Title
Conspicuous Computing: Software Development as a Knowledge Practice

Author
Erickson, Seth

Publication Date
2018

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA

Los Angeles

Conspicuous Computing:
Software Development as a Knowledge Practice

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Information Studies

by

Seth Erickson

2018

ABSTRACT OF THE DISSERTATION

Conspicuous Computing:
Software Development as a Knowledge Practice

by

Seth Erickson
Doctor of Philosophy in Information Studies
University of California, Los Angeles, 2018
Professor Christopher M. Kelty, Chair

This dissertation is a study of scientific software development based on ethnographic fieldwork in a computational physics lab. I present scientific software development as the ongoing materialization of epistemic objects. *Conspicuousness* is a term adopted from phenomenology to describe a reflective engagement with software that is distinct from the unreflective, seamless, and transparent relationship that designers often aspire to. To the extent that scientific software matches the trajectory of inquiry, it is frequently conspicuous in this sense—in fact, it must be so if it is to function as a setting in which new insights emerge. The maturity of computational physics makes it an ideal context for understanding how knowledge production inheres in code work and for identifying how recent changes in computational and scientific norms have affected scientific software practices. The field site was a university lab specializing in the development and use of codes for simulating plasma. I characterize the memory practices of the lab as efforts to make the unexpected behaviors of code productive. I describe the lab's deep-seated ambivalence toward open science initiatives and reflect on the conditions that allow scientists to maintain autonomy and control over their technical practices. Finally, lab members differentiated between production codes, reference codes, frameworks, classroom codes, and toy codes: I present these actor categories as a set of commonplaces that allowed simulation developers to manage the epistemological concerns of software development and the reception of their code in the broader community.

The dissertation of Seth Erickson is approved.

Jean-François Blanchette

Johanna R. Drucker

Jacob Gates Foster

Christopher M. Kelty, Committee Chair

University of California, Los Angeles

2018

TABLE OF CONTENTS

1 Introduction **1**

1.1 Review of Scientific Software Development 3

1.1.1 Improving Scientific Software 4

1.1.2 Conceptualizing Scientific Software Practices 6

1.1.3 Justifying Scientific Software 9

1.2 What is Conspicuous Computing? 11

1.3 Research Design 15

1.4 The Site 17

1.5 Overview of the Dissertation 21

1.6 Bibliography 24

2 Simulation Development as a Memory Practice **29**

2.1 Abstract 29

2.2 Introduction: A Scientific Software Literature 29

2.3 Memory Practices 36

2.4 The Performative Memory of Cybernetics 39

2.5 Software Engineering's Formalization of Software 42

2.5.1 'Opacity' of Simulation 46

2.6 Making the Unexpected Productive 48

2.6.1 'We've seen this for years' 49

2.6.2 Disputing the Unexpected 53

2.6.3 The 'Historiality' of the Unexpected 54

2.7 Conclusion 57

2.8 Bibliography 59

3 Opening Plasma Codes 62

3.1 Abstract 62

3.2 Introduction 62

3.3 Policy Visions of Scientific Software 69

3.3.1 Cyberinfrastructure 69

3.3.2 Open Science 73

3.4 Simulationists' Ambivalence to Openness 76

3.4.1 Crafting Input Decks 78

3.4.2 'Developers are the best users' 81

3.4.3 Tensions Between Users and Developers 84

3.4.4 The User-Developer Regress 89

3.5 Conclusion 90

3.6 Bibliography 92

4 Commonplaces of Plasma Simulation 96

4.1 Abstract 96

4.2 Introduction 96

4.3 The Multiplexity of Simulation Codes 101

4.4 The Particle-in-cell Algorithm 103

4.5 Production Codes 105

4.5.1 An historical example: ISIS and the charge-conserving scheme 106

4.5.2 Unfolding Production Codes in Graduate Research 109

4.6 Technical Condition: Reference Codes and Frameworks 113

4.6.1 Reference Codes 115

4.6.2 Frameworks 117
4.7 Classroom Codes 118
4.8 Toy Codes 121
4.9 Discussion 123
4.10 Conclusion 126
4.11 Bibliography 128

5 Conclusion 131
5.1 Poetics and Pragmatics 131
5.1.1 Implications for Open Science & Software Curation 132
5.2 Conspicuousness Revisited 134
5.3 Bibliography 136

LIST OF FIGURES

3.1 The NSF's (2012) three-tiered model of cyberinfrastructure software. 72

4.1 Lars' Guiding Center Simulation (fieldnotes) 97

4.2 Particle-in-Cell Grid 104

ACKNOWLEDGMENTS

I am grateful for the attention of the readers who have guided this dissertation, particularly my committee members, Jean-François Blanchette, Johanna Drucker, and Jacob Foster. I am especially thankful for the generosity and kindness of Christopher Kelty, my advisor and committee chair, who has saved this precarious vessel from the rocks on many occasions.

I have benefited tremendously from the support of my colleagues—Marika Cifor, Roderic Crooks, Morgan Currie, Patricia Garcia, Robert Montoya, and Stacy Wood—with whom I shared the emotional swings of graduate school and dissertation writing.

