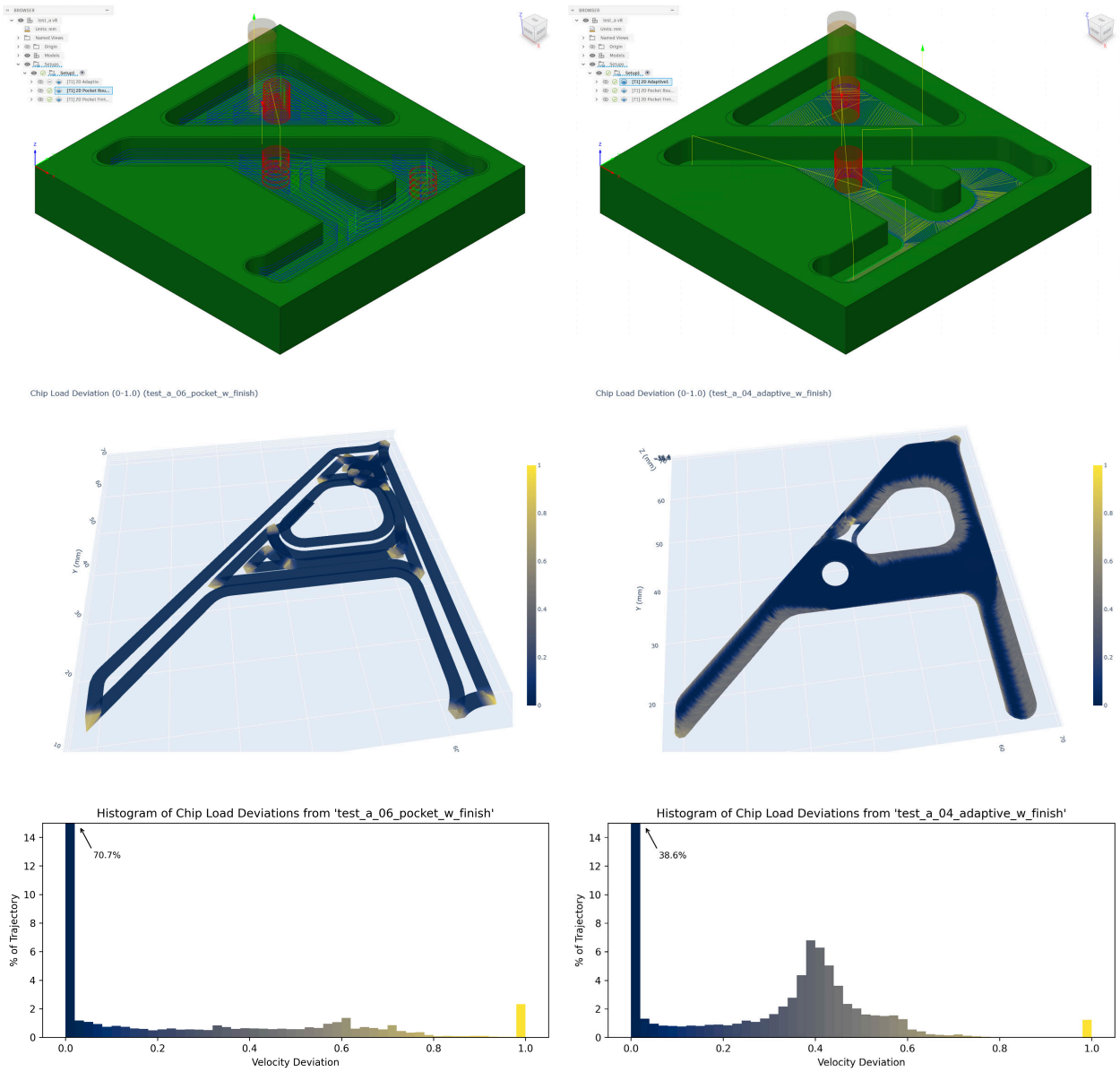


Figure 6.23: This plot compares chip load deviation (6.3.1) for the pocket (left) and adaptive (right) toolpaths, including a sample of the 3D trajectory with deviations highlighted and histograms from the entirety of each toolpath (not including rapid / traversing motion). All plots are normalized to the same range $(0-1)$ where 1 indicates a full stop (total deviation) and 0 indicates that the machine is moving at the target feed rate, matching the programmed chip load.

Another note here is that this cornering is done *partially* during the cut and *partially* during traverse into the next cut. CAM tools do not calculate cutter engagement area, so it is difficult from those alone to ascertain where in these distributions the velocity deviation may cause issues (if the deviations appear only when we are linking two cuts, they matter only in terms of total part time). In our case, we can combine our cut force estimates with chip load deviations to plot deviations *only* where our cut force is larger than some value: below I chose $25N$ as a dead band.

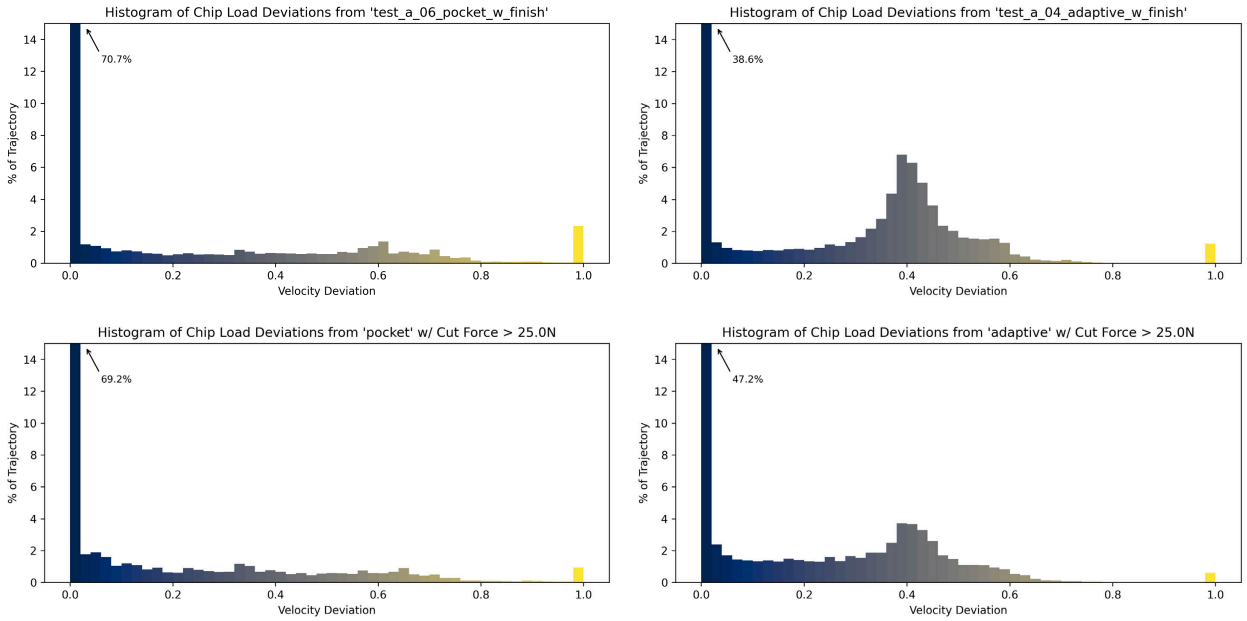


Figure 6.24: Distributions of chip load deviations from all moves *labelled* as cutting moves in CAM outputs (above) vs. from all moves *with cut force estimates* $> 25.0N$.

The differences in the distributions above is subtle, but shows us that about half of the adaptive path's worst deviations (around 0.4) take place in areas of low engagement / low cut force. Finally I will note that the *worst* deviations in the adaptive path (where there are many corners, but most with some radius) are smaller than the *worst* in the pocket toolpath (where there are less corners overall, but those corners have sharper angles).

The pocket toolpath was faster overall; a total of $153s$ of operation. The adaptive path took $216s$.

Comparing cut force estimates in both toolpaths provides some more insight as to the strengths and weaknesses of either strategy. In the histograms above, we can see that the adaptive path uses overall lower cut force, and has a much smaller maximum force (around $60N$) than the pocket ($\sim 115N$). The 3D plot shows us where these hot-spots are: in the regions where the pocket toolpath must *slot,* i.e. cut with the full width of the tool. This is the primary weakness of traditional toolpaths: slotting is inevitable in parts with narrow channels to cut, which leads to high variability in cutting loads during machining. Slots also present chip evacuation issues, as the cutter has limited space around it to throw chips.

6.4.5 Summarizing Discussion

In this chapter, I showed that basic motor and kinematics models can be used to:

1. Help us to design machines (6.4.2), in virtually combining motor and axes faster than would have been possible were we to physically couple and measure each, and in using those models to intelligently select optimal motion control parameters with guarantees (rather than guesses) about those parameters' feasibility.

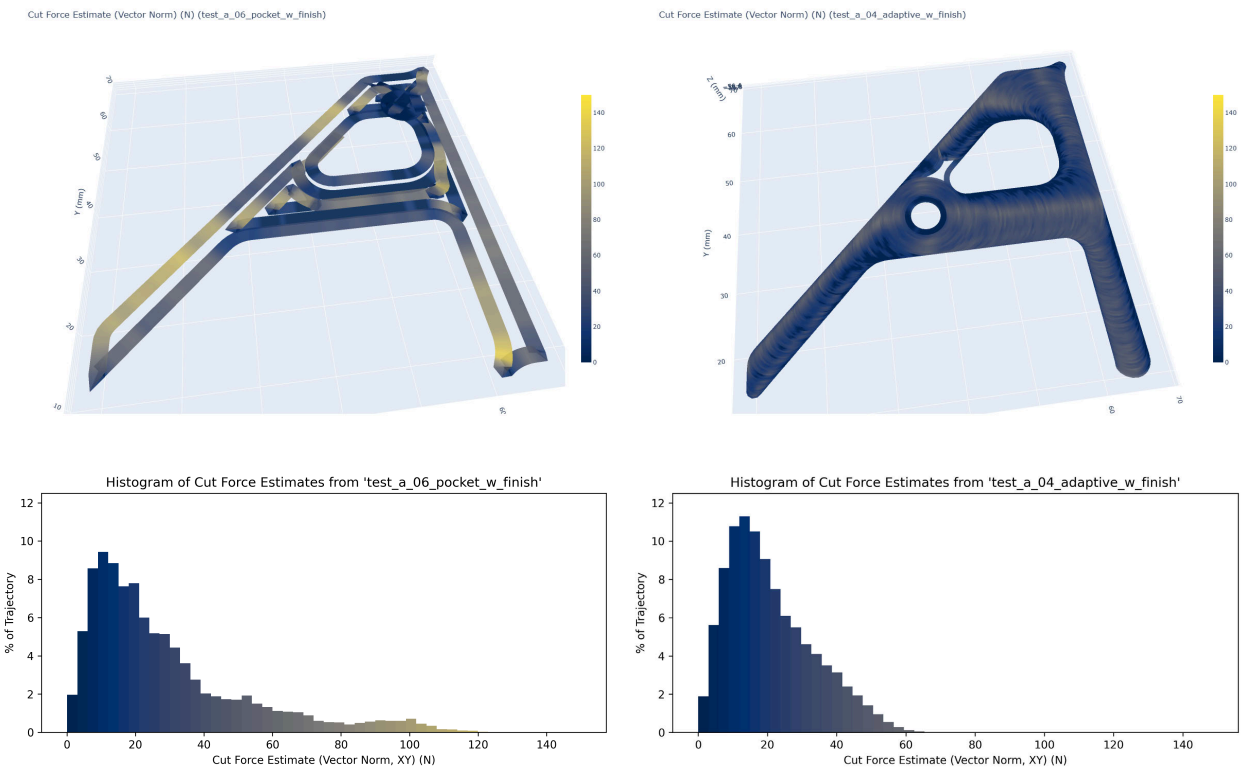


Figure 6.25: This plot compare cutting force estimates (6.3.2) for the pocket (left) and adaptive (right) toolpaths, including a sample of the 3D trajectory with forces highlighted and histograms from the entirety of each toolpath (not including rapid / traversing motion). All plots are normalized to the same range $(0 - 150N)$.

2. Help us to predict what will *really* happen when we run our machines (6.3.1), in simulating *as-optimized* toolpath velocities using our software-defined machine controller.
3. Generate new data (cutting force estimates, 6.3.2) that can be used to understand what is happening *as* we run our machines.

Each of these contributions still relies heavily on tacit knowledge in machine design and in machine operation. The design workflow required heuristic estimates for *requirements,* i.e. in estimating *how much force bandwidth is enough* and *what types of feed rate will be desired during operation.* The machining workflow incorporated tacit knowledge for chip load (and it's relationship to feed rates and spindle speed), an understanding of how cutter engagement area relates to cut force, and an understanding of spindle limits.

As in the FrankenPrusa (5.4.2), our workflow doesn't add any new instruments to the machine (besides motors, which are of course required in any configuration), it only modifies control software. This makes progress towards our mentioned goal of retrofitting existing hardware with smarter controllers.

6.5 Future Work

This work covers only the very basics of what model-based CNC might entail. We want to go towards a more complete workflow, where models enable machinists to make better parts, more rapidly, and where models can help to train new operators.

I think the framework for doing this is clear; we need *computational models* of our machines, i.e. software objects that can be used to simulate our hardware *and* its control. These can be inspected at a high level and inform heuristic parameter selection, and (as they are developed) be used to directly optimize pieces of the process. By developing models *on the hardware where they are used,* we avoid the common issue where models diverge from reality, becoming more troublesome than they are useful.

The CNC industry is old and resistant to sudden change; we cannot realistically propose that we should retrofit everyone's hardware to accomplish these goals. However, there may be minimal changes available, and much of the modelling work can be useful even at coarse precision within existing workflows.

Other researchers that I have mentioned here and in other background sections are also making progress in this regard, but as I have noted the architectures that they use to assemble their controllers are limited in important ways. In particular, each of them wraps their new methods around existing GCode interpreters. I am proposing that once we can remove that bottleneck we will be enabled to approach the problem more directly. From where we are at the moment, I would proceed in a few steps.

6.5.1 More Instrumentation and Faster Networks

I have shown in Chapter 2 how the control framework I develop in this thesis allows us to attach new sensors to hardware and easily correlate those readings to machine control states. In Chapter 5 I showed how these readings can be used to generate models for machine operation, and other researchers have added similar instruments to CNC machines to perform similar works in this context. [348] is an excellent review of the emerging sensors used in CNC machining research. The authors note that one of the pressing research gaps lies in learning how those sensor readings can be integrated with machine controllers and with CAM tools, and how data can be synchronously collected across workflows.

I think that the tools that I have developed here show promise in that regard, but there are clear bandwidth requirements described in that paper that lie beyond what I have shown here. To that end, improving overall performance of the tools that I have developed will be a critical step if we are to extend them towards an end-to-end autonomous machining workflow.

This may only require that we integrate faster link layers into OSAP; at the moment I mostly use 3000 Kbps network segments and USB-CDC devices. Integrating Ethernet-based network segments or even simpler custom embedded links based on SPI (where e.g. 50000 Kbps is not difficult to

achieve using modern microcontrollers) may be the method of choice. This will also require faster microcontrollers than the ones I primarily use, but those too are becoming ubiquitous.

Roughly in order of their value to the modelling efforts, the sensors that I think are most valuable to add on top of smart motor drives are:

1. Spindle power readings, either via integration with existing spindle controllers or using out-of-band current measurements via hall-effect current clamps.
2. Accelerometers and microphones for high-frequency data.
3. Load cells to measure forces directly.
4. Small thermal cameras and sensors.

6.5.2 Predicting Cutting Forces

Cutting models themselves are already very well understood, the challenge in our framework is to fit them with instrumentation available on our machines.

It's clear that *spindle* load is a primary limit in many machining systems, stalling spindles is more common than stalling a machine's kinematics. In the hardware that I developed here I didn't have time to instrument the spindle, but doing this would have the clear benefit *first* of understanding its power constraints, but also as a tool to develop cutting force models: the torque required to shear the material apart, at a known tool tip diameter, is probably a key instrument in learning those models. In fact, where realtime feedback *is* available to CNC operators, spindle load is typically the first to appear, meaning that these data are at least readily available within state-of-the-art machine controllers (but is presented as a simple load percentage — it is measured in amps, as in any other motor model, so the last step is fitting some k_t, which I expect is well understood by machine and spindle manufacturers).

The primary challenge that I expect in developing cutting models is measuring *cutter engagement area* — that is, how much of the tool's surface area is passing through uncut material? This is dependent on the machine's position in the part (of course!) but more importantly is dependent on the *current state* of the stock geometry, as it goes from a block to the finished part. Computational representations for constantly changing geometry are challenging; probably the simplest is a voxel space, but the memory required to store 3D voxels quickly becomes punishing as we increase resolution.

These are also highly dependent on variables that frequently change: the average CNC shop stocks hundreds of unique tools, and hundreds of thousands are available in the industry. Each has variable geometry, but also variable surface coatings, etc. Models developed for cutting forces would have to be transferable across this heterogeneity if they are to pose any practical use, *or* they need to be easily developed using test equipment available in most shops: i.e. the machine itself.

6.5.3 Intelligently Selecting Feeds, Speeds, and Engagement Areas

If we can predict cutting forces, selecting feeds (linear traversal rates), speeds (spindle RPMs), and engagement areas (step-down, step-over, etc.) is likely an easier task. At a coarse step, this is as simple as writing down ideal chip load heuristics, and then selecting cutter engagement areas such that spindle load and motion system loads do not exceed the machine's limits. There is a space of these solutions that lie along two degrees of freedom: linear feed rate corresponds directly to an optimal spindle RPM (given a chip load), and total cut area can be framed either with a set step-down (producing a step-over such that area remains equal) or a set step-over (producing a step-down…). As I showed in the printing section, we can then apply heuristics that describe how *hard* we want to work: how much proportion of the machine's underlying limits do we want to deploy in the cut?

The other constraint here (in these selection problems, constraints are helpful) has to do with *chatter,* which I discussed only briefly in this chapter but is in practice probably the predominant problem faced by machinists and CNC programmers. Chatter occurs when the cutting edge excites the machine / tool / part structural system at its resonant frequency. It is primarily a function of three things: frequency response of that system, input frequency (spindle speed × number flutes), and the cutting force (excitation amplitude). Perhaps the most challenging of all is that *part stiffness* changes during the course of the job. Modelling that in a feed-forward manner is difficult but not impossible, it just requires tight coupling between geometric representations and the controller (as already mentioned above for cutting models: voxels are heavy).

It may be more productive to rely on real-time estimates of cutter force (of the kind that I explored here), and known values for excitation and the *machine and tool's* stiffness — leaving just the part stiffness and damping as free parameters. These change not only as the part becomes smaller, but also when we excite the part from different locations. During operation of one-such chatter detecting machine, we may be able to use a *rolling estimate* of that excitation to differentiate *which direction* in the stability plot we are heading (into *more* or *less* excitation, which could tell us *which direction* we need to turn in (RPM, $v(s)$) in order to reduce chatter).

This sounds like a good start, but practitioners will note that *once you have entered chatter* you are already done for, the part may be ruined even if you correct the issue shortly thereafter. This is why predictive models are important. Either we need to model geometry so completely that feed-forward controllers can reliably avoid chatter, or our chatter detection models need to be tuned well enough that they can detect the conditions that arise *just before* chatter begins.

6.5.4 CNC Machining as an Explicit Constrained Optimization Problem

So, if we can manage cutting force models, spindle instrumentation, and chatter, we can probably pick *most* parameters for any given CNC operation, if we are provided with a tool and a machining *strategy* — i.e. one of those shown in Figure 6.3. However, when I have spoken to practical machinists about this project, they cite these early steps (tool selection, strategy selection and ordering, and *part fixturing*) as their largest burdens for overall productivity.

In this thesis I showed that it is possible to solve machine "speeds and feeds" in FDM, using an online optimization of motion and process via models. This included similar components to those that would be applicable in machining. It is my expectation that it will soon become possible to solve this problem (and the others) using direct optimization. The computational linchpin is the geometric representation of the stock being machined, and the memory required to properly represent that as it changes. Luckily, GPUs with exceptionally large memories are being developed for AI workloads, and those will be applicable here. In Section 7.7 I discuss at more length how the work in this thesis connects to the reinforcement learning methods that may be applicable for machining.

6.5.5 Generating Useful Outputs for Machinists

I want to close the section with a note on the *utility* of automated systems, and who they are made useful for. Overall I think that the goal of completely automating machining is suspect: as in other cases in automation, developers of technology can be too eager to replace humans with intelligent systems. Especially in machining, which is filled with corner cases, gotchas, and valuable tacit knowledge that is very difficult to articulate in computational models, we should not make this a primary goal.

Instead, work on modelling machining is probably most useful (and most immediately productive) in helping machinists to better develop their understanding of the process: to understand cutting physics *with* models, rather than to develop models that *replace* their intuition.

7 Discussion

In the introduction I posed a set of guiding questions about machines (1.6). The particular answers are spread across the chapters above; it is worth summarizing here how each was answered and pointing back to where each was demonstrated.

To the question of *systems architecture* for feedback-based machine control, Section 2.3 looks at how heterogeneous networking enables the *networking* of high- and low-level machine control components, and Section 2.4 shows how that can be combined with software representations of controllers which can be rapidly reconfigured. Using networks and *networked software components* rather than file formats (like GCode) enables us to build machine workflows across planning and control steps. Describing workflows that combine variable control configuration steps *alongside* machine tasking is then managed by *combining* dataflow representations with scripting capability, which allows us to easily switch between machine configurations as testing devices (to build preliminary models) and as output devices (to make parts). I showed that the systems can be re-configured (rather than re-built) to develop both 3D printing and CNC machining controllers (5.4, 6.4.1), as well as a collection of pen plotters (2.7.5.1) and various other ad hoc machine projects.

On the question of what a *complete* feedback-based digital fabrication workflow might look like, Section 5.3 develops one that is mostly complete. It is missing the trajectory *generation* step, but builds processes for both high- and low-level planning using models alongside explicit optimization. The result is a substantially reduced parameter space that must be tuned by hand (5.10.3).

On the question of what *models* we need to capture machine physics, and whether we can build them using the machines themselves: Section 4.4 describes a motor controller that can fit its own models (4.5.1), and Section 4.5.2 shows how kinematic models (4.3.2) can be fit from machine data. In Chapter 5 I show how models for polymer melt flow (5.5) and print cooling (5.7) can be developed that combine machine and material physics together using *computational metrology*. In all of these modelling examples I use the machines *where the models will be used* to build the models. This is enabled by the systems architecture mentioned above, and greatly simplifies the deployment of model-based control because models are automatically aligned to hardware. Other model-based approaches require an "offline loop" where models developed using e.g. SI-unit specifications must be carefully calibrated for use across many pieces of hardware, each of which has its own small variabilities (4.8.4).

Reframing machine control from the state-of-the-art's *implicit* constrained optimization (one solved by operators using tacit knowledge) into an *explicit*, numerically solved one, Section 4.6 shows that velocity planners — where state-of-the-art solutions *are* already explicit formulations — can be improved by replacing heuristic parameters with high-fidelity motor and kinematic models. This enables our planners to utilize more of our motion systems' underlying dynamic range, which enables faster overall processing rates (4.7.1) and shows promise to noticeably improve

the productivity of CNC machining (4.8.1). In 3D printing, the optimization-based workflow lets us move more fluidly around in process parameter space which, when coupled with high-level optimizations, leads to decreased printing time compared to the state-of-the-art — especially for larger parts (Section 5.10.4).

In the introduction I also expressed some concern that model- and optimization-based approaches could turn productive processes into black boxes, hiding the intuition that operators and designers already bring to their machines. In Section 5.10.3 I show how workflows that use explicit optimization steps need not become black boxes, by developing techniques that let us mix heuristics with models (5.8.1). In 6.3.1 I showed how virtual machines can be used as a tool for CNC operators to virtually compare toolpaths' velocity deviations (~ deviations from their target chip loads), and to estimate cutting forces from machine data (6.3.2). To demonstrate the use of these tools I combine them with my own tacit knowledge to iteratively develop a machining toolpath in Section 6.4.3. The outputs from our planners are themselves informative: we are after all producing lots of process data every time we use these machines. In Section 5.11.3 I show how process models can teach us about materials, and Section 5.11.1 shows how planner- and machine-sourced datasets can teach us about our hardware's limits and how they interact with process geometry and parameters.

Finally, on the question of how models might be used in machine *design*, Section 5.11.1 (above) doubles as a designer's feedback tool, and Section 6.4.2 demonstrates how models can be applied by designers in their selection of motors and motion parameters for heuristically tuned velocity planners.

7.1 Crossing the Gap: Partitioning Machines

The introduction to this thesis discusses how today's machine control systems partition (split) dynamical operation of hardware (in GCode interpreters) from planning (in CAM and slicers). Relevant background sections covered how this is largely still true even in state-of-the-art research efforts and in distributed systems architectures for robotics. This results in simple problems like computational misconfigurations where systems-scale programs don't have ground-truth knowledge about what devices they are co-ordinating. It also leads to more complex *lossiness* issues where high-level planning steps don't understand what will really happen when their plans are sent to hardware (or what that hardware is capable of / constrained by).

In Section 1.4.1 I explored the economic-scale reasons that even out-of-date partitions remain "sticky" in industry, and Section 1.4.2 describes the underlying technical limits that drive partitioning; those are mostly explained by network timing and computational constraints. Those both lead to a situation where most systems are partitioned across what I called the "realtime gap" between high- and low-level computing. But these split representations — and GCode in particular — prevent us from addressing machine control as a constrained optimization problem. Doing *that* requires that we connect low-level physical models to operation across scales, Section 1.5 explains why making that connection is a key goal in this thesis.

To cross the gap, Chapter 2 rebuilds machine control as set of distributed system modules; components in both high- and low-level systems are both first class citizens here. The new architectural tools here combine patterns from simple embedded networks for spacecraft with patterns from larger, messier and more reconfigurable systems on the web. Section 2.4 overlays programs in those networks to combine dataflow with scripting for configuration and control, and Section 2.5 builds core, reusable motion control software modules in that framework. This mixes fast, stateless operation with smarter high level configuration in an attempt to build high performance and deterministic systems that span the gap and are *also* reconfigurable.

These tools don't eliminate the partitioning problem; parts of machine controllers need to run in embedded devices and parts should run in workstation-scale computing. But they build machine controllers in such a way that the partitions and networks are exposed within the systems description itself rather than hidden in abstract data layers and static interfaces. In Section 2.8.5 I argue that this takes an important step towards the assembly of systems design tools that can express systems design itself (not just machine operation) as a kind of constrained optimization. That would be an improvement over the state-of-the-art, where re-configurability of embedded systems is perhaps not limited by software architectures and interfaces themselves, but by our inability solve the scheduling problem when it is spread across many layers and modules.

The flexibility inherent in the architectures that I built enables me to assemble complex systems, for example bootstrapping model-based control by building simple, single-device models and tests and then sequentially assembling them into higher-order systems (Section 2.7.2). Because we don't change the underlying systems representation in these cycles, those assemblies remain editable and inspectable even once they are "complete;" the result of the transformation is not to *replace* GCode with some new language or representation, it is to expose underlying representations in a unified and more navigable way: we go from GCodes to just Codes (Section 2.7.3).

That means that we can rapidly reassign hardware for different tasks my modifying software-defined controllers, for example using machines both as process modelling tools and as process operation equipment (Section 2.7.4). It also means that we can reconfigure software and hardware modules for new applications, i.e. use the same set of MAXL blocks, circuits, networks, and motion planners and models in both FFF 3D printers, CNC milling machines, and other ad hoc machine projects (Sections 2.7.5 and 2.7.10).

Although the architectures that I built are designed to represent "almost any" partitioning scheme, there are still some key limits (Section 2.8.1) in that regard. Section 2.9 explores how an expanded set of tools in this framework would enable more heterogeneity in this regard, but in this work I mostly locate high-level tasks (velocity planning, system configuration, and tasking) on workstation-scale computing in Python scripts.

Locating the majority of a machine controller's description and state in the operating system has some major benefits; we can deploy much more computing there (Section 2.7.8), and can more easily expose important internal control states (Section 2.7.6). Both of these properties are critical when we move on to apply model-based control directly to machines. When we do apply model-

based methods here, we can use *literally the same* model definitions (in `.py` files) across fitting, tuning, and control steps because each of those take place in workstation-scale computing.

Doing so requires that the realtime gap is crossed carefully because operating systems are inherently non-deterministic; Section 2.5.2 on the "MAXL Gap" explains how I do this. I trade some bottom-line bandwidth performance for determinism; a fixed-gap, fixed-interval trajectory computing and transmission system makes for consistent network and compute loads, and both can be configured against performance measurements made using the system itself. I explored that idea in the section that measures the architecture's network and compute performance (2.7.9) and connects those measurements to MAXL's bottom line performance (which still surpasses bandwidths reported by other state-of-the-art model-based machine control researchers, see Sections 2.7.9.1 and 2.7.9.4).

7.2 Wait, it's all Models?

Always has been…

The first and second halves of the thesis (systems, and then models) at first feel somewhat unrelated, each living in distinct fields (rheologists rarely think about embedded computing), but really they both try to address the same problem: that *machines are poorly represented* in the state-of-the-art, as I discussed in Section 1.3.2.

Since we are talking about representation in both cases, of course the theme of *modelling* connects the two halves. It is not a new idea that machines could be represented computationally: there is much talk about using digital twins for this purpose. The trouble is that twins can easily diverge from their real-world counterparts; misconfigurations between real and virtual worlds can easily emerge when twins are defined by hand / from the top down. This makes subtle differences hard to identify and correct.

I have tried to articulate in a number of places that manufacturing is messy and why machines are heterogeneous in nature. Trying to homogenize systems with rigid top-down definitions is bound to cause more errors than it solves; in Section 1.4 I provided a few real-world and very costly examples. Instead, I think that we should try to design systems that help us (humans) reflect on and manage that complexity based on feedback.

In both halves of this thesis, I drive model building with bottom-up discovery; this means that alignments are automatic and misconfigurations can be detected against *the real* ground truth, which is located in the world — not the hard drive.

7.2.1 As a Core Representation for Systems Integration

In the systems half, I built the *Systems Object Model* (SOM, 2.4.4.1: it's like the browser's DOM but for mechatronics). That provides a view of our machine's *programmatic layout,* i.e. how each

of the machine's hardware and software components relate to one another over networks, and what dataflow links exist between each. In this step, I focussed on developing tools that source that representation from the systems themselves. Network maps and program configurations are discovered via graph traversals, and software definitions are managed via programming tools that reflect on source code itself rather than relying on the manually authored "interface description languages" (IDLs), which are common in other state-of-the-art distributed systems architectures.

These two steps ensure that the systems representations we author high-level configurations and tasks against are automatically aligned to reality — at least in so far as reality is defined by network, firmware, and software configurations. This greatly eases the programming burden on systems developers because they do not need to manually align high-level proxies to low-level firmwares. It also makes it easier to reconfigure systems because changes that may be required in lower level components can more readily be integrated by the application-specific controllers that will rely on them.

7.2.2 Computational Metrology

A good SOM may help us to make sense of how our machines works programmatically, i.e. how we talk to, configure, and task them, but it doesn't make sense of what will actually happen when we do so. In Sections 1.1 and 1.3.1 I discussed how GCode-based systems struggle because of the heterogeneity that machine *physics* present; even if we can describe machines with more flexible APIs, this fact will not go away[98] — even two motors with the same SKU can have meaningfully different characteristics (4.8.4).

The second half of the thesis addresses this heterogeneity by developing a set of physical models that represent dynamical behaviour computationally. A key idea in this aspect of the problem is to develop *computational metrology,* which James Warren, my advisors Neil Gershenfeld, and Jon Seppala, my colleague and fellow systems programming enjoyer Erik Strand and I explored in [349]. I have mentioned this in the thesis on a number of occasions, but should make it (and its value to us) explicit.

As in the systems half of the thesis where I try to source systems representations from systems themselves, here we are trying to source models of dynamical behaviour from machine physics themselves.

In computational modelling, we focus not on the development of e.g. SI-units based measurements of underlying material properties, but on fitting simulations to observed phenomenology. The difference can be subtle, but the models for polymer flow I develop in Section 5.5 are a good example; they predict system behaviour based on abstract internal models that correlate to rheology but don't directly measure rheological properties like viscosity and shear rate. The insight is that models are for making predictions rather than measurements: in the example of 3D printing

[98]Nor should we hope that it would, heterogeneity is inevitable, fun, and being able to specialize our machine designs for particular tasks is obviously productive.

we don't much care what our polymer's viscosity is across melt flow temperatures, but we do care how flows will evolve over time.

In the state-of-the-art it is more common to measure material properties using offline / standalone equipment like rheometers and tensile testing machines, and then integrate those measurements into predictive models. Here we leverage the systems development work to combine sensing with operation and then fit models directly against behaviour. This basically amounts to skipping a few normalization steps: flow models for the RheoPrinter don't make sense in the FrankenPrusa because they combine material and machine behaviour, but they don't need to because both can build their own representations.

There are a few interesting results of the approach that are worth calling out. The first is that we can fit filament compressibility and flow against load cell force readings. I sometimes call that measurement "pressure;" it is analogous to pressure but is actually just a measurement of force applied to the filament by the extruder motor. In Section 5.2.3.3 I relayed how state-of-the-art systems rely on direct measurement of outgoing flowrate for this step; adding computational modelling allows us to remove these observations and instead fit against the internal state — we rely on models to connect flowrate from the nozzle tip to force and flowrate at the nozzle entrance; the models specify only that the system is internally consistent. In the earlier paper that I wrote using a similar measurement [305], these force readings were not even calibrated to Newtons, they were just normalized to the system's max load.

Another printer-related example is in the cooling models. I apply computational modelling to data generated for the steady-state flow test to extract a measurement of a filament's heat capacity (in Section 5.7.1). For the estimate for *how cool* the material should become before another layer can be printed on top, I use the flow models; each has a "zero flow temperature" that estimates the minimum temperature at which material can be extruded at all. Although that is not a direct measurement of the material's glass transition temperature (where it becomes molten), it is a good proxy for how cold it should be to provide structure to the layer above. To combine them to estimate layer cooling time, I do add an extra step to normalize the heat capacity measurement in terms of real-world SI units (J/mm^3). Here this is a required step because I connect that measurement to a cooling model that uses estimated values for heat transfer (into air and into the layer below) that are not "computationally modelled;" I rely on estimates from engineering literature that are relayed in similar units. In Section 5.12.3.3 I explore how the printer's microbolometer (a small thermal camera) could be used to close the loop on these internal measurements using computational modelling to relate observed filament thermodynamics to both heating and cooling phenomena in the printer.

There are also the motor models. Torque curves are normally measured directly using dynamomemters: those are standalone devices that can apply bias torque to a motor while it is run through a range of speeds. Not everyone machine builder has one, but we all have motor drivers themselves. By deploying motor models here, we can replace the dynamometer with a simple disk of some known rotor inertia: models that estimate torque based on current (and against internal measurements for voltage and speed) can then back out the motor's k_t, which correlates current to

torque. These models are invaluable across just about every other machine modelling and control task in the thesis.

Having applied computational modelling to machines, the next subsections relay how I used them to improve dynamics planning and higher level process planning, and how they can provide feedback to machine operators and designers.

7.2.2.1 In Dynamics Planning

In representing machine physics using models, we can more easily describe dynamics planners that optimize *over* those models to control hardware, rather than encoding a separate set of parameters for those planners. Alongside the development of computational models for motion, this step also leveraged the systems architecture developments in the thesis (to apply big computing to the task) and required that I develop a unique planner formulation that expresses the problem in a manner that is computationally parallelizable (based on similar but slightly different formulations from the state-of-the-art in model predictive control).

A description of the planner itself is in Section 4.6, and in Section 4.7.1 I show that it out-performs a type of planner (based on trapezoids) that is common in the state-of-the-art. The performance gain emerges mostly because the model-based planner can plan against a rich description of the machine's underlying constraints. Whereas other planners define per-axis maximum jerk, acceleration, and velocity parameters, this one uses models directly; those relay that e.g. motors have more torque at lower speeds and that machine friction helps us to decelerate.

I enumerated a long list of sophisticated motion control physics in Section 4.2.2 because the new planner formulation shows promise in solving over "almost any" motion dynamics that we can articulate via computational modelling. I show this directly by extending it to solve flow models for FFF 3D printing in Section 5.9. Section 5.11.5 shows that the planner can produce solutions even against the time-varying nozzle thermodynamics present in the printer's rheological system and Section 4.9.2 explores how similar time-varying physics in motors themselves could be included in future versions. I compare the planner's printing speed with the state-of-the-art in Section 5.10.4. It is faster, but it is difficult to say if in this case the speed-up was due to model-based control or was instead based on other higher-lever choices made by the workflow that I developed there. Section 5.11.7.1 calls out the lack of clear evaluation across components in the printer chapter as an overall limit to the work.

Using model-based velocity planning in a formulation where internal states of the planner are exposed also has the effect that running the machine is tantamount to simulating the machine; because the models used by the planner are sourced from hardware, we operate the machine with the same digital twin that describes it. The systems architecture allows us to flexibly deploy software on variable hardware; to run an offline simulation this just means modifying a systems configuration to include *no* hardware.

7.2.2.2 In High Level Planning and Tuning

Models can also be used piecewise in higher level planning and configuration / tuning steps. Whereas I developed machine models primarily with the goal of deploying them in optimization

based dynamics planners, I was surprised to find that they show huge value in these steps. Because new dynamics planners are currently less deterministic than their state-of-the-art counterparts, these may be more immediately applicable results in terms of broader adoption of the work in this thesis.

The basic insight is that models let us select operating parameters with respect to real underlying constraints, whereas state-of-the-art workflows are based on "guess and check" workflows. Modern CAM tools use direct parameters to describe how machines should operate but don't relay to users how those parameters (which are directly encoded in GCodes) relate to physics. In Section 5.8.1 I develop a method for selecting machine speeds and flowrates against flow models using heuristically defined *proportions* of underlying physical limits. Rather than specifying how to operate and guessing whether this will exceed machine capability, we declare that operation should operate at e.g. 90% of its maximum flowrate for infill zones in a print and 50% of its maximum during more detailed parts. This is applied to motion control as well by regulating deployment of motor currents.

Those selections can then be combined with other optimizations against physics, the dynamics planner is the obvious example (it constrains against dynamical limits whereas selected parameters describe steady-state parameter selections alone), but I also provided the example in Section 5.8.4 of selecting nozzle temperature with a combination of cooling models and heuristics-and-model-based flowrate parameters. This combines declarative operation with computation that can select otherwise unconstrained parameters, Section 5.11.4 shows why leaving e.g. the nozzle temperature itself as a free parameter can lead to substantial speed-ups in printing.

I argue in Section 5.11.2 that this is a better way to combine tacit knowledge about machine dynamics with computational control of those dynamics. Instead of throwing parametric darts at the machine constraint wall, we describe the subset of viable parameter spaces that our intuition tells us the machine should use in particular zones and leave the rest up to intelligent algorithms. This shows up also Section 6.2.2 for the CNC milling machine, where I select motion parameters for a classical trapezoid-based velocity planner using models.

Methods like this also greatly simplify the task of machine operation, Section 5.10.3 shows clearly how the end-to-end feedback-based printer control workflow that I develop reduces the space of parameters that must be selected by hand. Those parameters — as I've just mentioned — are also mediated by models, so one intuitively developed set can be "broadcast" against new physical configurations.

To subsequently develop that intuition, the new tools that I built also enable us to visualize rich outputs from our hardware once we have run jobs (or before we do). Section 5.11.1 shows this most clearly: underlying physical limits in hardware are overlaid on printed (or simulated) trajectories to connect machine physics with process outcomes. We can use these data to understand why certain approaches work and others don't, and also to understand why errors occurred; see Section 5.10.5.1. I show how these may be used in CNC machining as well in Section 6.3.

In this work I don't engage in any path planning optimizations; each of the systems I work on operates based on trajectory geometry that has been defined in existing state-of-the-art tools. In the printing chapter I ingest GCodes from slicers, strip them of all but the geometric operation, and develop the process parameters and dynamics control on top of that and in the machining chapter I run my systems only for velocity planning and operation of the machine (and data collection). Section 7.7.1 briefly describes how the models and methods here could be used directly by — or extended to become — path planning algorithms that are also model-based.

7.2.2.3 In Machine Design

In the very beginning of the thesis I was making a workforce development argument; the new control systems we build should help more people become machine *builders* as well as users. The systems development chapter is most directly related to this because it is explicitly oriented around the way that machine controllers are assembled, reconfigured etc. But the tools that I've just described also have a natural relation to machine design: if we can express machine operation directly in relation to underlying physical constraints, it becomes more obvious how machine design decisions relate to operation. A goal there is to provide machine users with the thread they need to pull on so that they might become machine builders.

That is, we operate machines against their physical constraints and design them in an attempt to remove or minimize as many of those constraints. But it is often difficult even to sophisticated machine designers how all the decisions that they make will ultimately relate to bottom line performance. This is especially true when machines will interact with a heterogeneous set of materials and be used to make a highly variable set of part geometries; machines that work well to make small detailed parts (where precision above all is important) may perform poorly when tasked with fabricating larger, simpler geometries (where total power deployable is more important).

So, the outputs that I showed in Section 5.11.1 (on "Learning from Machines") have obvious utility for machine designers too. I also showed in Section 5.11.3 how flow models can be used to understand materials and nozzles in more detail. In the CNC machining chapter I drew up a much more explicit example of this, using in Section 6.4.2 a set of model-based tools to select motors for an existing CNC machine kit. With models of my own motors and the *particular* hardware that I had assembled, I was able to push the machine's performance well beyond what the kit manufacturer recommended.

7.2.3 Over Machine Lifespans

Model based control is probably an excellent basis for the maintenance and monitoring of machines over their lifecycles as well; in Section 4.7.3 I quickly showed that data collected from a machine can be used to improve models for control. Again, because the controllers work directly with those models, refitting models is equivalent to updating controllers to reflect changes that have occurred during use. This is "continuous improvement and deployment" (CI/CD) for machines. Although the models that I use for polymer flows are formulated in a manner that makes this difficult to do for the 3D printing process, I discuss how they could be updated to remove this limit in

Section 5.12.2. The work here already makes some progress in that regard because these machines can generate more data than they consume — in the printers this is about one gigabyte per hour (on plans that are encoded in a few megabytes); more data of course being an important basis for any future work on modelling.

This connects also to error detection, in Section 5.10.5.1 I showed that the areas where models are poorly matched to data clearly overlap with areas that printed poorly. That probably means that those areas represent state spaces in machine operation that were underrepresented in the initial data sets; operating machines in those regions naturally generates data to improve fits there. But once model fits are well established, areas where models disagree with data are more likely to predict genuine out-of-band errors or disturbances; those could be used to identify and correct errors (potentially on-the-fly), or identify upstream changes, for example variable batches of input material or other unintended or unforeseen supply chain edits.

7.2.4 But Leave Room For Heuristics

I've already made reference to the application of additional heuristics on top of / alongside model-based control. Modelling is fundamentally about developing compressed representations: we use them to concisely describe complex systems. This means that good and bad models all miss *some* phenomenology. The models I develop here are no different, in particular the motion controller doesn't understand machine vibration or stiffness and the printer workflow doesn't really understand geometry, details about part cooling, and e.g. that outer perimeters should be more carefully printed than infill tracks.

If I had stopped to develop models for each of these aspects of the problem I would never have finished this work; where models miss it is important to be able to still use heuristics. Developing the methods that I deployed to mix heuristics with models was some of the most interesting work perhaps specifically because they are somewhat non-rigorous and simultaneously the most useful. In some sense this is the same point to the one I just made in Section 7.2.2.2; because there are *so* many competing concerns in machine design and control, simpler approaches or distillations of that complexity can be more valuable than making attempts to exhaustively map[99] them.

[99]One of the all time great modelling anecdotes is Norbert Wiener's *"The best material model for a cat is another, or preferably the same, cat,"* - as in, messy systems are best modelled by themselves. This is kind of the point of the machine that measures its own models, but the counterpoint is just as valuable, and it takes the form of Borges' small story **On Exactitude in Science** which is itself quite small, I can even just copy paste it in here, check it out: *"…In that Empire, the Art of Cartography attained such Perfection that the map of a single Province occupied the entirety of a City, and the map of the Empire, the entirety of a Province. In time, those Unconscionable Maps no longer satisfied, and the Cartographers Guilds struck a Map of the Empire whose size was that of the Empire, and which coincided point for point with it. The following Generations, who were not so fond of the Study of Cartography as their Forebears had been, saw that that vast Map was Useless, and not without some Pitilessness was it, that they delivered it up to the Inclemencies of Sun and Winters. In the Deserts of the West, still today, there are Tattered Ruins of that Map, inhabited by Animals and Beggars; in all the Land there is no other Relic of the Disciplines of Geography."* … The point being that no, a complete model is not useful, because it does not do the thing that we want models to do, which is to reduce a complex thing into a simpler representation that we can understand. OK.

In other cases, using heuristics for control can be preferable because they are simple and interpretable in terms of the outputs that they generate even though their input parameters are set directly. A trapezoid based velocity planner will not tell us why our machine can't move faster than $100mm/s$, we know that it *will* because that is the limit that we manually configured it to. Alongside the computing constraints that I have discussed, this is perhaps another reason why state-of-the-art industrial controllers (see Section 4.2.6) are still configured against direct parameters rather than models and solvers.

But again models are not incompatible with these systems, we have the methods that I have described which mix heuristics with models, and also the work I've described that uses models to select those direct parameters (6.2.2). We can also overlay models on plans that were generated using these more classical planners, as in the section in the machining chapter where I estimate cutting forces (Section 6.3.2).

7.2.5 With Regard to Economic Partitions

In the introduction I mentioned the *economic* partitioning problem (Section 1.4.1) and its relation to GCode's pervasiveness: systems integration must happen across firms who each develop *parts* of an integrated system. Where internal representations of those systems are difficult to see or edit, integrations are harder and existing partitions and formats become *stickier* [51].

In modular or networked control systems we have a similar challenge; control components are developed by many firms and they each have the challenge of exposing their contributions to systems integrators. For technically complex components that are tightly integrated into other systems this is ultimately about explaining how the components *work.* The standard practice here is to write a datasheet [350], i.e. a document that explains some of the component's inner working model, how systems integrators should expect it to function in certain states, and how they can interact with it (over which protocols, accessing which memory fields etc.). These documents can be extensive, for example the manual for Delta Tau motion controllers (a state-of-the-art modular motion system) is over 800 pages long [236] which represents a clear burden for systems integrators: presumably they will have to read it.

Ultimately this step is about reaching from the component's actual implementation (engineering work done within the firm) through a compressed representation of how it works (the datasheet) and back into another integrated system. Datasheets are sometimes lossy for the same reason that other manually authored configurations are: reality as defined in the implementation can diverge from what was last written down. In industry, companies who develop a particular device are sometimes purchased by other firms who then subsequently move or modify these documents: they become inherited artefacts and the original engineers who authored them may be long gone at the time of their use.

I hope that this work's relevance here is clear: the systems in this thesis base schema collection on discovery, which is subsequently based on automated tools that scrape schemas directly from source code. That is, I try to reach directly from the implementation layer to the description layer to

ensure consistency across both. The world of software design does not include datasheets (especially where source code is open source) because well written software *APIs* convey enough information. It helps that computing is much more homogeneous than hardware: that makes operation of those APIs more predictable. But again, I am proposing that these dynamical models can help in that regard.

The patterns that I deploy to collect data schemas from source code do exist in some software serialization libraries for distributed systems (see Section 2.6.1.1), but they have so-far not been deployed in hardware oriented systems. I would contend that they are more useful here exactly because of the economic and physical heterogeneity present in machine systems. We have not yet had access to the level of embedded computing required to bring them down this far the stack; they consume extra memory and compute cycles to run. The compiler tools that I used to build them (for "template programming") are even somewhat new relative to the build systems used for most firmwares; I had to manually configure most of my firmware development tools to target these devices with more modern versions of C++. But that is changing; microcontrollers are faster and cheaper than ever and languages like Rust are emerging as clear improvements over template-programming based strategies. In particular, open source hardware has a documentation problem; people who develop open hardware devices are burdened by the extra effort required to maintain it [351]. Developing discoverable schema in this way may help to ease that burden.

Overall, each of the topics that I have tried to cover for digital fabrication world are different aspects of one larger constrained optimization problem (see Section 1.5) which has many degrees of freedom. It is impossible to work on them in isolation, and it is difficult for individuals or individual firms who master one aspect to understand exactly how their contribution relates to the rest of the state-of-the-art. The machine control industry remains largely "partitioned;" a number of firms each offer complete and competing technological stacks that they protect with IP and secrecy [43]. For some time the computing industry was different, an emergence of standards for interoperation within computer systems led to what the authors of *Design Rules Vol. 1* [37] call a "globally distributed, asynchronous systems design optimization effort;" competing firms all work together to produce components (CPUs, RAM, GPUs, displays, network interfaces, etc.) that are subsequently integrated into individually unique computers that remain globally addressable at the application layer via the power of code portability and computer science.

Key to that economic layout is the fact that component performance and specification can be easily relayed between firms. In this regard I would argue that models which are developed piecewise could fill a similar economic role. For example controls and CAM developers could virtually test their algorithms on a myriad of machine systems if those systems were well described as computational models or more complete "digital twins" (that include 3D models of hardware). In the same way, software developers can deploy their code on a litany of different computers via software virtualization; when the developer of an iOS app tests it, they often use a virtual iPhone. In the future work Section 8.1 I discuss the possibility of developing a standard for this purpose. Section 5.11.3 shows how models for flow can express dynamical performance of nozzles and materials, those are richer descriptions than the thin specifications currently provided by

manufacturers of both and could help users and integrators make more informed decisions when choosing either. Good motor manufacturers already include torque curves for their products. Those can change depending on driver voltage and type, and a torque curve does not describe how the motor will integrate with (for example) another firm's linear axis' properties (inertia and friction). In Section 4.3 I show how piecewise models for each can be developed, and in Section 6.2.1 how they can be combined to intelligently select combinations of the two. I had to build those models, which required that I buy the machine kit and the motors to test. Providing computational models instead of datasheets could cut through some marketing layers and provide engineers with better tools for decision-making. I am not the first to imagine this type of economic layout, here I only want to draw the connection between the contributions that I did make in this regard and their broader impacts.

7.3 Wait, it's all Connected?

A lot of the work in this thesis is oriented around the modularization of integrated systems. This is harder than simply building integrated systems that each do one specific task, except for the repeated labour that can be saved via those modularizations and e.g. the ability we then have to rapidly reconfigure the components into new systems. If the point of the thesis was simply to control 3D printers using feedback, all of Chapter 2 could seem somewhat irrelevant. Section 2.8.4 on OSAP and the OSI model is a start to this discussion: architects of the internet (which itself does not exactly follow the OSI model) make some points in an RFC that revolve around the reality that modularizations / abstractions can lead to inefficiencies and e.g. repeated computing steps because data hidden by one layer remains relevant in another.

This is an ongoing tension in systems design: abstractions are great except for where they fail, and in those cases they can be severely limiting. Whereas the work that I present here follows the end-to-end principle [62] in some regards, in reality decisions made at each layer were informed by my own understanding of constraints that emerge in others.

For example in Section 5.2.4.1 I shared how the FFF work in this thesis is primarily related to a line of research begun by TJ Coogan and David Kazmer, who developed an instrumented extruder similar to ours in [301] — it also adds a load cell between the extruder motor and the nozzle to estimate melt flow pressures. The key idea there is that we should be able to turn our processing hardware into process measurement equipment, i.e. use the extruder itself as a rheometer via computational metrology. Filippos Tourlomousis (a former collaborator) and Jon Seppala (one of my advisors) get the credit for bringing this idea to our research group at the CBA. Filippos built a prototype of the RheoPrinter's hotend (Section 5.4.1) and I extended that with a design pattern developed by an FFF YouTuber to implement the filament sensor [325] (adding a feed rate encoder, and calibrating width measurements), and then began the long project of using that device as a tool to connect measurements to modelling and control.

This turned out to involve almost all the other work that I have described so far on improvements to systems architectures, motion controllers, and motor controllers. For example the closed-loop motor controller in Section 4.4 eventually supplanted the need for the RheoPrinter's filament sensor and enabled me to develop a coupling between rheological models and the motors that drive them, the flexibility of MAXL's basis spline representation (Section 2.5.3) was a key requirement for *transmitting* the complex motion instructions involved in nonlinear control of flow and MAXL's piecewise assembly of motion controllers enabled me to run both *model-building* controllers on the RheoPrinter (which must be operable even when we do not yet have models) and the *model-based* controllers. OSAP and PIPES were key to all of this as connective tissue and to enable assembly of those piecewise controllers, and their flexibility allowed me to rapidly retrofit the FrankenPrusa (Section 5.4.2) in the later stages of this work in order for me to compare this new workflow to the state-of-the-art.

That is to say that I largely organized my systems design activities around a few core tasks. I tried to architect them such that they will be extensible in other use cases, but in reality that remains untested. I was able in Chapter 6 to rapidly redeploy them for a new machine and process (and they showed value there), and in Section 2.7 includes examples of a handful of other kinematic and ad hoc systems. Section 2.8.1 recounts the ways in which I anticipate that these architectural updates remain limited, but it is impossible to know for sure how developments or omissions I made there may limit other machine developer's applications.

7.4 Basis Splines and Systems Assembly

In MAXL (Section 2.5) I use cubic basis splines (2.5.3) to encode motion. I discussed why these are useful with regard to the *composability* of various kinematics and other controls tasks in 2.8.2.

They do introduce some interpolation errors (I quantify those under the next heading) but they also relate neatly to constraints from our motion systems *and* our computational systems, which warrants some discussion.

7.4.1 Spline Errors and Motion Constraints

One issue with these splines is that they are not a direct interpolation.[100] This represents some deviation between the control points that MAXL blocks transmit and the positions that motors actually interpolate through. In one sense the basis spline is a low pass filter on these points, which is an obvious problem for high precision motion control.

So, how *much* of a problem is it? Well, I would like to first point out that motion through a truly sharp corner at nonzero velocity requires an instantaneous change in velocity on at least

[100] i.e. the interpolated path does not actually pass through the control points.

one axis (an "infinite" spike in the acceleration of that axis),[101] so we need to do some smoothing of linear piecewise paths *or* accept that our machines will flex and vibrate in unpredictable ways at these junctions.[102] The better approach to the problem is to actually encode smooth paths according to a geometric tolerance, which is the basis of filter-based velocity planning approaches (see Section 4.2.5 and 4.2.8.2) but piecewise linear paths remain common in practice and in these cases encoding planned trajectories over line segments as basis splines actually helps because we eliminate these infinite spikes in acceleration from the curve which is followed by our hardware.

Still of course we would like to actually quantify the error. As it turns out, Delta Tau controllers[103] also use basis splines to transmit motion to servo controllers. In their datasheet, they quantify the exact deviation error ([236] page 744):

$$\text{Err} = \frac{V^2 T^2}{6R} \tag{7.1}$$

Where V is velocity along the interpolated path, T is the timestep and R is the radius of curvature at the point of measurement. So if we take a relatively high speed and tight curvature ($V = 500mm/s, R = 10mm$) and use the ~ four millisecond timestep that I used most often in the MAXL systems discussed in this document, we only see about $70\mu m$ deviation:

$$\text{Err} = \frac{250000.0 \cdot 0.004096^2}{60.0} = 0.0699mm \tag{7.2}$$

In practice high speeds and tight corners don't appear simultaneously; commercial 3D printers may advertise speeds of up to $500mm/s$ but only ever achieve these velocities along straight paths - see for example Figure 1.8 in Section 1.3.3. Machines tend towards lower velocities in areas of high geometric complexities because cornering velocities are limited by actuator torque and machine stiffness[104] so even if we blindly produce control points at a fixed time interval we end up packing more points into areas where geometries need more definition.

To show how this applies to real trajectories, I plotted interpolation errors along one of the planned CNC machining trajectories from Section 4.8.1 in Figure 7.1 below. These are measured numerically by taking the distance between each control point and the corresponding interpolated point.

7.4.2 Splines and Systems Integration Constraints

I think this discussion is especially interesting because it also relates to constraints from the *systems integration* perspective. For example some GCode interpreters work well with simple toolpaths

[101] This can be a bit non-intuitive because we regularly see robots and machines carry velocity through corners. Mathematically, imagine a 2D path in the XY plane that starts along the X axis and then makes a 45° turn into Y and consider how the Y velocity changes as we pass through that corner; if we begin the second segment with any velocity V along the path, the Y velocity instantly jumps from 0 to $sin(45) \cdot V$. See some more notes on this in the section on trapezoidal velocity planners 4.2.4.

[102] In practice this means that belts will flex, following errors in servo loops will briefly increase, or stepper motor rotors will lag farther than normal from the stator's field. These junctions are where steppers most often skip steps.

[103] A state-of-the-art modular motion control system, discussed in Section 4.2.6.

[104] Basically all of Chapter 4 is about exploring this relationship.



Figure 7.1: Interpolation errors (top row, millimeters) along a trajectory in three axes; x (blue), y (orange), and z (green). These correlate well with axes acceleration (second row). The trajectory is the adaptive machining test generated with my velocity planner from Section 4.8.1.

but struggle with complex geometries that are encoded using very many small line segments. This happens for three reasons. The first is that GCode interpreters are often configured with a fixed *number* of segments in their planning queue whereas machine dynamics require that their velocity planners can see out to a minimum *stopping distance.* The second is that extra segments require extra compute to plan over. To ameliorate these issues most interpreters include a minimum segment length.[105]

The third pertains to network limits themselves in cases where we have to transmit GCodes from a host computer into the interpreter. While we don't normally think of USB or SD cards as "network links," both have real bandwidth limits.

[105]Enforcing a minimum segment length means that a minimum *number* of segments will also correspond to a minimum physical distance which can be configured such that this distance is equal to the machine's actual stopping distance. It also approximates geometric filtering over vanishingly small line segments whose size is at or below the machine's realistic resolution requirements. I discussed this alongside those filter-based approaches to planning in Section 4.2.5.

For reliable control over networks we prefer to have deterministic network loads (Section 2.2.2.1) and using a fixed time interval accomplishes this. It also means that in the cases where we do need some buffering we can specify the size of that buffer against its real constraint which is time based,[106] not spatial.

There is also a computational constraint at the terminal end of the problem where trajectories are actually actuated by stepper motors. To increase precision and smoothness of steppers modern systems use *microstepping* drives that interpolate full steps with smaller increments through the stepper motors' electrical phases.[107] At high speeds the stepping rate can become quite fast indeed, for example a linear axis moving at $500mm/s$ with a $2mm$ pitch belt on a pulley with 20 teeth requires:

$$\text{Steps per Second} = \frac{\text{Speed } (mm/s)}{mm/\text{ Rev}} \cdot \text{Steps per Rev} \cdot \text{Microsteps} = \frac{500}{40} \cdot 200 \cdot 128 = 320000Hz$$

This is fast: at $320kHz$ we have only $3.125\mu s$ of time between step pulses, on a modest $48MHz$ microcontroller that's only 150 clock cycles in between each step. When we interpolate through step-counting representations for motion we cannot miss any of these pulses whereas splines can be evaluated at any time, which relieves the burden on interpolators. It does require that we drive steppers at a higher level, but this is becoming common as in Prusa's phase stepping. The motor control architecture that I use here also enables "phase stepping" via direct control of H-Bridges (see Section 4.4).

7.4.3 Constraints-Based Spline Configuration

The overall point in this section is that the representations we use for motion relate to constraints that emerge from all ends of the machine control task: real dynamical limits based on physics and computational limits in our networks and computing systems.

For example, we could summarize a motion system's dynamic constraints into one time constant[108] and its control network constraints in terms of bandwidth and delay. The dynamical constraint provides an upper bound for the spline: intervals much larger than this value will introduce nonnegligible errors in areas of high dynamic change. Network constraints provide a lower bound: intervals much smaller than which will saturate link bandwidths.

This is one particularly salient example of a *type* of problem that is present throughout the machine design and control task overall. That fast machines also need fast networks and bigger computing

[106] We can also calculate machines' minimum stopping *time* as easily as we can calculate their minimum stopping *distance.*

[107] Stepper motors typically have only 200 "full steps." In motor parlance this means that they have 50 electrical rotations per mechanical rotation. A full step makes 90° of an electrical rotation, energizing one coil entirely and de-energizing the other. To interpolate, microstepping drives apply sinusoidal currents to these coils.

[108] In case *time constant* hasn't appeared yet in your reading of this text, or you're unfamiliar, this value (normally τ and in seconds) is a system characteristic that approximates how long it takes to respond to inputs. More precisely it is the time after which a system has reached 63.2% of the *steady state* value that it will eventually reach after a new input has been applied.

is not immediately obvious but can be an important point of consideration; in this thesis I tried to develop tools that would reflect these constraints to machine designers and users: that means models for motion but also computational systems that are more inspectable than their state-of-the-art counterparts. Combined properly, these systems could be used together to help understand and avoid these issues earlier on in machine design and prototyping cycles.

7.5 Open vs. Closed Loop Motion, and Why Stepper Motors

Closed-loop / feed-back control is typically seen as being superior to open-loop or feed-forward control. The work in this thesis combines aspects of both in slightly unexpected ways: I control stepper motors (which are normally open-loop) as closed-loop motors using field oriented control (Section 4.4) and then develop a feedback-based velocity planner that does not actually consume new data from hardware at runtime. This warrants some discussion.

Firstly on the velocity planner. In a number of places I note that it uses feedback from hardware to develop the models that it uses to solve velocity plans with. This means that although it does not integrate new data while it is operating, it does use an understanding of the system under its control that is data-based; it is a feedback controller in the larger loop through those modelling steps. If there are no unforeseen disturbances during operation and models used in feed-forward control are well-matched to reality, the approach is actually preferable because it does not introduce any additional system lag. Feed-back controllers rely on error measurements to operate and so can only operate where there is *some* error — that is, perfect operation is impossible using feedback only. The goal of most model-predictive control is to merge feed-forward predictions with feedback. The planners that I develop here would likely be improved if they could combine real-world measurements at runtime, but instead I combine them with lower level feed-back controllers for disturbance rejection and this works well enough for the time being.

Those lower-level feed-back controllers are then critical for model building. Entirely open-loop motor control is typically lower performance not only because it limits online disturbance rejection but because it prevents us from modelling equipment; closed-loop controllers can be used as sensors if we can read their internal states but their open-loop equivalents cannot.

But starting from scratch with closed-loop actuators is difficult because they cannot be controlled at all without first developing gains for their internal loops. This is where I think stepper motors, which can be successfully controlled in both modes, are invaluable for machine systems. For example in this work I use the motors' internal geometry to calibrate nonlinear magnetic encoders, effectively treating the stepper motors' open loop (and unloaded) control mode as a ground truth. Their internals are quite precise because they are designed for this purpose, the small teeth that align steps are stamped from magnet steel in a highly repeatable manner. I then use those calibrations as a basis for closed-loop control of the motor.

But I then develop low level controller gains for these motors by hand, which I called out as a limit to my methods in Section 5.11.7.5. The mechanical frequencies that are achievable by

lightweight 3D printer motion systems are probably still beyond the limits presented by these motor's closed-loop drives (especially with non-optimal gains), but not beyond the bandwidths that can be achieved using optimal feed-forward control and open-loop motor driving. Tuning closed-loop gains to their maximal stiffness is difficult because extremely reactive gains can also lead to control instabilities.

This is the primary reason that I think stepper motors are actually a superior basis for future work in advanced motion control; they are more precise under open-loop control which is a likely choice when feed-forward models are sufficiently developed. Those should be combined with disturbance rejection using local feed-back. The other reason has to do with bootstrapping: while we are developing machine models we can rely on open-loop motor operation, that is simpler than developing low-level gains alongside models.

Finally, stepper motors are also ubiquitous and available in a wide array of sizes and specifications; machine designers often want to choose motors that are uniquely suited to their hardware. They are also cheap.

7.6 About GPUs for Motion Control

So you're telling me I need a GPU to control my machine?

While the controllers that I develop here are measurably better than other solutions, they run on expensive computing hardware. For example, during the development of this thesis I used an RTX 4070 GPU ($700) to run machines whose BOM costs were between $200 and $1200 each. Of course this trade-off might not seem as backwards were we talking about half-million dollar milling machines.

However, the cost of these devices will likely drop because their development is an incredibly active area of investment and research. Smaller, simpler versions are being deployed in hardware throughout the field of robotics, and it is likely that the planner that I developed here could run on those cheaper GPUs. A main cost driver is in the GPU's memory, the planner only consumes about one half gigabyte of that.

I would also point out that as in the discussion above, the GPU-deployed realtime velocity planner is only one component of this work. The value of models in slower planning steps covers more distance from the currently deployed state-of-the-art to the future than the last mile benefits that come from direct control using models. That is to say that models are probably better feedback tools *first* for the humans who operate and design machines, and as a shared representation amongst workflow components.

7.7 The Connection to "AI" and Reinforcement Learning

For a thesis written in god's year 2026, there is scant mention of AI and Reinforcement Learning (its control counterpart) in this thesis. There are a few reasons for this.

The primary of which goes well with the prior section on models: models that are partitioned more easily let us use them *across* tasks for machine building. Neural nets and RL black box these aspects and run them end-to-end. For example, we wouldn't be able to pull heat capacity from a flow model and relate it to a cooling model using neural-network based approaches, or combine motor models and axes models to compare motor choices if both systems were modelled end-to-end.

Assembling smaller models into systems also has the benefit that each component can be inspected, as in the outputs that I showed in Section 5.11.1. Using real representations of physics enables the heuristics-based tuning workflows that I discussed above and in Section 5.11.2.

These models also form the basis of RL-based methods. Those are largely trained offline using simulation, so ground-truth modelling of the systems they use is obviously important. As I've mentioned the state-of-the-art in machine control makes it difficult to develop those models, and I make progress in developing systems that enable this.

Machine heterogeneity is worth thinking about here as well; end-to-end / RL-based controller need to be re-trained when the systems that they control change or new physics are added. That means that every new piece of hardware requires expensive compute time in order to bring RL models to bear on them. Classical controllers still lie under most RL-based methods for these reasons, e.g. a robot dog or humanoid still runs on lower level servos that track velocity or torque values.

Machine systems also require much higher bandwidths than the systems where RL is typically applied; a robot dog's foot can miss its target by centimeters at no loss to the overall controller, but in machine control every hundredth of a millimeter counts.

There is also inspectability to contend with. In a few places I have noted that classical controllers are preferred industrially because it is clear why they operate in the ways that they do. Controls that are based on direct parameters are the most inspectable in this regard, RL-based controls are the least inspectable. The velocity planners that I develop are less predictable than trapezoidal and FIR-based planners, but they are clearly not black boxes, as I have showed in a number of places already.

7.7.1 Connections to Path Planning

If anything I think that reinforcement learning has clear advantage in higher levels of these systems. They should be developed on top of or alongside of model-based machine controllers for path planning. In that step, goals are often more difficult to describe and the space of possible actions is much greater. Reinforcement learning algorithms are excellent in this regard, and the

computational frameworks used to develop them can handle messier geometric representations that may not be differentiable.

In fact, having brought constraints from lower levels of our systems up through velocity planning and feed-and-speeds configurations, the natural next step is to use the same for path planners themselves. Reinforcement learning algorithms could train against these constraints (rather than learning them indirectly), but there are other ways we can connect the two layers that are less involved.

Probably the most obvious example of that is to automatically determine cornering radii for adaptive machining paths (see Section 6.4.3). These radii could be related to motion models and the target feed rate because achievable cornering velocity is directly related to permissible acceleration. In FFF, slicers should prefer to cut corners (but thicken fill) as they recede from exterior perimeters of a part: but by how much, and how much thickening? Motion and flow models can tell us.

8 Future Work

Each chapter contains notes on future work that relate directly to those topics, in systems 2.9, motion 4.9, 3D printing 5.12, and machining 6.5. The topics below relate across components.

8.1 Developing A Spec for Machine Digital Twins

One of the ideas that developed as I was working on these controllers is that of interfacing the (already software-based) controller to a "digital twin" of the whole machine. I think the most productive version of this idea is to use the digital twin as an interface between the machine designers' understanding of their hardware's kinematics and the controller.

For example, we can take for granted that a machine designer will have a mostly complete CAD model of their output. Those CAD models are normally assembled using *constraints* that generate mechanisms: linear slides, rotary joints, etc. — everything we need for a rigid body simulation. A hope had been that we could develop a representation of these assemblies that could be exported from a CAD tool and plugged directly into the planner: it could then run and visualize the machine *virtually* to ensure that the designer's idea of how the machine was going to move matched to reality. Data from motors could be used to verify and improve these alignments in all the ways

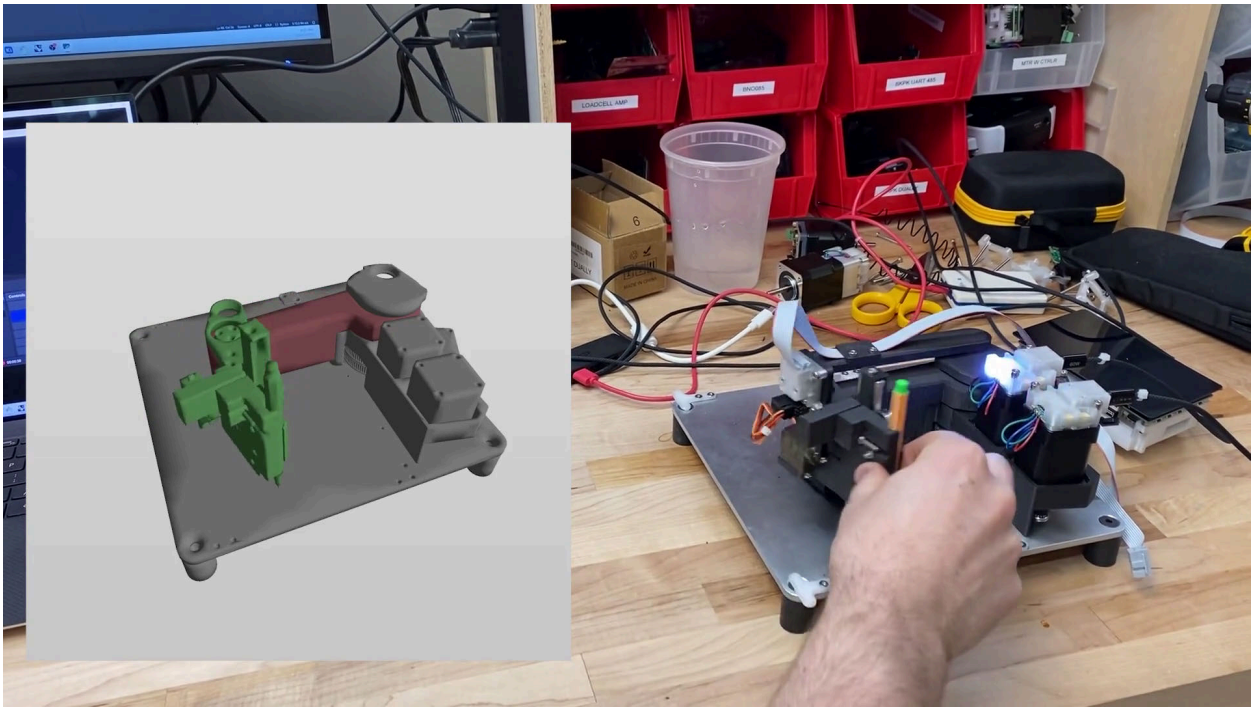


Figure 8.1: A digital twin for the Little Guy.

I've discussed previously. This representation could be as simple as a set of `.stl` or `.step` of the rigid bodies, and some `.xml` or `.json` outlining their relations to one another (joint types, etc.), as well as linkages to actuators.

The goal of this tool would be to enable machine designers to quickly bring their machines to life, using a representation and interface (their CAD model) that they are already familiar with. Some such projects exist [352]. Done well, the tool could also help machine designers learn about their designs' performance *before* they build it, e.g. that a motor should be made more or less powerful, or a transmission element made stiffer, or that their mechanism could be improved to increase the machine's range of motion, etc.

8.1.1 Discovering Kinematics

There is a lot that I didn't get to on the topic of motion control. Probably the most frustrating of which is the task of more generally fitting kinematic models from motion data, which could be done separately or alongside the digital twin specification I just described. The idea here would be that we could develop well known motor models, attach those to new machines, and automatically discover how they are configured vis-à-vis the machine's transmission elements and mechanisms. This would be invaluable to machine builders because working out kinematic models is cumbersome first in getting configurations correct (motor offsets, transmission ratios), and then for refining those models: machines never go together exactly the way they are planned (no two lines are completely parallel, etc.), and to achieve real precision we often need to fit kinematic models to data. For example the simple task of figuring out a machine's exact working volume is normally done with simple trial and error.

In some proposed schemes, we could develop a library of possible kinematic arrangements, attach an accelerometer to the end effector (as I have done here to work out resonances), and learn how each actuator's acceleration corresponds to end effector motion. Backing out linear transforms is presumably as easy as filling in the matrix that relates one system to the other, but nonlinear transforms would also require that we guess the composition of the transform function and its inverse. Imprecise models like neural networks might get close in this regard, but are probably too much of a black-box to be truly suitable to the task: in machine design and control, it is invaluable to have precise definitions of our core systems. New approaches are emerging that use regression to find sparse representations for nonlinear systems [353], those may be the right vein of techniques to start with.

Adding computer vision to learn kinematic models more concretely is another proposal (or hybrids of each), and my colleague Quentin Bolsée is working on this currently at the CBA. There are a litany of other data collection approaches for this task: probing known geometries (like the method outlined for bed leveling in this work) is a common one, and could be extended to correct for XY kinematics as well as in Z (as is in fact common on some 3D printers).

8.2 End-to-End Fitting of Models

Although building models piecewise is a valuable capability during systems assembly, once machines are together it can be more productive to fit models in a more-end-to-end manner.

For example, in Section 4.3, I estimate parameters for motor models and frictions separately, and measure inertias by hand. It would be ideal if we could update estimate more of these parameters without disassembling the hardware, e.g. fitting motor model parameters, kinematic parameters, and friction parameters all from the same datasets, using limited prior information. This also would be important for retrofitting unknown equipment and for improving models over time using data generated during normal operation.

For a simple example, the motor encoder calibration from Section 4.4 is sometimes imperfect and results small oscillations that are visible under certain conditions in time-series data. Fixing these calibrations manually would require disassembling machines to decouple the rotor. But those oscillations show up exactly at the frequency of the motor's rotor, so we should be able to isolate those errors and correct them using "generic" machine data, i.e. the type that it produces during everyday operation. But in other cases the isolation is not as simple; for example when a motor rotor is connected only to one other mass it is difficult to ascertain from motion data alone if the overall inertia is attached to the rotor shaft or to the linear axis that it drives; we need a spare state to differentiate the two.

However, model components often *lie in the same axes* of our systems' state space. For example a friction term applied to the motor rotor and a friction term applied to the sliding axis of one of the CNC machine's axes are *degenerate.* We could increase one, or the other, to net the same new estimate for the machine's behaviors because they are linearly coupled. I discussed this at more length with regard to printers in particular in Section 5.12.2, but the ambition applies across machine types.

8.3 An Open Ecosystem for Machine Control

I have mentioned in a few places how other researchers work is limited by closed motion control systems; Sections 4.2.8 and 5.2.4 both contain some examples; there is much work in modelling and sensor integration at a hardware level, but researchers cannot readily combine their contributions with their machine's *existing* controllers. Where they do, systems are surprisingly complex (Section 2.6.2). In particular, the velocity planners present issues to these researchers because they cause their test path velocities to deviate in ways that are difficult to predict. This means that time-series data are difficult to reconcile with the toolpath geometry, which makes all subsequent modelling steps more challenging. It also prevents them from easily integrating new online control strategies.

This is true in other domains where machines need to be "hacked" and in mechatronics integration in general, for the assembly of all manner of automation equipment. I discussed the economic

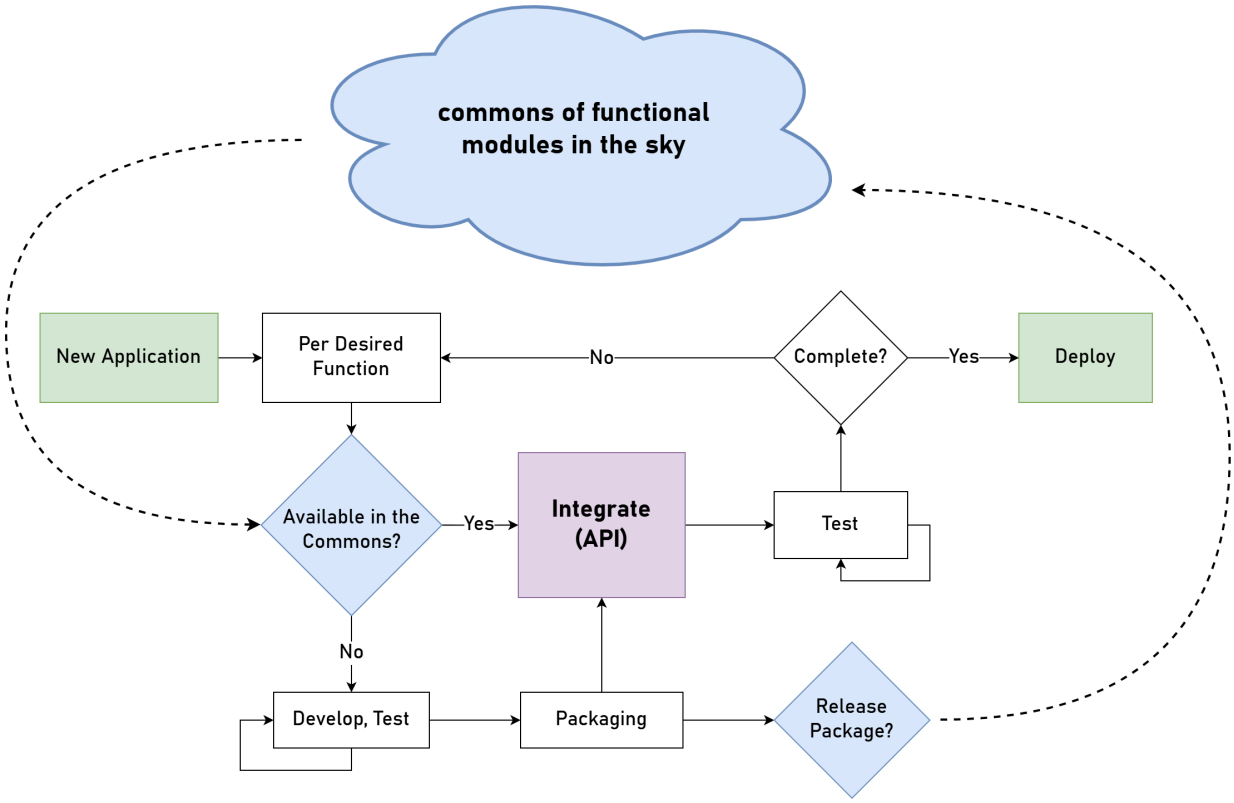


Figure 8.2: A diagram of how users of Open Source Software developers interchangeably use components from a commons of functional modules, and develop and publish their own. Software has many "built-in" tools for modularity, but hardware tends to resist generalization. Modular hardware approaches try to bridge this gap, to enable the development of a commons of reusable devices.

partitioning problem directly in Section 1.4.1 because I think that this existing arrangement is what really fundamentally limits our progress in the development of new and better machine and manufacturing systems. Those are always deeply interdependent on one another, and in the state-of-the-art there is often simply not enough information provided by certain core components to rapidly build new types of systems that push the envelope of what is possible.

The clear next step for the work in this thesis is to continue developing the controllers into an ecosystem that other researchers, machine builders and users can easily integrate into.

I originally developed OSAP, and then PIPES with this outcome in mind: the results for improvements to machine control were secondary. Section 7.2.5 already explained why I think the contributions there are valuable in this regard. Applications of the same principles (modularization, new systems assembly tools, portability) have driven the runaway success of open source software efforts as explored in [39] and [38], who note that the modular ecosystems that enable distributed collaboration on open source software are themselves modular, performant and extensible. That is, the systems that we use to compose systems are themselves composable and inspectable etc; they are open.

There are clear technical improvements to make, but the real remaining challenge is to understand how an organization can make and maintain the required standards for this type of system while also surviving as an entity. There are some good examples in open source software of organizations (like Mozilla from 1.4.1) that have pulled this off, but software does not have incremental costs to manufacture. I do not think that it has to be a charity, in fact most of the modern world is engineered against standards that are maintained by private organizations [354]. It is not either that these standards do not exist in industrial control already: EtherCAT, ProfiNET, CANBus etc are all maintained in this way. But they are perhaps too slow.

In open software, standards emerge in a body of practice that is shared amongst a community of practitioners; maybe we need *guilds* for machine control who make and maintain a set of tools for their own use. On that topic, I found [355] to be extremely informative: the simple sharing of ideas and methods by practitioners (in this case, early steam engine designers) led to incredibly impactful improvements to performance. Those drove the *First* Industrial Revolution. Rather than focussing on technology itself, the people and the practice may be the place to start. They are enabled by explainable, inspectable technologies.

9 Conclusion

9.1 What is Possible Now that Wasn't Before?

In the current state-of-the-art, machine workflows are computationally feed-forward: we set parameters in CAM tools, they are compiled into GCodes, which the machine follows to the letter. Neither the CAM nor the machine have formalized knowledge of machine or process physics. This means that machine users must rely on tacit knowledge, intuition, or someone else's pre-determined settings to successfully use machines.

In this thesis I modified these workflows by adding models of key machine components: their motors, their kinematics, and their processes. Using models, we can build workflows that are fundamentally different in the state-of-the-art. Rather than feeding parameters forwards directly at a low level, we can describe *optimal outcomes* for our tasks, apply heuristics at a high level, and use models and model-based control to mediate between real-time physics and our targets. This amounts — partially — to machine systems that "know what they are doing," in that they use simulations of real world physics to choose their actions.

Models enable not just better machine control, but can help us to design machines by showing us hypothetical performance under different parameters. They can also help us to better design machine toolpaths by simulating those virtually before we test them in the real world.

Machines are heterogeneous: many types and models exist, and even individual instances of the same hardware will vary (and vary over time) in subtle ways. That is why a big focus in this thesis was to use *the machines themselves* to build models. This not only simplifies the alignment of controller expectations to reality, it also enables us to detect changes to machinery over time, watch for errors, and improve models over time.

None of this would have been possible with GCode based machine controllers. The work in Section 2.3, Section 2.4, and Section 2.5, each of which works to replace some aspect of the firmwares and controllers that run *underneath* GCode (and some layers above it) was a critical prerequisite. With those contributions, I also showed that we can develop this new family of controllers with a reusable set of hardware and software modules rather than re-writing firmware or re-designing circuits for each new process.

9.2 Feedback Systems Assembly

I want to return to an observation that I made in the introduction (1.1), which is that GCode based workflows are akin to compilers in computing; they rely on the regularization of the low levels of a system (and heuristics about those systems) to function, and they are fundamentally feed-forward. But the regularization of machine systems is an impossible task. There are many types of machines made of many different components. Those are built by a mixture of firms and individuals, each understands a part of the problem but not all of it.

If feed-forward doesn't work, we need feed-back systems assembly; not just for the use of these systems, but for their assembly and integration. In Chapter 7 I explained how models can help here. They provide representations of heterogeneous hardware that can be programmed against; from a systems viewpoint (Section 7.2.1) and also a physical one (Section 7.2.2). In building these models with data from systems themselves, we can ensure that the views we have of our systems as we build applications are accurate. That can eliminate the need for heuristics; we don't need to make assumptions about what will happen because we can inspect models to find out.

9.3 Closing Note: Imaginary Worlds

Like many people who are exposed to it, I always found digital fabrication exciting for the possibilities it opened up as a craftsperson and engineer. I fell down the rabbit hole when I started trying to build my own machines, which I did mostly because my time was free, I didn't have any money, and I wanted to have machines of my own.

I started the research arc that led to this thesis because every time I tried to modify these machines (to repurpose 3D printer firmware to make a five axis milling machine, or to use a machine interactively), I was stymied by obscure systems with strange representations, running firmware whose core principles were not explained anywhere.

When I came to MIT I found that even *here* the nuts and bolts of what makes a machine tick are not very well understood *by the people who use them.* However, almost everyone who spends much time with them comes to see the same frustrations. As Ilan Moyer would say, they do not *feel* like tools, but it seems like the aught to.

At the dawn of the First Industrial Revolution in 1745 Piranesi etched his *Imaginary Prisons,* one of which is above in Figure 9.1, depicting a dark future where humans are indentured to machines and systems of our own making. The future for digital fabrication is not *so* dire, but we seem to have failed in building machine systems that work *for* humans: they should be teaching us, and should enable us to willfully modify them. In their current configurations, machines for digital fabrication require serious engagement with occult knowledge. There has been excellent progress on this front recently towards consumer-facing equipment (mostly 3D printers), but we have yet to make *the development of new automation* straightforward. This seems urgent at a time

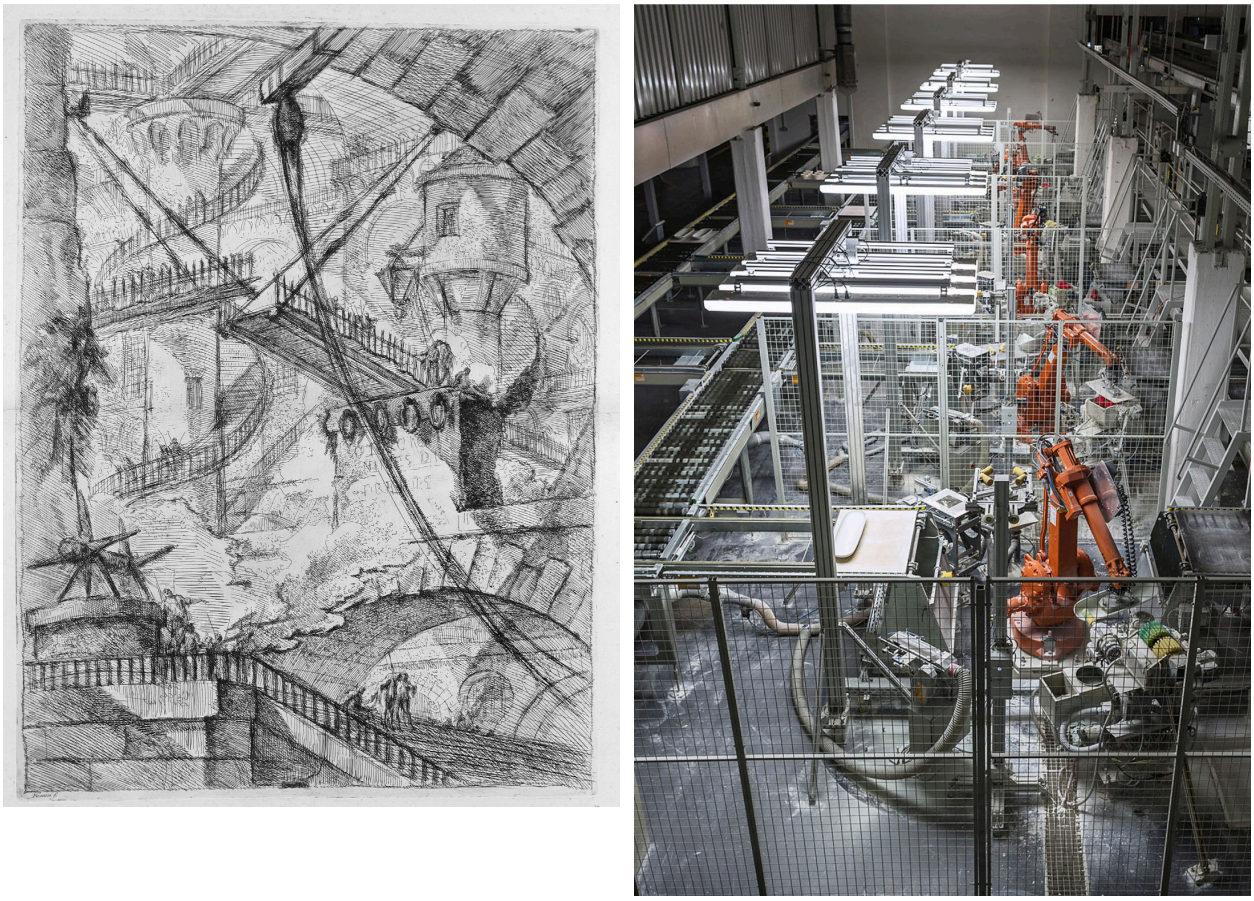


Figure 9.1: Piranesi's *Imaginary Prisons* (Plate 5) (left), and ABB Robots in cells (right).

when our economic, demographic and environmental concerns will require us to build lots of tools very quickly.

In this thesis, I tried overall to contribute where I could to make machine operation and development make more sense, trying to transform these mysterious systems using common place tools: everyday programming design patterns and physics that we each have some lived intuition for.

Bibliography

[1] M. Eberhardt and F. Teal, "Econometrics for grumblers: a new look at the literature on cross-country growth empirics," *Journal of Economic Surveys*, vol. 25, no. 1, pp. 109–155, 2011, doi: 10.1111/j.1467-6419.2010.00624.x.

[2] L. Hendricks and T. Schoellman, "Human capital and development accounting: New evidence from wage gains at migration," *The Quarterly Journal of Economics*, vol. 133, no. 2, pp. 665–700, 2018, doi: 10.1093/qje/qjx047.

[3] D. Acemoglu and D. Autor, "Skills, Tasks and Technologies: Implications for Employment and Earnings," *Handbook of Labor Economics*, vol. 4. Elsevier, pp. 1043–1171, 2011. doi: 10.1016/S0169-7218(11)02410-5.

[4] NPR Planet Money, "Why Aren't Americans Filling the Manufacturing Jobs We Already Have?." Accessed: Nov. 14, 2025. [Online]. Available: https://www.npr.org/sections/planet-money/2025/05/13/g-s1-66112/why-arent-americans-filling-the-manufacturing-jobs-we-already-have

[5] Bureau of Labor Statistics, "Manufacturing: NAICS 31-33." Accessed: Nov. 14, 2025. [Online]. Available: https://www.bls.gov/iag/tgs/iag31-33.htm

[6] Statistics Canada, "Job Vacancies, Payroll Employees, and Job Vacancy Rate by Industry Sector, Monthly, Adjusted for Seasonality." Accessed: Nov. 14, 2025. [Online]. Available: https://www150.statcan.gc.ca/t1/tbl1/en/tv.action?pid=1410040601

[7] M. Labrunie, C. Lopez-Gomez, and T. Minshall, "Manufacturing Remains a Core Driver of Economic Growth," *The Economist*, Jul. 2025, Accessed: Nov. 14, 2025. [Online]. Available: https://www.economist.com/letters/2025/07/03/manufacturing-remains-a-core-driver-of-economic-growth

[8] D. F. Noble, *Forces of Production: A Social History of Industrial Automation*. Routledge, 1984. doi: 10.4324/9780203791806.

[9] D. Acemoglu and P. Restrepo, "Automation and New Tasks: How Technology Displaces and Reinstates Labor," *Journal of Economic Perspectives*, vol. 33, no. 2, pp. 3–30, May 2019, doi: 10.1257/jep.33.2.3.

[10] B. Armstrong, V. K. Chen, A. Cuellar, A. Forsey-Smerek, and J. A. Shah, "Automation from the Worker's Perspective," *arXiv preprint arXiv:2409.20387*, 2024, [Online]. Available: https://arxiv.org/abs/2409.20387

[11] B. Armstrong and J. Shah, "A Smarter Strategy for Using Robots," *Harvard Business Review*, vol. 101, no. 3–4, pp. 35–42, 2023.

[12] S. Berger and B. Armstrong, "The Puzzle of the Missing Robots," *MIT Industrial Performance Center Working Paper*, 2022, doi: 10.21428/2c646de5.9461c9fb.

[13] B. Subbaraman, O. de Lange, S. Ferguson, and N. Peek, "The Duckbot: A System for Automated Imaging and Manipulation of Duckweed," *PLOS ONE*, vol. 19, no. 1, p. e296717, 2024, doi: 10.1371/journal.pone.0296717.

[14] M. Politi *et al.*, "A High-Throughput Workflow for the Synthesis of CdSe Nanocrystals Using a Sonochemical Materials Acceleration Platform," *Digital Discovery*, vol. 2, no. 4, pp. 1042–1057, 2023, doi: 10.1039/D3DD00033H.

[15] N. Gershenfeld, "How to Make Almost Anything: The Digital Fabrication Revolution," *Foreign Affairs*, vol. 91, no. 6, pp. 43–57, 2012.

[16] P. Blikstein, "Digital Fabrication and "Making" in Education: The Democratization of Invention," pp. 203–221, 2013.

[17] J. Jacobs and N. Peek, "Learning Remotely, Making Locally: Remote Digital Fabrication Instruction During a Pandemic," *Interactions*, vol. 27, no. 6, pp. 73–77, 2020, doi: 10.1145/3427444.

[18] J. Rutter and N. Gershenfeld, "Haystack Labs." [Online]. Available: https://www.haystack-mtn.org/haystack-labs

[19] G. Benabdallah and N. Peek, "Technical Mentality: Principles for HCI Research and Practice," in *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–14. doi: 10.1145/3613904.3642720.

[20] N. Peek, "Making Machines that Make: Object-Oriented Hardware Meets Object-Oriented Software," Ph.D. dissertation, 2016.

[21] C. Hess, E. Ostrom, and G. M. McCombs, "Understanding Knowledge as a Commons: From Theory to Practice," *College and Research Libraries*, vol. 69, no. 1, pp. 92–94, 2008, doi: 10.7551/mitpress/6980.001.0001.

[22] W. M. Pease and J. O. McDonough, "Final Report on Digitally Controlled Milling Machine," Cambridge, MA, technical report, 1952.

[23] Electronic Industries Association, "EIA Standard RS-274: Interchangeable Variable Block Data Format for Positioning, Contouring, and Contouring/Positioning Numerically Controlled Machines," Washington, D.C., Standard, 1963.

[24] D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972, doi: 10.1145/361598.361623.

[25] J. L. Dávila, B. M. Manzini, M. A. d'Ávila, and J. V. L. da Silva, "Open-Source Syringe Extrusion Head for Shear-Thinning Materials 3D Printing," *Rapid Prototyping Journal*, vol. 28, no. 8, pp. 1452–1461, Aug. 2022, doi: 10.1108/RPJ-09-2021-0245.

[26] N. A. Mendez and G. Corthey, "Pipettin Bot." [Online]. Available: https://gitlab.com/pipettin-bot/pipettin-bot

[27] P. Dettinger *et al.*, "Open-Source Personal Pipetting Robots with Live-Cell Incubation and Microscopy Compatibility," *Nature Communications*, vol. 13, no. 1, p. 2999, May 2022, doi: 10.1038/s41467-022-30643-7.

[28] D. Florian, "OTTO Liquid Handling Robot." [Online]. Available: https://openliquidhandler.com/

[29] A. M. Madni and M. Sievers, "Systems Integration: Key Perspectives, Experiences, and Challenges," *Systems Engineering*, vol. 17, no. 1, pp. 37–51, 2014, doi: 10.1002/sys.21249.

[30] N. Gershenfeld, "How to Make Almost Anything." [Online]. Available: https://fab.cba.mit.edu/classes/863.24/

[31] N. Gershenfeld, "Rapid-Prototyping of Rapid-Prototyping Machines: How to Make Something that Makes (almost) Anything." [Online]. Available: https://fab.cba.mit.edu/classes/865.24/index.html

[32] A. G. Stephenson *et al.*, "Mars Climate Orbiter Mishap Investigation Board Phase I Report," technical report, Nov. 1999. [Online]. Available: https://llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf

[33] J.-L. Lions, "ARIANE 5 Flight 501 Failure: Report by the Inquiry Board," Paris, France, technical report, Jul. 1996. [Online]. Available: https://esamultimedia.esa.int/docs/esa-x-1819eng.pdf

[34] N. Clark, "The Airbus Saga: Crossed Wires and a Multibillion-Euro Delay," *New York Times / International Herald Tribune*, Dec. 2006.

[35] M. Hobday, A. Davies, and A. Prencipe, "Systems Integration: A Core Capability of the Modern Corporation," *Industrial and Corporate Change*, vol. 14, no. 6, pp. 1109–1143, 2005, doi: 10.1093/icc/dth080.

[36] A. Prencipe, A. Davies, and M. Hobday, *The Business of Systems Integration*. Oxford University Press, 2003. doi: 10.1093/0199263221.001.0001.

[37] C. Y. Baldwin and K. B. Clark, *Design Rules, Volume 1: The Power of Modularity*. MIT Press, 2000. doi: 10.7551/mitpress/2366.001.0001.

[38] Y. Benkler, "Coase's Penguin, or, Linux and "The Nature of the Firm"," *Yale Law Journal*, vol. 112, no. 3, pp. 369–446, 2002, doi: 10.2307/1562247.

[39] N. Eghbal, *Working in Public: The Making and Maintenance of Open Source Software*. Stripe Press, 2020.

[40] B. W. Lampson, "Software Components: Only the Giants Survive," *Computer Systems: Theory, Technology, and Applications*. Springer, pp. 137–145, 2004. doi: 10.1007/0-387-21821-1_21.

[41] Y. Benkler, A. Shaw, and B. M. Hill, "Peer Production: A Form of Collective Intelligence," *Handbook of Collective Intelligence*, vol. 175, 2015.

[42] R. C. Allen, "Collective Invention," *Journal of Economic Behavior & Organization*, vol. 4, no. 1, pp. 1–24, 1983, doi: 10.1016/0167-2681(83)90023-9.

[43] B. Landau-Taylor and O. Dixon-Luinenburg, "Machine Tools: A Case Study in Advanced Manufacturing Industrial Economics," *Bismarck Analysis*, vol. 2, pp. 1–18, 2020, [Online]. Available: https://www.bismarckanalysis.com/Machine_Tools_Case_Study.pdf

[44] The Economist, "How Technology Is Redrawing the Boundaries of the Firm," *The Economist*, Jan. 2023, Accessed: Nov. 14, 2025. [Online]. Available: https://www.economist.com/business/2023/01/08/how-technology-is-redrawing-the-boundaries-of-the-firm

[45] O. E. Williamson, "Transaction Cost Economics: The Natural Progression," *American Economic Review*, vol. 100, no. 3, pp. 673–690, 2010, doi: 10.1257/aer.100.3.673.

[46] E. Ostrom, *Governing the Commons: The Evolution of Institutions for Collective Action*. Cambridge University Press, 1990. doi: 10.1017/cbo9780511807763.

[47] E. Ostrom, J. Walker, and R. Gardner, "Covenants With and Without a Sword: Self-Governance Is Possible," *American Political Science Review*, vol. 86, no. 2, pp. 404–417, 1992, doi: 10.2307/1964229.

[48] M. E. Conway, "How Do Committees Invent?," *Datamation*, vol. 14, no. 4, pp. 28–31, 1968.

[49] W. S. Lovejoy, "Rationalizing the Design Process," *Management of Design: Engineering and Management Perspectives*. Springer, pp. 233–253, 1994. doi: 10.1007/978-94-011-1390-8_13.

[50] T. J. Marion, M. H. Meyer, and G. Barczak, "The Influence of Digital Design and IT on Modular Product Architecture," *Journal of Product Innovation Management*, vol. 32, no. 1, pp. 98–110, 2015, doi: 10.1111/jpim.12240.

[51] R. P. Rumelt, "Inertia and Transformation," *Resource-Based and Evolutionary Theories of the Firm: Towards a Synthesis*. Springer, pp. 101–132, 1995. doi: 10.1007/978-1-4615-2201-0_5.

[52] A. MacCormack, J. Rusnak, and C. Y. Baldwin, "Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code," *Management Science*, vol. 52, no. 7, pp. 1015–1030, 2006, doi: 10.1287/mnsc.1060.0552.

[53] L. Lessig, *The Future of Ideas: The Fate of the Commons in a Connected World*. Vintage, 2002.

[54] I. Oshri, H. J. de Vries, and H. de Vries, "The Rise of Firefox in the Web Browser Industry: The Role of Open Source in Setting Standards," *Business History*, vol. 52, no. 5, pp. 834–856, 2010, doi: 10.1080/00076791.2010.499431.

[55] P. Gawthrop, I. Loram, M. Lakie, and H. Gollee, "Intermittent Control: A Computational Theory of Human Control," *Biological Cybernetics*, vol. 104, no. 1, pp. 31–51, 2011, doi: 10.1007/s00422-010-0416-4.

[56] J. Di Carlo, P. M. Wensing, B. Katz, G. Bledt, and S. Kim, "Dynamic Locomotion in the MIT Cheetah 3 Through Convex Model-Predictive Control," in *2018 IEEE/RSJ Interna-*

tional Conference on Intelligent Robots and Systems (IROS), 2018, pp. 1–9. doi: 10.1109/iros.2018.8594448.

[57] N. Gershenfeld, Q. Bolsée, and R. Hart, "Urumbu: Minimal Machine Building," in *Proceedings of the 7th Annual ACM Symposium on Computational Fabrication (SCF '22)*, Association for Computing Machinery, 2022, pp. 1–2. doi: 10.1145/3559400.3565599.

[58] K. O'Connor and Klipper Contributors, "Klipper: 3D Printer Firmware." [Online]. Available: https://github.com/Klipper3d/klipper

[59] Google Research, "JAX – The Sharp Bits." [Online]. Available: https://docs.jax.dev/en/latest/notebooks/Common_Gotchas_in_JAX.html

[60] F.-L. Lian, J. R. Moyne, and D. M. Tilbury, "Performance Evaluation of Control Networks: Ethernet, ControlNet, and DeviceNet," *IEEE Control Systems*, vol. 21, no. 1, pp. 66–83, Feb. 2001, doi: 10.1109/37.898793.

[61] International Organization for Standardization, "Information Technology—Open Systems Interconnection—Basic Reference Model: The Basic Model." Geneva, Switzerland, 1994. [Online]. Available: https://www.iso.org/standard/20269.html

[62] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-End Arguments in System Design," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 4, pp. 277–288, 1984, doi: 10.1145/357401.357402.

[63] ETSI Industry Specification Group, "Next Generation Protocols (NGP); An Example of a Non-IP Network Protocol Architecture Based on RINA Design Principles." [Online]. Available: https://www.etsi.org/deliver/etsi_gr/NGP/001_099/009/01.01.01_60/gr_ngp009v010101p.pdf

[64] Robert Bosch GmbH, "CAN Specification Version 2.0," Stuttgart, Germany, technical report, 1991. [Online]. Available: http://esd.cs.ucr.edu/webres/can20.pdf

[65] CAN in Automation (CiA), "CiA 301: CANopen Application Layer and Communication Profile," Nuremberg, Germany, technical report, 2011. [Online]. Available: https://www.can-cia.org/canopen/

[66] SAE International, "SAE J1939: Serial Control and Communications Heavy Duty Vehicle Network," Warrendale, PA, USA, technical report, 2024.

[67] D. Jansen and H. Buttner, "Real-time Ethernet: the EtherCAT solution," *Computing and Control Engineering*, vol. 15, no. 1, pp. 16–21, 2004, doi: 10.1049/cce:20040104.

[68] G. Prytz, "A Performance Analysis of EtherCAT and PROFINET IRT," in *2008 IEEE International Conference on Emerging Technologies and Factory Automation*, 2008, pp. 408–415. doi: 10.1109/etfa.2008.4638425.

[69] J. C. Eidson, M. Fischer, and J. White, "IEEE-1588 Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems," in *Proceedings*

of the 34th Annual Precise Time and Time Interval Systems and Applications Meeting, 2002, pp. 243–254.

[70] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer, 1997.

[71] I. E. Moyer, "A Gestalt Framework for Virtual Machine Control of Automated Tools," Master's thesis, 2013.

[72] N. Gershenfeld, R. Krikorian, and D. Cohen, "The Internet of Things," *Scientific American*, vol. 291, no. 4, pp. 76–81, 2004, doi: 10.1038/scientificamerican1004-76.

[73] N. Gershenfeld, "Asynchronous Packet Automata." 2012.

[74] N. Peek and N. Gershenfeld, "Mods: Browser-Based Rapid Prototyping Workflow Composition," in *Proceedings of the 38th Annual Conference of the Association for Computer Aided Design in Architecture (ACADIA)*, 2018, pp. 190–201. doi: 10.52842/conf.acadia.2018.066.

[75] T. Lock and D. Crocker, "Duet 3D Printer Controller Hardware." [Online]. Available: https://github.com/Duet3D/Duet-2-Hardware

[76] D. Crocker and Duet3D Contributors, "RepRapFirmware: OO C++ Firmware for CNC and 3D Printing." [Online]. Available: https://github.com/Duet3D/RepRapFirmware

[77] RepRap Community, "G-code." Accessed: May 15, 2026. [Online]. Available: https://reprap.org/wiki/G-code

[78] S. K. Jeon, S. S. Skogsrud, and J. Geisler, "Grbl: An Open Source, Embedded, High Performance G-code Parser and CNC Milling Controller." [Online]. Available: https://github.com/gnea/grbl

[79] L. Zhang, H. Gao, and O. Kaynak, "Network-Induced Constraints in Networked Control Systems—A Survey," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 403–416, 2012, doi: 10.1109/tii.2012.2219540.

[80] X.-M. Zhang *et al.*, "Networked Control Systems: A Survey of Trends and Techniques," *IEEE/CAA Journal of Automatica Sinica*, vol. 7, no. 1, pp. 1–17, 2019, doi: 10.1109/jas.2019.1911651.

[81] F.-L. Lian, J. Moyne, and D. Tilbury, "Network Design Consideration for Distributed Control Systems," *IEEE Transactions on Control Systems Technology*, vol. 10, no. 2, pp. 297–307, 2002, doi: 10.1109/87.987076.

[82] A. Iwainsky, E. Canuto, O. Taraszow, and A. Villa, "Network Decomposition for the Optimization of Connection Structures," *Networks*, vol. 16, no. 2, pp. 205–235, 1986, doi: 10.1002/net.3230160209.

[83] P. Daoutidis, W. Tang, and A. Allman, "Decomposition of Control and Optimization Problems by Network Structure: Concepts, Methods, and Inspirations from Biology," *AIChE Journal*, vol. 65, no. 10, 2019, doi: 10.1002/aic.16708.

[84] J. K. Yook, D. M. Tilbury, and N. R. Soparkar, "Trading Computation for Bandwidth: Reducing Communication in Distributed Control Systems Using State Estimators," *IEEE Transactions on Control Systems Technology*, vol. 10, no. 4, pp. 503–518, 2002, doi: 10.1109/tcst.2002.1014671.

[85] A. Alhammad and R. Pellizzoni, "Trading Cores for Memory Bandwidth in Real-Time Systems," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2016, pp. 1–11. doi: 10.1109/rtas.2016.7461364.

[86] S. Gupta and A. Lozano, "Computation-Bandwidth Trading for Mobile Edge Computing," in *2019 16th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, 2019, pp. 1–6. doi: 10.1109/ccnc.2019.8651745.

[87] M. Di Natale, "Scheduling the CAN Bus with Earliest Deadline Techniques," in *Proceedings 21st IEEE Real-Time Systems Symposium*, 2000, pp. 259–268. doi: 10.1109/real.2000.896015.

[88] D. L. Mills, "Internet Time Synchronization: The Network Time Protocol," *IEEE Transactions on Communications*, vol. 39, no. 10, pp. 1482–1493, 1991, doi: 10.1109/26.103043.

[89] A. Ciuffoletti, "Using Simple Diffusion to Synchronize the Clocks in a Distributed System," in *14th International Conference on Distributed Computing Systems*, Poznan, Poland: IEEE Comput. Soc. Press, 1994, pp. 484–491. doi: 10.1109/ICDCS.1994.302457.

[90] H. Kopetz and W. Ochsenreiter, "Clock Synchronization in Distributed Real-Time Systems," *IEEE Transactions on Computers*, no. 8, pp. 933–940, 1987, doi: 10.1109/tc.1987.5009516.

[91] M. Á. Ruiz-Sánchez, E. W. Biersack, and W. Dabbous, "Survey and Taxonomy of IP Address Lookup Algorithms," *IEEE Network*, vol. 15, no. 2, pp. 8–23, 2002, doi: 10.1109/65.912716.

[92] N. Yazdani and P. S. Min, "Fast and Scalable Schemes for the IP Address Lookup Problem," in *ATM 2000. Proceedings of the IEEE Conference on High Performance Switching and Routing*, 2000, pp. 83–92. doi: 10.1109/hpsr.2000.856650.

[93] S. Nilsson and G. Karlsson, "IP-Address Lookup Using LC-Tries," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 6, pp. 1083–1092, 2002, doi: 10.1109/49.772439.

[94] D. Pao, C. Liu, A. Wu, L. Yeung, and K. S. Chan, "Efficient Hardware Architecture for Fast IP Address Lookup," in *Proceedings of the Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2002, pp. 555–561. doi: 10.1109/infcom.2002.1019300.

[95] J. Aweya, "IP Router Architectures: An Overview," *International Journal of Communication Systems*, vol. 14, no. 5, pp. 447–475, 2001, doi: 10.1002/dac.505.

[96] Y.-S. Chu, P.-F. Lin, J.-H. Lin, H.-K. Su, and M.-J. Chen, "ASIC Design of Fast IP-Lookup for Next Generation IP Router," in *2005 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2005, pp. 3825–3828. doi: 10.1109/iscas.2005.1465464.

[97] X. Sun and Y. Q. Zhao, "An On-Chip IP Address Lookup Algorithm," *IEEE Transactions on Computers*, vol. 54, no. 7, pp. 873–885, 2005, doi: 10.1109/tc.2005.107.

[98] K. C. Lee and S. Lee, "Performance evaluation of switched Ethernet for real-time industrial communications," *Computer Standards & Interfaces*, vol. 24, no. 5, pp. 411–423, 2002, doi: https://doi.org/10.1016/S0920-5489(02)00070-3.

[99] C. A. Sunshine, "Source Routing in Computer Networks," *ACM SIGCOMM Computer Communication Review*, vol. 7, no. 1, pp. 29–33, 1977, doi: 10.1145/1024853.1024855.

[100] S. M. Parkes and P. Armbruster, "SpaceWire: A Spacecraft Onboard Network for Real-Time Communications," in *14th IEEE-NPSS Real Time Conference, 2005*, 2005, pp. 6–10. doi: 10.1109/RTC.2005.1547397.

[101] N. J. Boden *et al.*, "Myrinet: A Gigabit-per-Second Local Area Network," *IEEE Micro*, vol. 15, no. 1, pp. 29–36, 1995, doi: 10.1109/40.342015.

[102] D. Bertozzi and L. Benini, "Xpipes: A Network-on-Chip Architecture for Gigascale Systems-on-Chip," *IEEE Circuits and Systems Magazine*, vol. 4, no. 2, pp. 18–31, 2004, doi: 10.1109/MCAS.2004.1330747.

[103] K. Goossens, J. Dielissen, and A. Radulescu, "Æthereal Network on Chip: Concepts, Architectures, and Implementations," *IEEE Design & Test of Computers*, vol. 22, no. 5, pp. 414–421, 2005, doi: 10.1109/MDT.2005.99.

[104] M. Chiang, S. H. Low, A. R. Calderbank, and J. C. Doyle, "Layering as Optimization Decomposition: A Mathematical Theory of Network Architectures," *Proceedings of the IEEE*, vol. 95, no. 1, pp. 255–312, 2007, doi: 10.1109/jproc.2006.887322.

[105] R. Hartert *et al.*, "A Declarative and Expressive Approach to Control Forwarding Paths in Carrier-Grade Networks," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 15–28, 2015, doi: 10.1145/2829988.2787495.

[106] N. McKeown *et al.*, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008, doi: 10.1145/1355734.1355746.

[107] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking Control of the Enterprise," *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 4, pp. 1–12, 2007, doi: 10.1145/1282427.1282382.

[108] N. Gude *et al.*, "NOX: Towards an Operating System for Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008, doi: 10.1145/1384609.1384625.

[109] S. Jain *et al.*, "B4: Experience with a Globally-Deployed Software Defined WAN," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 3–14, 2013, doi: 10.1145/2534169.2486019.

[110] D. Firestone *et al.*, "Azure Accelerated Networking: SmartNICs in the Public Cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 51–66.

[111] D. Firestone, "VFP: A Virtual Switch Platform for Host SDN in the Public Cloud," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 315–328.

[112] A. Putnam *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," no. , pp. 13–24, 2014, doi: 10.1109/ISCA.2014.6853195.

[113] S. J. Oks *et al.*, "Cyber-Physical Systems in the Context of Industry 4.0: A Review, Categorization and Outlook," *Information Systems Frontiers*, vol. 26, no. 5, pp. 1731–1772, 2024, doi: 10.1007/s10796-022-10252-x.

[114] E. A. Lee, "The Past, Present and Future of Cyber-Physical Systems: A Focus on Models," *Sensors*, vol. 15, no. 3, pp. 4837–4869, 2015, doi: 10.3390/s150304837.

[115] A. Köcher, C. Hildebrandt, L. M. Vieira da Silva, and A. Fay, "A Formal Capability and Skill Model for Use in Plug and Produce Scenarios," in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2020, pp. 1663–1670. doi: 10.1109/ETFA46521.2020.9211874.

[116] H. Kern, F. Stefan, and V. Dimitrieski, "Intelligent and Self-Adapting Integration Between Machines and Information Systems," *IADIS International Journal on Computer Science & Information Systems*, vol. 10, no. 1, pp. 47–63, 2015.

[117] A. S. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms*, 2nd ed. Upper Saddle River, NJ: Pearson Prentice Hall, 2007.

[118] P. A. Bernstein, "Middleware: A Model for Distributed System Services," *Communications of the ACM*, vol. 39, no. 2, pp. 86–98, 1996, doi: 10.1145/230798.230809.

[119] ISO/IEC, "Message Queuing Telemetry Transport (MQTT) v3.1.1," ISO/IEC, 2016. doi: 10.3403/30328634.

[120] P. Hintjens, *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, 2013.

[121] G. Pardo-Castellote, "OMG Data-Distribution Service: Architectural Overview," in *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.*, 2003, pp. 200–206. doi: 10.1109/ICDCSW.2003.1203555.

[122] A. N. Lam and Ø. Haugen, "Implementing OPC-UA Services for Industrial Cyber-Physical Systems in Service-Oriented Architecture," in *IECON 2019 – 45th Annual Conference of the IEEE Industrial Electronics Society*, 2019, pp. 5486–5492. doi: 10.1109/iecon.2019.8926972.

[123] S. Dehnavi, D. Goswami, M. Koedam, A. Nelson, and K. Goossens, "Modeling, Implementation, and Analysis of XRCE-DDS Applications in Distributed Multi-Processor Real-Time Embedded Systems," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 1148–1151. doi: 10.23919/DATE51398.2021.9474221.

[124] K. Belsare *et al.*, "Micro-ROS," *Robot Operating System (ROS): The Complete Reference (Volume 7)*. Springer, pp. 3–55, 2023. doi: 10.1007/978-3-030-75472-3.

[125] A. Mamani-Saico and P. R. Yanyachi, "Implementation and Performance Study of the Micro-ROS/ROS2 Framework to Algorithm Design for Attitude Determination and Control System," *IEEE Access*, vol. 11, pp. 128451–128460, 2023, doi: 10.1109/ACCESS.2023.3330441.

[126] M. Müller, E. Wings, and L. Bergmann, "Developing Open Source Cyber-Physical Systems for Service-Oriented Architectures Using OPC UA," in *2017 IEEE 15th International Conference on Industrial Informatics (INDIN)*, 2017, pp. 83–88. doi: 10.1109/INDIN.2017.8104751.

[127] Z. Kang and A. Dubey, "Evaluating DDS, MQTT, and ZeroMQ Under Different IoT Traffic Conditions," in *M4IoT '20: Proceedings of the International Workshop on Middleware and Applications for the Internet of Things*, 2020, pp. 7–12.

[128] M. Quigley *et al.*, "ROS: An Open-Source Robot Operating System," in *ICRA Workshop on Open Source Software*, Kobe, Japan, 2009, p. 5.

[129] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the Performance of ROS2," in *Proceedings of the 13th International Conference on Embedded Software (EMSOFT '16)*, Pittsburgh, Pennsylvania: Association for Computing Machinery, 2016. doi: 10.1145/2968478.2968502.

[130] I. Malavolta, G. A. Lewis, B. Schmerl, P. Lago, and D. Garlan, "Mining Guidelines for Architecting Robotics Software," *Journal of Systems and Software*, vol. 178, p. 110969, 2021, doi: 10.1016/j.jss.2021.110969.

[131] S. Kolak, A. Afzal, C. Le Goues, M. Hilton, and C. S. Timperley, "It Takes a Village to Build a Robot: An Empirical Study of the ROS Ecosystem," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, pp. 430–440. doi: 10.1109/ICSME46990.2020.00048.

[132] P. Canelas, B. Schmerl, A. Fonseca, and C. S. Timperley, "Understanding Misconfigurations in ROS: An Empirical Study and Current Approaches," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1161–1173. doi: 10.1145/3650212.3680350.

[133] J. Cano, A. Bordallo, V. Nagarajan, S. Ramamoorthy, and S. Vijayakumar, "Automatic Configuration of ROS Applications for Near-Optimal Performance," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016, pp. 2217–2223. doi: 10.1109/iros.2016.7759347.

[134] C. S. Timperley, T. Dürschmid, B. Schmerl, D. Garlan, and C. Le Goues, "ROSDiscover: Statically Detecting Run-Time Architecture Misconfigurations in Robotics Systems," in *2022 IEEE 19th International Conference on Software Architecture (ICSA)*, 2022, pp. 112–123. doi: 10.1109/ICSA53651.2022.00019.

[135] S. Popić, D. Pezer, B. Mrazovac, and N. Teslić, "Performance Evaluation of Using Protocol Buffers in the Internet of Things Communication," in *2016 International Conference on Smart Systems and Technologies (SST)*, 2016, pp. 261–265. doi: 10.1109/SST.2016.7765670.

[136] C. Currier, "Protocol Buffers," in *Mobile Forensics – The File Format Handbook: Common File Formats and File Systems Used in Mobile Devices*, C. Hummert and D. Pawlaszczyk, Eds., Cham: Springer International Publishing, 2022, pp. 223–260. doi: 10.1007/978-3-030-98467-0_9.

[137] D. Raghavan, P. Levis, M. Zaharia, and I. Zhang, "Breakfast of Champions: Towards Zero-Copy Serialization with NIC Scatter-Gather," in *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*, 2021, pp. 199–205. doi: 10.1145/3458336.3465287.

[138] E. Maltsev and R. U. Amin, "Impact of Serialization Format on Inter-Service Latency," *ACPS*, vol. 9, no. 2, pp. 89–94, 2024, doi: 10.23939/acps2024.02.089.

[139] J. Kodosky, "LabVIEW," *Proceedings of the ACM on Programming Languages*, vol. 4, no. HOPL, pp. 1–54, 2020, doi: 10.1145/3386328.

[140] M. Lekić and G. Gardašević, "IoT Sensor Integration to Node-RED Platform," in *2018 17th International Symposium INFOTEH-JAHORINA (INFOTEH)*, 2018, pp. 1–5. doi: 10.1109/INFOTEH.2018.8345544.

[141] K. Ferencz and J. Domokos, "Using Node-RED Platform in an Industrial Environment," *XXXV. Jubileumi Kandó Konferencia, Budapest*, pp. 52–63, 2019.

[142] J. Nordby, "Building Distributed Systems with MsgFlo: Flow-Based-Programming over Message Queues." [Online]. Available: https://archive.fosdem.org/2017/schedule/event/iot_msgflo/

[143] S. Vinoski, "Advanced Message Queuing Protocol," *IEEE Internet Computing*, vol. 10, no. 6, pp. 87–89, 2006, doi: 10.1109/MIC.2006.116.

[144] R. T. Fielding, "Architectural Styles and the Design of Network-Based Software Architectures," Ph.D. dissertation, 2000.

[145] R. T. Fielding and R. N. Taylor, "Principled Design of the Modern Web Architecture," *ACM Transactions on Internet Technology (TOIT)*, vol. 2, no. 2, pp. 115–150, 2002, doi: 10.1145/514183.514185.

[146] G. E. Krasner, S. T. Pope, and others, "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System," *Journal of Object Oriented Programming*, vol. 1, no. 3, pp. 26–49, 1988.

[147] A. Leff and J. T. Rayfield, "Web-Application Development Using the Model/View/Controller Design Pattern," in *Proceedings Fifth IEEE International Enterprise Distributed Object Computing Conference*, 2001, pp. 118–127. doi: 10.1109/EDOC.2001.950428.

[148] J. Thönes, "Microservices," *IEEE Software*, vol. 32, no. 1, p. 116, 2015, doi: 10.1109/MS.2015.11.

[149] Prusa Research a.s., "Prusa CORE One: Fully Enclosed CoreXY 3D Printer." [Online]. Available: https://blog.prusa3d.com/introducing-prusa-core-one-fully-enclosed-corexy-3d-printer-with-active-temperature-control_105477/

[150] K. O'Connor and Klipper Contributors, "Klipper: Benchmarks." [Online]. Available: https://www.klipper3d.org/Benchmarks.html

[151] I. E. Moyer, D. Frost, E. Yu, M. Yang, and J. Jacobs, "StepDance: A Toolkit for Redesigning CNC Machines Using Physical Metaphors," in *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems*, 2026, pp. 1–34. doi: 10.1145/3772318.3791107.

[152] I. E. Moyer, S. Bourgault, D. Frost, and J. Jacobs, "Throwing Out Conventions: Reimagining Craft-Centered CNC Tool Design Through the Digital Pottery Wheel," in *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–22. doi: 10.1145/3613904.3642361.

[153] A. Ng, M. Annett, P. Dietz, A. Gupta, and W. F. Bischof, "In the blink of an eye: Investigating latency perception during stylus interaction," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2014, pp. 1103–1112. doi: 10.1145/2556288.2557037.

[154] W. R. Ferrell, "Remote manipulation with transmission delay," *IEEE Transactions on Human Factors in Electronics*, no. 1, pp. 24–32, 1965, doi: 10.1109/THFE.1965.6591253.

[155] F. Fossdal, R. Heldal, and N. Peek, "Interactive Digital Fabrication Machine Control Directly Within a CAD Environment," in *Proceedings of the 6th Annual ACM Symposium on Computational Fabrication (SCF '21)*, 2021, pp. 1–15. doi: 10.1145/3485114.3485120.

[156] F. H. Fossdal, V. Nguyen, R. Heldal, C. L. Cobb, and N. Peek, "Vespidae: A Programming Framework for Developing Digital Fabrication Workflows," in *Proceedings of the 2023 ACM Designing Interactive Systems Conference (DIS '23)*, 2023, pp. 2034–2049. doi: 10.1145/3563657.3596106.

[157] H. Twigg-Smith and N. Peek, "Dynamic Toolchains: Software Infrastructure for Digital Fabrication Workflows," in *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (UIST '23)*, 2023, pp. 1–20. doi: 10.1145/3586183.3606802.

[158] J. Tran O'Leary, G. Benabdallah, and N. Peek, "Imprimer: Computational Notebooks for CNC Milling," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, 2023, pp. 1–15. doi: 10.1145/3544548.3581334.

[159] J. Vasquez, H. Twigg-Smith, J. Tran O'Leary, and N. Peek, "Jubilee: An Extensible Machine for Multi-Tool Fabrication," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020, pp. 1–13. doi: 10.1145/3313831.3376425.

[160] K. Dunn, C. Feng, and N. Peek, "Jubilee: A Case Study of Distributed Manufacturing in an Open Source Hardware Project," *Journal of Open Hardware*, vol. 7, no. 1, p. 4, 2023, doi: 10.5334/joh.51.

[161] N. Peek and L. Pozzo, "Pathways to Open-Source Hardware for Laboratory Automation." [Online]. Available: https://depts.washington.edu/machines/scienceautomation/

[162] S. Propst and J. Mueller, "Time Code for Multifunctional 3D Printhead Controls," *Nature Communications*, vol. 16, no. 1, p. 1035, 2025, doi: 10.1038/s41467-025-56140-1.

[163] Z. Zhao *et al.*, "Dynamic Speed Adjustment Based on Time-Code Enhances Filament Consistency in Continuous Switching 3D Food Printing," *Journal of Food Engineering*, p. 112973, 2026, doi: 10.1016/j.jfoodeng.2026.112973.

[164] J. R. Read, N. Peek, and N. Gershenfeld, "MAXL: Distributed Trajectories for Modular Motion," in *Proceedings of the 8th ACM Symposium on Computational Fabrication (SCF '23)*, Association for Computing Machinery, 2023, pp. 1–11. doi: 10.1145/3623263.3623362.

[165] A. Gleadall, "FullControl GCode Designer: Open-Source Software for Unconstrained Design in Additive Manufacturing," *Additive Manufacturing*, vol. 46, p. 102109, 2021, doi: 10.1016/j.addma.2021.102109.

[166] B. Subbaraman and N. Peek, "p5.fab: Direct Control of Digital Fabrication Machines from a Creative Coding Environment," in *Proceedings of the 2022 ACM Designing Interactive Systems Conference (DIS '22)*, 2022, pp. 1148–1161. doi: 10.1145/3532106.3533496.

[167] J. R. Read, "Distributed Dataflow Machine Controllers," S.M. Thesis, 2020. [Online]. Available: https://dspace.mit.edu/handle/1721.1/145214

[168] S. Cheshire and M. Baker, "Consistent overhead byte stuffing," *SIGCOMM Comput. Commun. Rev.*, vol. 27, no. 4, pp. 209–220, Oct. 1997, doi: 10.1145/263109.263168.

[169] R. N. Williams, "A Painless Guide to CRC Error Detection Algorithms," Adelaide, Australia, Technical Report, Aug. 1993.

[170] International Telecommunication Union, "Code-Independent Error-Control System," Recommendation V.41, 1988.

[171] S. Parkes, "SpaceWire for Adaptive Systems," in *2008 NASA/ESA Conference on Adaptive Hardware and Systems*, 2008, pp. 77–82. doi: 10.1109/ahs.2008.30.

[172] D. Rutten, "Grasshopper: Visual Programming for Rhinoceros 3D." [Online]. Available: https://www.grasshopper3d.com/

[173] A. Ronacher, "Jinja: A Fast, Expressive, Extensible Templating Engine for Python." [Online]. Available: https://jinja.palletsprojects.com/

[174] F. Holmer, "The Continuity of Splines." [Online]. Available: https://www.youtube.com/watch?v=jvPPXbo87ds

[175] P. Virtanen *et al.*, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, no. 3, pp. 261–272, 2020, doi: 10.1038/s41592-019-0686-2.

[176] I. Moyer, "CoreXY: A Mechanism for Motion Control in 3D Printers and CNC Machines." [Online]. Available: https://corexy.com/

[177] M. Henning, "API Design Matters: Why Changing APIs Might Become a Criminal Offense," *Queue*, vol. 5, no. 4, pp. 24–36, 2007, doi: 10.1145/1255421.1255422.

[178] J. Bloch, "How to Design a Good API and Why It Matters," in *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '06)*, 2006, pp. 506–507. doi: 10.1145/1176617.1176622.

[179] K. Varda, "Cap'n Proto: Insanely Fast Data Interchange Format and Capability-Based RPC System." 2013.

[180] F. Gündling, "Cista++: Simple, High-Performance, Zero-Copy C++ Serialization & Reflection." 2018.

[181] D. Tolnay and E. Tryzelaar, "Serde: A Generic Serialization/Deserialization Framework for Rust." 2015.

[182] H. Abelson, G. J. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs*, Second. MIT Press, 1996. doi: 10.5555/547755.

[183] G. Rakow, R. Schnurr, C. Dailey, and K. Shakoorzadeh, "Reliable Transport over SpaceWire for James Webb Space Telescope (JWST) Focal Plane Electronics (FPE) Network," in *2003 IEEE Aerospace Conference*, 2003. doi: 10.1109/aero.2003.1235152.

[184] S. Haas, "The IEEE 1355 Standard. Developments, performance and application in high energy physics," Doctoral dissertation, 1998. [Online]. Available: https://repository.cern/records/dpx9m-j4r46/files/cer-002474543.pdf

[185] C. Whitby-Strevens, "Transputers-past, present and future," *IEEE Micro*, vol. 10, no. 6, pp. 16–19, 1990, doi: 10.1109/40.62725.

[186] S. Parkes, C. McClements, D. McLaren, A. F. Florit, and A. G. Villafranca, "SpaceFibre: A multi-Gigabit/s interconnect for spacecraft onboard data handling," in *2015 IEEE Aerospace Conference*, 2015, pp. 1–13. doi: 10.1109/AERO.2015.7119317.

[187] P. Mohapatra, "Wormhole routing techniques for directly connected multicomputer systems," *ACM Comput. Surv.*, vol. 30, no. 3, pp. 374–410, Sep. 1998, doi: 10.1145/292469.292472.

[188] H. Flatt, J. Jasperneite, and F. Schewe, "An FPGA based cut-through switch optimized for one-step PTP and real-time Ethernet," in *2013 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication (ISPCS) Proceedings*, 2013, pp. 7–12. doi: 10.1109/ISPCS.2013.6644755.

[189] T. Yuasa *et al.*, "SpaceWire/RMAP-Based Data Acquisition Framework for Scientific Instruments: Overview, Application and Recent Updates," in *Proceedings of the International SpaceWire Conference*, 2010.

[190] P. Mendham, S. Mills, and S. Parkes, "Network Management and Configuration Using RMAP," in *International SpaceWire Conference, Dundee*, 2007, pp. 119–129.

[191] R. Ward *et al.*, "Machining Digital Twin Using Real-Time Model-Based Simulations and Lookahead Function for Closed Loop Machining Control," *The International Journal of Advanced Manufacturing Technology*, vol. 117, no. 11, pp. 3615–3629, 2021, doi: 10.1007/s00170-021-07867-w.

[192] X. Tong, Q. Liu, Y. Zhou, and P. Sun, "A Digital Twin-Driven Cutting Force Adaptive Control Approach for Milling Process," *Journal of Intelligent Manufacturing*, vol. 36, no. 1, pp. 551–568, 2025, doi: 10.1007/s10845-023-02193-2.

[193] X. Tong, Q. Liu, S. Pi, and Y. Xiao, "Real-Time Machining Data Application and Service Based on IMT Digital Twin," *Journal of Intelligent Manufacturing*, vol. 31, no. 5, pp. 1113–1132, 2020, doi: 10.1007/s10845-019-01500-0.

[194] "TwinCAT ADS/AMS Specification." [Online]. Available: https://infosys.beckhoff.com/content/1033/tcadscommon/12440276875.html

[195] A. Wolf, J. Morris, and Smoothieware Contributors, "Smoothieware: Modular, Open Source, High Performance G-code Interpreter and CNC Controller." [Online]. Available: https://github.com/Smoothieware/Smoothieware

[196] J. R. Read, "The Blair Winch Project." [Online]. Available: https://ekswhyzee.com/2025/06/09/blair-winch-project.html

[197] J. Rutter and N. Gershenfeld, "Haystack Labs." [Online]. Available: https://www.haystack-mtn.org/haystack-labs

[198] R. Bush and D. Meyer, "RFC3439: Some Internet Architectural Guidelines and Philosophy." RFC Editor, USA, 2002. doi: 10.17487/RFC3439.

[199] A. Harrison and P. Newman, "TICSync: Knowing When Things Happened," in *2011 IEEE International Conference on Robotics and Automation*, 2011, pp. 356–363. doi: 10.1109/icra.2011.5980112.

[200] L. Schenato and G. Gamba, "A Distributed Consensus Protocol for Clock Synchronization in Wireless Sensor Network," in *2007 46th IEEE Conference on Decision and Control*, 2007, pp. 2289–2294. doi: 10.1109/CDC.2007.4434671.

[201] P. Horowitz, W. Hill, and V. Elings, *The Art of Electronics*. Cambridge University Press, 1981.

[202] R. Melville, "Resources: Resurrecting the "Dead Bug" Method: An Unconventional Approach to Prototyping High-Frequency Circuits," *IEEE Spectrum*, vol. 55, no. 10, pp. 16–17, 2018, doi: 10.1109/mspec.2018.8482414.

[203] J. R. Read, L. McElroy, Q. Bolsée, B. Smith, and N. Gershenfeld, "Modular-Things: Plug-and-Play with Virtualized Hardware," in *Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems (CHI EA '23)*, Association for Computing Machinery, 2023, pp. 1–6. doi: 10.1145/3544549.3585642.

[204] JEDEC Solid State Technology Association, "Compression Attached Memory Module (CAMM2) Common Standard," Arlington, VA, technical report JESD318B, 2024. [Online]. Available: https://www.jedec.org/standards-documents/docs/jesd318b

[205] JEDEC Solid State Technology Association, "LP5CAMM2 Connector Performance Standard," Arlington, VA, technical report PS-7A, Oct. 2024. [Online]. Available: https://www.jedec.org/standards-documents/docs/ps-007a

[206] IEEE, "IEEE Standard for Ethernet." IEEE, 2022.

[207] RMII Consortium, "Reduced Media Independent Interface (RMII) Specification," technical report, Mar. 1998.

[208] SFF Committee, "SFP (Small Formfactor Pluggable) Transceiver MultiSource Agreement," technical report INF-8074i, 2001.

[209] USB Implementers Forum, "USB Power Delivery Specification," technical report, 2021. [Online]. Available: https://www.usb.org/document-library/usb-power-delivery

[210] B. Douglas, "PID Control – A Brief Introduction." [Online]. Available: https://www.youtube.com/watch?v=UR0hOmjaHp0

[211] S. Abrate, "Vibrations of Belts and Belt Drives," *Mechanism and Machine Theory*, vol. 27, no. 6, pp. 645–659, 1992, doi: 10.1016/0094-114X(92)90064-O.

[212] M. Callegari, F. Cannella, and G. Ferri, "Multi-Body Modelling of Timing Belt Dynamics," *Proceedings of the Institution of Mechanical Engineers, Part K: Journal of Multi-body Dynamics*, vol. 217, no. 1, pp. 63–75, 2003, doi: 10.1243/146441903763049450.

[213] J. Moon and J. A. Wickert, "Non-Linear Vibration of Power Transmission Belts," *Journal of Sound and Vibration*, vol. 200, no. 4, pp. 419–431, 1997, doi: 10.1006/jsvi.1996.0709.

[214] A. Nabhan, M. R. El-Sharkawy, and A. Rashed, "Monitoring of Belt-Drive Defects Using the Vibration Signals and Simulation Models," *International Journal of Aerospace and Mechanical Engineering*, vol. 13, no. 5, pp. 332–339, 2019.

[215] D. A. Vicente, R. L. Hecker, F. J. Villegas, and G. M. Flores, "Modeling and Vibration Mode Analysis of a Ball Screw Drive," *The International Journal of Advanced Manufacturing Technology*, vol. 58, no. 1, pp. 257–265, 2012, doi: 10.1007/s00170-011-3375-6.

[216] Q. Wu, F. Gu, A. Ball, and H. Huang, "Hybrid Model for the Analysis of the Modal Properties of a Ball Screw Vibration System," *Journal of Mechanical Science and Technology*, vol. 35, no. 2, pp. 461–470, 2021, doi: 10.1007/s12206-021-0104-4.

[217] W. G. Lee, J. W. Lee, M. S. Hong, S.-H. Nam, Y. Jeon, and M. G. Lee, "Failure Diagnosis System for a Ball-Screw by Using Vibration Signals," *Shock and Vibration*, vol. 2015, no. 1, p. 435870, 2015, doi: 10.1155/2015/435870.

[218] L. Cao and H. M. Schwartz, "Oscillation, Instability and Control of Stepper Motors," *Nonlinear Dynamics*, vol. 18, no. 4, pp. 383–404, 1999, doi: 10.1023/a:1026413118091.

[219] J. P. Chen, W. Cheng, and W. Han, "Analysis and Simulation of Stepper Motor Disturbance Considering Structural Coupling," *Applied Mechanics and Materials*, vol. 526, pp. 103–108, 2014, doi: 10.4028/www.scientific.net/amm.526.103.

[220] L. Dosiek and P. Pillay, "Cogging Torque Reduction in Permanent Magnet Machines," *IEEE Transactions on Industry Applications*, vol. 43, no. 6, pp. 1565–1571, 2007, doi: 10.1109/icelmach.2016.7732526.

[221] X. Ge, Z. Q. Zhu, G. Kemp, D. Moule, and C. Williams, "Optimal Step-Skew Methods for Cogging Torque Reduction Accounting for Three-Dimensional Effect of Interior Permanent Magnet Machines," *IEEE Transactions on Energy Conversion*, vol. 32, no. 1, pp. 222–232, 2016, doi: 10.1109/tec.2016.2620476.

[222] T. T. Saito, "Diamond Turning of Optics," *Optical Engineering*, vol. 15, no. 5, pp. 431–434, 1976, doi: 10.1117/12.7972015.

[223] H. K. McCue, "The Motion Control System for the Large Optics Diamond Turning Machine (LODTM)," in *Contemporary Methods of Optical Manufacturing and Testing*, 1983, pp. 68–75. doi: 10.1117/12.936791.

[224] C. Brecher and C. Wenzel, "Ultraprecision and Micro Machine Tools for Micro Cutting," *Micro-Cutting: Fundamentals and Applications*. Wiley, pp. 63–85, 2013. doi: 10.1002/9781118536605.ch4.

[225] C. E. Groppi, B. Love, M. Underhill, and C. Walker, "Automated CNC Micromachining for Integrated THz Waveguide Circuits," in *Proceedings of the 21st International Symposium on Space Terahertz Technology*, 2010.

[226] I. A. Robinson and S. Schlamminger, "The Watt or Kibble Balance: A Technique for Implementing the New SI Definition of the Unit of Mass," *Metrologia*, vol. 53, no. 5, pp. A46–A74, 2016, doi: 10.1088/0026-1394/53/5/a46.

[227] J. S. Albus, A. J. Barbera, R. N. Nagel, and others, *Theory and Practice of Hierarchical Control*. Gaithersburg, MD, USA: National Bureau of Standards, 1980.

[228] D. J. Hyun, S. Seok, J. Lee, and S. Kim, "High Speed Trot-Running: Implementation of a Hierarchical Controller Using Proprioceptive Impedance Control on the MIT Cheetah," *The International Journal of Robotics Research*, vol. 33, no. 11, pp. 1417–1445, 2014, doi: 10.1177/0278364914532150.

[229] A. SaLoutos, H. Kim, E. Stanger-Jones, M. Guo, and S. Kim, "Towards Robust Autonomous Grasping with Reflexes Using High-Bandwidth Sensing and Actuation," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 2023, pp. 10254–10260. doi: 10.1109/ICRA48891.2023.10160930.

[230] K. M. Lynch and F. C. Park, *Modern Robotics: Mechanics, Planning, and Control*. Cambridge University Press, 2017. doi: 10.1017/9781316661239.

[231] S. K. Jeon, “Improving Grbl: Cornering Algorithm.” [Online]. Available: https://onehossshay.wordpress.com/2011/09/24/improving_grbl_cornering_algorithm/

[232] P. Conroy, “Computing Junction Deviation for Marlin Firmware.” [Online]. Available: http://blog.kyneticcnc.com/2018/10/computing-junction-deviation-for-marlin.html

[233] R. Tedrake, “Dynamic Programming,” *Underactuated Robotics: Algorithms for Walking, Running, Swimming, Flying, and Manipulation*. 2024. Accessed: Nov. 14, 2025. [Online]. Available: https://underactuated.mit.edu/dp.html

[234] J. Hong and X. Tan, “Method of Curvature Controlled Data Smoothing.” [Online]. Available: https://patents.google.com/patent/US20070085850A1/en

[235] S. Lavernhe, C. Tournier, and C. Lartigue, “Optimization of 5-Axis High-Speed Machining Using a Surface Based Approach,” *Computer-Aided Design*, vol. 40, no. 10–11, pp. 1015–1023, 2008, doi: 10.1016/j.cad.2008.08.006.

[236] Omron, “Delta Tau Power PMAC User's Manual.” 2023. [Online]. Available: https://files.omron.eu/downloads/latest/manual/en/o014_power_pmac_users_manual_en.pdf?v=1

[237] R. Ward, “Smooth Trajectory Generation for 5-Axis CNC Machine Tools,” EngD thesis, 2022.

[238] S. Tajima and B. Sencer, “Accurate Real-Time Interpolation of 5-Axis Tool-Paths with Local Corner Smoothing,” *International Journal of Machine Tools and Manufacture*, vol. 142, pp. 1–15, 2019, doi: 10.1016/j.ijmachtools.2019.04.005.

[239] D.-N. Song, J.-W. Ma, Y.-G. Zhong, D. Xiao, J.-J. Yao, and C. Zhou, “A Fully Real-Time Spline Interpolation Algorithm with Axial Jerk Constraint Based on FIR Filtering,” *The International Journal of Advanced Manufacturing Technology*, vol. 113, no. 7, pp. 1873–1886, 2021, doi: 10.1007/s00170-021-06738-8.

[240] W. Singhose and L. Pao, “A Comparison of Input Shaping and Time-Optimal Flexible-Body Control,” *Control Engineering Practice*, vol. 5, no. 4, pp. 459–467, 1997, doi: 10.1016/S0967-0661(97)00025-7.

[241] S. D. Jones and A. G. Ulsoy, “An Approach to Control Input Shaping with Application to Coordinate Measuring Machines,” *Journal of Dynamic Systems, Measurement, and Control*, vol. 121, no. 2, pp. 242–247, 1999, doi: 10.1115/1.2802461.

[242] M. O. T. Cole and T. Wongratanaphisan, “A Direct Method of Adaptive FIR Input Shaping for Motion Control with Zero Residual Vibration,” *IEEE/ASME Transactions on Mechatronics*, vol. 18, no. 1, pp. 316–327, 2011, doi: 10.1109/TMECH.2011.2174373.

[243] L. Biagiotti and C. Melchiorri, “FIR Filters for Online Trajectory Planning with Time- and Frequency-Domain Specifications,” *Control Engineering Practice*, vol. 20, no. 12, pp. 1385–1399, 2012, doi: 10.1016/j.conengprac.2012.08.005.

[244] L. Biagiotti and C. Melchiorri, "Trajectory Generation via FIR Filters: A Procedure for Time-Optimization Under Kinematic and Frequency Constraints," *Control Engineering Practice*, vol. 87, pp. 43–58, 2019, doi: 10.1016/j.conengprac.2019.03.017.

[245] P. Besset and R. Béarée, "FIR Filter-Based Online Jerk-Constrained Trajectory Generation," *Control Engineering Practice*, vol. 66, pp. 169–180, 2017, doi: 10.1016/j.conengprac.2017.06.015.

[246] S. Tajima and B. Sencer, "Online Interpolation of 5-Axis Machining Toolpaths with Global Blending," *International Journal of Machine Tools and Manufacture*, vol. 175, p. 103862, 2022, doi: 10.1016/j.ijmachtools.2022.103862.

[247] R. Ward, B. Sencer, G. Panoutsos, and E. Ozturk, "On-The-Fly CNC Interpolation Using Frequency-Domain FFT-Based Filtering," *Procedia CIRP*, vol. 107, pp. 1571–1576, 2022, doi: 10.1016/j.procir.2022.05.193.

[248] Heidenhain, "Dynamic Precision – Machining Dynamically and with High Accuracy," technical report ID1081196, 2013. [Online]. Available: https://www.heidenhain.de/fileadmin/pdf/en/01_Products/Technische_Dokumentation/TI_Dynamic_Precision_ID1081196_en.pdf

[249] Heidenhain, "TNC 640 HSCI OEM Information," technical report, 2017.

[250] Siemens AG, "SINUMERIK ONE: The First Digital-Native CNC System." [Online]. Available: https://www.siemens.com/us/en/products/automation/systems/sinumerik-cnc/machine-tools/sinumerik-one.html

[251] H. Fraunhofer, "Method for Adaptive Feed Rate Regulation on Numerically Controlled Machine Tools." [Online]. Available: https://patents.google.com/patent/US7346423B2/en

[252] J. Jeppsson, "Adaptive Feed Rate Override System for a Milling Machine." [Online]. Available: https://patents.google.com/patent/US4698773A/en

[253] F. Kohler, "Method and Device for Controlling a Milling Machine." Feb. 2018.

[254] N. Kerner, "Verfahren zur Parametrierung einer Dämpfung von Ratterschwingungen im Regelkreis einer Numerischen Steuerung." Sep. 2023.

[255] M. E. Moran, "Evolution of Robotic Arms," *Journal of Robotic Surgery*, vol. 1, no. 2, pp. 103–111, 2007, doi: 10.1007/s11701-006-0002-x.

[256] S. Kuindersma *et al.*, "Optimization-Based Locomotion Planning, Estimation, and Control Design for the Atlas Humanoid Robot," *Autonomous Robots*, vol. 40, pp. 429–455, 2016, doi: 10.1007/s10514-015-9479-3.

[257] G. Torrente, E. Kaufmann, P. Föhn, and D. Scaramuzza, "Data-Driven MPC for Quadrotors," *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 3769–3776, 2021, doi: 10.1109/lra.2021.3061307.

[258] R. Tedrake, "Trajectory Optimization," *Underactuated Robotics: Algorithms for Walking, Running, Swimming, Flying, and Manipulation*. 2024. Accessed: Nov. 14, 2025. [Online]. Available: https://underactuated.mit.edu/trajopt.html

[259] S. L. Brunton and J. N. Kutz, *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control*, 2nd ed. Cambridge, United Kingdom: Cambridge University Press, 2022. doi: 10.1017/9781009089517.

[260] J. A. E. Andersson, J. Gillis, G. Horn, J. B. Rawlings, and M. Diehl, "CasADi – A Software Framework for Nonlinear Optimization and Optimal Control," *Mathematical Programming Computation*, vol. 11, no. 1, pp. 1–36, 2019, doi: https://doi.org/10.1007/s12532-018-0139-4.

[261] R. Verschueren *et al.*, "acados – A Modular Open-Source Framework for Fast Embedded Optimal Control," *Mathematical Programming Computation*, vol. 14, no. 1, pp. 147–183, 2022, doi: https://doi.org/10.1007/s12532-021-00208-8.

[262] R. A. Bartlett, L. T. Biegler, J. Backström, and V. Gopal, "Quadratic Programming Algorithms for Large-Scale Model Predictive Control," *Journal of Process Control*, vol. 12, no. 7, pp. 775–795, 2002, doi: https://doi.org/10.1016/s0959-1524(02)00002-1.

[263] D. Rowell, "State-Space Representation of LTI Systems." [Online]. Available: https://web.mit.edu/2.14/www/Handouts/StateSpace.pdf

[264] Wikipedia contributors, "State-Space Representation." [Online]. Available: https://en.wikipedia.org/wiki/State-space_representation

[265] Google Research, "JAX: Autograd and XLA for High-Performance Machine Learning Research." [Online]. Available: https://github.com/google/jax

[266] DeepMind, "Optax: Composable Gradient Processing and Optimization Library for JAX." [Online]. Available: https://github.com/deepmind/optax

[267] E. Kaufmann, L. Bauersfeld, A. Loquercio, M. Müller, V. Koltun, and D. Scaramuzza, "Champion-Level Drone Racing Using Deep Reinforcement Learning," *Nature*, vol. 620, no. 7976, pp. 982–987, 2023, doi: 10.1038/s41586-023-06419-4.

[268] J. Y. Luo, Y. Song, V. Klemm, F. Shi, D. Scaramuzza, and M. Hutter, "Residual policy learning for perceptive quadruped control using differentiable simulation," in *2025 IEEE International Conference on Robotics and Automation (ICRA)*, 2025, pp. 1–8. doi: 10.1109/ICRA55743.2025.11127448.

[269] I. M. Banks, *Surface Detail*. in Culture novel. Orbit, 2011. [Online]. Available: https://books.google.com/books?id=2e4ecAAACAAJ

[270] M. Ay, M. Schwenzer, S. Stemmler, A. K. Rüppel, D. Abel, and T. Bergs, "Practical Nonlinear Model Predictive Control of a CNC Machining Center with Support Vector Machines," in *2022 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM)*, IEEE, 2022, pp. 1625–1631. doi: 10.1109/AIM52237.2022.9863354.

[271] P. Bosetti, M. Leonesio, and P. Parenti, "On Development of an Optimal Control System for Real-Time Process Optimization on Milling Machine Tools," *Procedia CIRP*, vol. 12, pp. 31–36, 2013, doi: 10.1016/j.procir.2013.09.007.

[272] P. Wu, K. S. Ramani, and C. E. Okwudire, "Accurate Linear and Nonlinear Model-Based Feedforward Deposition Control for Material Extrusion Additive Manufacturing," *Additive Manufacturing*, vol. 48, p. 102389, 2021, doi: 10.1016/j.addma.2021.102389.

[273] P. Wu, "Modeling and Feedforward Deposition Control in Fused Filament Fabrication," Ph.D. dissertation, 2024. [Online]. Available: https://deepblue.lib.umich.edu/handle/2027.42/194671

[274] R. Badarinath and V. Prabhu, "Real-time sensing of output polymer flow temperature and volumetric flowrate in fused filament fabrication process," *Materials*, vol. 15, no. 2, p. 618, 2022, doi: 10.3390/ma15020618.

[275] P. Bosetti and E. Bertolazzi, "Feed-Rate and Trajectory Optimization for CNC Machine Tools," *Robotics and Computer-Integrated Manufacturing*, vol. 30, no. 6, pp. 667–677, 2014, doi: 10.1016/j.rcim.2014.03.009.

[276] E. Bertolazzi, F. Biral, and M. Da Lio, "Real-Time Motion Planning for Multibody Systems: Real Life Application Examples," *Multibody System Dynamics*, vol. 17, no. 2, pp. 119–139, 2007, doi: 10.1007/s11044-007-9037-7.

[277] C. M. Giorgio Bort, M. Leonesio, and P. Bosetti, "A Model-Based Adaptive Controller for Chatter Mitigation and Productivity Enhancement in CNC Milling Machines," *Robotics and Computer-Integrated Manufacturing*, vol. 40, pp. 34–43, 2016, doi: 10.1016/j.rcim.2016.01.006.

[278] Jason S, "Understanding motor constants Kt and Kemf for comparing brushless DC motors." [Online]. Available: https://electronics.stackexchange.com/q/33332

[279] P. M. Wensing, A. Wang, S. Seok, D. Otten, J. Lang, and S. Kim, "Proprioceptive Actuator Design in the MIT cheetah: Impact Mitigation and High-Bandwidth Physical Interaction for Dynamic Legged Robots," *IEEE Transactions on Robotics*, vol. 33, no. 3, pp. 509–522, 2017, doi: 10.1109/TRO.2016.2640183.

[280] D. Wilson, "Teaching Your PI Controller to Behave (Parts I–X)." [Online]. Available: https://e2e.ti.com/blogs_/b/industrial_strength/posts/teaching-your-pi-controller-to-behave-part-i

[281] J. R. Read, "Field Oriented Control with (almost any?) Stepper." [Online]. Available: https://ekswhyzee.com/2025/03/24/foc-stepper.html

[282] R. E. Kalman, "A New Approach to Linear Filtering and Prediction Problems," *Transactions of the ASME – Journal of Basic Engineering*, vol. 82, no. 1, pp. 35–45, 1960, doi: 10.1115/1.3662552.

[283] D. Simon, *Optimal State Estimation: Kalman, H-Infinity, and Nonlinear Approaches*. Wiley-Interscience, 2006.

[284] C. Zhu, R. H. Byrd, P. Lu, and J. Nocedal, "Algorithm 778: L-BFGS-B: Fortran Subroutines for Large-Scale Bound-Constrained Optimization," *ACM Transactions on Mathematical Software*, vol. 23, no. 4, pp. 550–560, 1997, doi: 10.1145/279232.279236.

[285] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*, 2015. [Online]. Available: https://arxiv.org/abs/1412.6980

[286] B. Katz, "Controls Ramblings: How to Get from Point A to Point B Very Fast (and Stop)." [Online]. Available: https://robot-daycare.com/posts/2017-12-31-controls-ramblings-how-to-get-from-point-a-to-poin/

[287] B. Katz, "Transmission Ratio Trajectory Optimization." [Online]. Available: https://robot-daycare.com/posts/2020-11-18-transmission-ratio-trajectory-optimization/

[288] Wikipedia contributors, "Phase Space." [Online]. Available: https://en.wikipedia.org/wiki/Phase_space

[289] A. H. Slocum, "FUNdaMENTALs of Design." [Online]. Available: https://meddevdesign.mit.edu/fundamentals-of-design/

[290] Prusa Polymers a.s., "Prusament PC Blend: Technical Datasheet," Prague, Czech Republic, technical report, Oct. 2022. [Online]. Available: https://prusament.com/wp-content/uploads/2022/10/PCBlend_Prusament_TDS_2022_16_EN.pdf

[291] Prusa Polymers a.s., "Prusament PC Blend CF: Technical Datasheet," Prague, Czech Republic, technical report, Apr. 2023. [Online]. Available: https://prusament.com/wp-content/uploads/2023/04/TDS_Prusament-PCCF_2023_EN.pdf

[292] T. J. Coogan and D. O. Kazmer, "Prediction of Interlayer Strength in Material Extrusion Additive Manufacturing," *Additive Manufacturing*, vol. 35, p. 101368, 2020, doi: 10.1016/j.addma.2020.101368.

[293] M. E. Mackay, "The Importance of Rheological Behavior in the Additive Manufacturing Technique Material Extrusion," *Journal of Rheology*, vol. 62, no. 6, pp. 1549–1561, 2018, doi: 10.1122/1.5037687.

[294] J. Go, S. N. Schiffres, A. G. Stevens, and A. J. Hart, "Rate Limits of Additive Manufacturing by Fused Filament Fabrication and Guidelines for High-Throughput System Design," *Additive Manufacturing*, vol. 16, pp. 1–11, 2017, doi: 10.1016/j.addma.2017.03.007.

[295] Prusa Research a.s., "3D Printing Handbook: Core One." 2025. [Online]. Available: https://www.prusa3d.com/downloads/manual/prusa3d_manual_coreone_101_en.pdf

[296] Prusa Research, "PrusaSlicer: Open-Source 3D Printer Slicer Software." [Online]. Available: https://github.com/prusa3d/PrusaSlicer/tree/version_2.9.4

[297] Prusa Research, "Linear Advance in FFF Printing." [Online]. Available: https://help.prusa3d.com/article/linear-advance_2252

[298] Prusa Research, "Pressure Advance." [Online]. Available: https://help.prusa3d.com/article/pressure-advance_814986

[299] M. Abbot, "Rubedo: A System for Automatically Calibrating Pressure Advance Using Laser Triangulation." [Online]. Available: https://github.com/furrysalamander/rubedo

[300] B. N. Turner, R. Strong, and S. A. Gold, "A Review of Melt Extrusion Additive Manufacturing Processes: I. Process Design and Modeling," *Rapid Prototyping Journal*, vol. 20, no. 3, pp. 192–204, 2014, doi: 10.1108/rpj-01-2013-0012.

[301] T. J. Coogan and D. O. Kazmer, "In-Line Rheological Monitoring of Fused Deposition Modeling," *Journal of Rheology*, vol. 63, no. 1, pp. 141–155, 2019, doi: 10.1122/1.5054648.

[302] D. O. Kazmer, A. R. Colon, A. M. Peterson, and S. K. Kim, "Concurrent Characterization of Compressibility and Viscosity in Extrusion-Based Additive Manufacturing of Acrylonitrile Butadiene Styrene with Fault Diagnoses," *Additive Manufacturing*, vol. 46, p. 102106, Oct. 2021, doi: https://doi.org/10.1016/j.addma.2021.102106.

[303] D. O. Kazmer, A. Colon, T. Coogan, A. Mubasshir, and A. Peterson, "Compressibility in Fused Filament Fabrication," *Society of Plastics Engineers Annual Technical Conference*, [Online]. Available: https://par.nsf.gov/biblio/10172592

[304] D. O. Kazmer and A. M. Peterson, "Compressibility in Fused Deposition Modeling," *Society of Plastics Engineering Annual Technical Conference, Additive Manufacturing Division*, [Online]. Available: https://par.nsf.gov/biblio/10341316

[305] J. R. Read, J. E. Seppala, F. Tourlomousis, J. A. Warren, N. Bakker, and N. Gershenfeld, "Online Measurement for Parameter Discovery in Fused Filament Fabrication," *Integrating Materials and Manufacturing Innovation*, vol. 13, no. 2, pp. 541–554, 2024, doi: 10.1007/s40192-024-00350-w.

[306] A. Moshe, D. O. Kazmer, M. J. Sobkowicz-Kline, S. P. Johnston, and S. Kenig, "Transient modeling of viscosity," *Polymer Engineering & Science*, vol. 57, no. 10, pp. 1110–1118, 2017, doi: 10.1002/pen.24486.

[307] M. E. Mackay, Z. R. Swain, C. R. Banbury, D. D. Phan, and D. A. Edwards, "The Performance of the Hot End in a Plasticating 3D Printer," *Journal of Rheology*, vol. 61, no. 2, pp. 229–236, 2017, doi: 10.1122/1.4973852.

[308] D. A. Anderegg *et al.*, "In-situ monitoring of polymer flow temperature and pressure in extrusion based additive manufacturing," *Additive Manufacturing*, vol. 26, pp. 76–83, 2019, doi: 10.1016/j.addma.2019.01.002.

[309] J. Zhang *et al.*, "Temperature analyses in fused filament fabrication: from filament entering the hot-end to the printed parts," *3D Printing and Additive Manufacturing*, vol. 9, no. 2, pp. 132–142, 2022, doi: 10.1089/3dp.2020.0339.

[310] G. C. Nzebuka, C. O. Ufodike, A. M. Rahman, C. M. Gwynn, and M. F. Ahmed, "Numerical modeling of the effect of nozzle diameter and heat flux on the polymer flow in fused filament

fabrication," *Journal of Manufacturing Processes*, vol. 82, pp. 585–600, 2022, doi: 10.1016/j.jmapro.2022.08.029.

[311] F. Pigeonneau, D. Xu, M. Vincent, and J.-F. Agassant, "Heating and flow computations of an amorphous polymer in the liquefier of a material extrusion 3D printer," *Additive Manufacturing*, vol. 32, p. 101001, 2020, doi: https://doi.org/10.1016/j.addma.2019.101001.

[312] P. Wu, C. Qian, and C. E. Okwudire, "Design, modeling and feedforward control of a hybrid extruder for material extrusion additive manufacturing," *Additive Manufacturing*, vol. 92, p. 104378, 2024, doi: 10.1016/j.addma.2024.104378.

[313] D. A. J. Brion and S. W. Pattinson, "Quantitative and Real-Time Control of 3D Printing Material Flow Through Deep Learning," *Advanced Intelligent Systems*, vol. 4, no. 11, p. 2200153, 2022, doi: 10.1002/aisy.202200153.

[314] D. A. Brion and S. W. Pattinson, "Generalisable 3D printing error detection and correction via multi-head neural networks," *Nature communications*, vol. 13, no. 1, p. 4654, 2022, doi: 10.1038/s41467-022-31985-y.

[315] C. S. Davis, K. E. Hillgartner, S. H. Han, and J. E. Seppala, "Mechanical Strength of Welding Zones Produced by Polymer Extrusion Additive Manufacturing," *Additive Manufacturing*, vol. 16, pp. 162–166, Aug. 2017, doi: https://doi.org/10.1016/j.addma.2017.06.006.

[316] A. R. Colon, D. O. Kazmer, A. M. Peterson, and J. E. Seppala, "Characterization of die-swell in thermoplastic material extrusion," *Additive Manufacturing*, vol. 73, p. 103700, 2023, doi: https://doi.org/10.1016/j.addma.2023.103700.

[317] S. K. Kim and D. O. Kazmer, "Non-isothermal non-Newtonian three-dimensional flow simulation of fused filament fabrication," *Additive Manufacturing*, vol. 55, p. 102833, 2022, doi: 10.1016/j.addma.2022.102833.

[318] J. Zhang, F. Meng, and E. Ferraris, "Temperature gradient at the nozzle outlet in material extrusion additive manufacturing with thermoplastic filament," *Additive Manufacturing*, vol. 73, p. 103660, 2023, doi: 10.1016/j.addma.2023.103660.

[319] C. McIlroy and P. D. Olmsted, "Disentanglement effects on welding behaviour of polymer melts during the fused-filament-fabrication method for additive manufacturing," *Polymer*, vol. 123, pp. 376–391, 2017, doi: 10.1016/j.polymer.2017.06.051.

[320] J. A. Afonso, J. L. Alves, G. Caldas, B. P. Gouveia, L. Santana, and J. Belinha, "Influence of 3D Printing Process Parameters on the Mechanical Properties and Mass of PLA Parts and Predictive Models," *Rapid Prototyping Journal*, vol. 27, no. 3, pp. 487–495, 2021, doi: 10.1108/rpj-03-2020-0043.

[321] S. Deshwal, A. Kumar, and D. Chhabra, "Exercising Hybrid Statistical Tools GA-RSM, GA-ANN and GA-ANFIS to Optimize FDM Process Parameters for Tensile Strength Improvement," *CIRP Journal of Manufacturing Science and Technology*, vol. 31, pp. 189–199, 2020, doi: 10.1016/j.cirpj.2020.05.009.

[322] A. Qattawi, B. Alrawi, A. Guzman, and others, "Experimental Optimization of Fused Deposition Modelling Processing Parameters: A Design-for-Manufacturing Approach," *Procedia Manufacturing*, vol. 10, pp. 791–803, 2017, doi: 10.1016/j.promfg.2017.07.079.

[323] O. Luzanin, D. Movrin, V. Stathopoulos, P. Pandis, T. Radusin, and V. Guduric, "Impact of Processing Parameters on Tensile Strength, In-Process Crystallinity and Mesostructure in FDM-Fabricated PLA Specimens," *Rapid Prototyping Journal*, vol. 25, no. 8, pp. 1398–1410, 2019, doi: 10.1108/rpj-12-2018-0316.

[324] P. Ferretti *et al.*, "Relationship Between FDM 3D Printing Parameters Study: Parameter Optimization for Lower Defects," *Polymers*, vol. 13, no. 13, p. 2190, 2021, doi: 10.3390/polym13132190.

[325] T. Sanladerer, "InFiDEL Filament Sensor." [Online]. Available: https://www.youtube.com/watch?v=RYgdLPe_T0c

[326] "Prusa Nextruder." [Online]. Available: https://blog.prusa3d.com/the-original-prusa-mk4s-is-here_100605/

[327] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu, "A Limited Memory Algorithm for Bound Constrained Optimization," *SIAM Journal on Scientific Computing*, vol. 16, no. 5, pp. 1190–1208, 1995, doi: 10.1137/0916069.

[328] Prusa Research, "New in PrusaSlicer: Consistent Surface Finish and Nerfing VFAs on the CORE One and Other Printers!." Accessed: Nov. 14, 2025. [Online]. Available: https://blog.prusa3d.com/new-in-prusaslicer-consistent-surface-finish-and-nerfing-vfas_120400/

[329] G. E. Blelloch, "Prefix Sums and Their Applications," technical report CMU-CS-90-190, 1990. [Online]. Available: https://www.cs.cmu.edu/~guyb/papers/Ble93.pdf

[330] "E3D Obxidian Nozzles." [Online]. Available: https://e3d-online.com/pages/revo-support-obxidian-nozzles

[331] Fillamentum, "Timberfill Filament," Prague, Czech Republic, technical report, Oct. 2020. [Online]. Available: https://fillamentum.com/collections/timberfill-filament/

[332] The Virtual Foundry, "Aluminum 6061 Filamet," Wisconsin, USA, technical report, 2025. [Online]. Available: https://thevirtualfoundry.com/

[333] R. S. Sutton, "The Bitter Lesson." [Online]. Available: http://www.incompleteideas.net/IncIdeas/BitterLesson.html

[334] C. E. Duty *et al.*, "Structure and mechanical behavior of Big Area Additive Manufacturing (BAAM) materials," *Rapid Prototyping Journal*, vol. 23, no. 1, pp. 181–189, 2017, doi: 10.1108/RPJ-12-2015-0183.

[335] F. Mornati, "Pareto Optimality in the work of Pareto," *European Journal of Social Sciences*, no. 51–2, pp. 65–82, 2013, doi: 10.4000/ress.2517.

[336] Y. Censor, "Pareto optimality in multiobjective problems," *Applied Mathematics and Optimization*, vol. 4, no. 1, pp. 41–59, 1977, doi: 10.1007/bf01442131.

[337] C. Sharma, "Automatic Modeling of Machining Processes," M.Eng. thesis, 2021.

[338] K. Dunwoody, "Automated Identification of Cutting Force Coefficients and Tool Dynamics on CNC Machines," M.A.Sc. thesis, 2010.

[339] V. F. C. Sousa *et al.*, "Cutting Forces Assessment in CNC Machining Processes: A Critical Review," *Sensors*, vol. 20, no. 16, 2020, doi: 10.3390/s20164536.

[340] D. Aslan and Y. Altintas, "On-Line Chatter Detection in Milling Using Drive Motor Current Commands Extracted from CNC," *International Journal of Machine Tools and Manufacture*, vol. 132, pp. 64–80, 2018, doi: 10.1016/j.ijmachtools.2018.04.007.

[341] T. L. Schmitz and K. S. Smith, *Machining Dynamics: Frequency Response to Improved Productivity*. Springer, 2009. doi: 10.1007/978-0-387-09645-2.

[342] Y. Altintaş and E. Budak, "Analytical Prediction of Stability Lobes in Milling," *CIRP Annals*, vol. 44, no. 1, pp. 357–362, 1995, doi: 10.1016/S0007-8506(07)62342-7.

[343] S. Butterworth and others, "On the Theory of Filter Amplifiers," *Wireless Engineer*, vol. 7, no. 6, pp. 536–541, 1930.

[344] "Rat Rig Mill." [Online]. Available: https://ratrig.com/products/rat-rig-mill

[345] "GPenny 1.5kW Spindle." [Online]. Available: https://gpennymachine.com/

[346] "Universal Fixture Plate Kit." [Online]. Available: https://saundersmachineworks.com/

[347] "Aluminum Nonferrous Coatings." [Online]. Available: https://www.secotools.com/article/a245_a345_family_of_end_mills

[348] A. Shokrani *et al.*, "Sensors for In-Process and On-Machine Monitoring of Machining Operations," *CIRP Journal of Manufacturing Science and Technology*, vol. 51, pp. 263–292, 2024, doi: 10.1016/j.cirpj.2024.05.001.

[349] J. Warren, J. Read, J. Seppala, E. Strand, and N. Gershenfeld, "Computational Metrology for Materials," *Journal of Materials Research*, vol. 40, no. 15, pp. 2197–2203, 2025, doi: 10.1557/s43578-025-01651-2.

[350] F. R. Dewey, "A complete guide to datasheets," *Sensors Magazine*, 1998, [Online]. Available: https://www.allegromicro.com/-/media/files/technical-documents/complete-guide-to-datasheets-pub26000.pdf

[351] J. Bonvoisin, J. Molloy, M. Häuer, and T. Wenzel, "Standardisation of Practices in Open Source Hardware," *Journal of Open Hardware*, vol. 4, no. 1, p. 2, Aug. 2020, doi: 10.5334/joh.22.

[352] M. Ivanou, S. Mikhel, and S. Savin, "Robot Description Formats and Approaches: Review," in *2021 International Conference "Nonlinearity, Information and Robotics" (NIR)*, 2021, pp. 1–5. doi: 10.1109/NIR52917.2021.9666120.

[353] U. Fasel, E. Kaiser, J. N. Kutz, B. W. Brunton, and S. L. Brunton, "SINDy with Control: A Tutorial," in *2021 60th IEEE Conference on Decision and Control (CDC)*, 2021, pp. 16–21. doi: 10.1109/CDC45484.2021.9683120.

[354] J. Yates and C. N. Murphy, *Engineering Rules: Global Standard Setting Since 1880*. JHU Press, 2019.

[355] A. Nuvolari, "Collective invention during the British Industrial Revolution: the case of the Cornish pumping engine," *Cambridge Journal of Economics*, vol. 28, no. 3, pp. 347–363, 2004, doi: 10.1093/cje/28.3.347.