Finally, I would like to thank my mother, father, sister, and brother—Brenda, Danny, Ansley, and Keefer—and the family I have found here in Los Angeles through Angel Diaz (and our dog, Wylie). To Angel: Thank you for your love, your patience, and your trust.

VITA

2004 B.A., Brown University, History of Art and Architecture

2014 M.L.I.S., University of California, Los Angeles, Graduate School of Education and Information Studies

PUBLICATIONS

Kelty, C., & Erickson, S. (2018). Two modes of participation: A conceptual analysis of 102 cases of Internet and social media participation from 2005–2015. *The Information Society*, 34(2), 71–87.

Montoya, R. D., & Erickson, S. R. (2017). Anachronism in Global Information Systems: the cases of Catalogue of Life and Unicode. *iConference 2017 Proceedings*, 2, 123–128.

Erickson, S. R., & Kelty, C. M. (2015). The Durability of Software. In I. Kaldrack & M. Leeker (Eds.), *There is no Software, There are Just Services*. Meson Press.

Kelty, C., Panofsky, A., Currie, M., Crooks, R., Erickson, S. R., Garcia, P., ... Wood, S. (2015). Seven dimensions of contemporary participation disentangled. *Journal of the Association for Information Science and Technology*, 66(3), 474–488.

CHAPTER 1

Introduction

While this dissertation is about software development in a computational physics lab, conceptually, it began in the humanities.[1] In 2012, I had the opportunity to work with an art historian on a project to analyze and visualize the transactions of one of the major European art dealers during the 19th and early 20th centuries. My role in the project was mostly technical—I wrote a small program to clean the data and generate network graphs of the market, and I included a brief "project narrative" in one of resulting publications (Fletcher and Helmreich 2012). Reflecting on the narrative, I noticed later that my way of talking about the project switched between two kinds of emphasis. Sometimes I emphasized the evolution of the software in technical terms.

> "The command-line tool [...] was written in Ruby because it is a robust scripting language [...] initially, the tool was implemented in a very procedural manner. [...] Later in the project [...] the tool was re-implemented using object-relational mapping and a SQLite database."

Other times, I explained the questions and choices that concerned the inquiry itself.

> "what kinds of network graphs should we generate? [...] Our initial strategy [...] was to generate graphs with three node types. Later in the project [...] we also implemented the one-type graphs for [dealer] locations linked by transactions with common artists [...]."

[1] And there I hope it may continue.

One might conjecture from this informal self-analysis, as I did, that there two ways of speaking about software in the context of research—one that follows the trajectory of 'code itself' and another that follows the trajectory of inquiry. The distinction between these types of accounts is not that surprising. It is an expression of the autonomy that programming has achieved as a distinct domain of practice with broad social relevance. After all, 'software development' is a collection of ways to deal with code—all code. Coding handbooks offer advice for writing code that is easy to understand and sustainable; version control systems are helpful for tracking code changes; automated tests exercise components separately to demonstrate their basic functionality and to prevent the accidental introduction of bugs. These strategies are part of the "unfolding" of software (Knorr-Cetina 2008). they are procedures that are continuously reapplied in order to make software development tractable, and they constitute its coherence as an area of practice. *Any* account of a software development project might be described as the exercise (or neglect) of these strategies.

However, the practices of software development are increasingly recognized as social, political, and epistemological concerns. The trajectories of code and inquiry are not independent. The "project narrative" itself expresses this fact. As the authors noted in a remark that prefaced my account of the project: "Digital scholarship involves technological experimentation and significant collaboration. [...] we document and reflect upon those processes.[...] we provide an overview of the data used in our analyses and our working process as well as brief introductions to the key individuals involved in this project and their roles and responsibilities" (Fletcher and Helmreich 2012). This statement reflects a uniquely *scholarly* sensibility and relationship to code work. The impulse to 'open up', describe, and understand the research process—so that it may be examined, adapted, and perhaps critiqued—is a very scholarly impulse. Researchers are in the habit of being able to evaluate their peers' claims, borrow techniques, adapt them to new situations, and to do so in a way that is intellectually honest and consistent with the epistemological commitments of their respective fields. These practices describe the unfolding of scholarly work.

There are risks in bringing these two trajectories together, however. Care is needed because of the voraciousness of the computational metaphor—the prevailing sense that the

programmable computer can stand in for most anything else, including thought. "Computational thinking" (Wing 2006) is marked by a seductive prestige; though it is attractive to many, it does not always play nicely with the interpretive traditions and epistemological orientations of researchers (Drucker 2009). Perhaps the notion of "software development as a knowledge practice"—the concept and question at the center of this project—involves a category mistake, a misguided crossing of multiple interpretive keys (Latour 2013). (For example, it is a category mistake to expect the "truth," in a scientific or objective sense, as an outcome of a legal proceeding). Perhaps objects of knowledge and objects of computation have two modes of existence, which though related, need be understood as distinct in essential ways.

This dissertation examines these concerns through an investigation of software practices in a computational physics lab, specifically a lab specializing in the development and use of plasma simulation codes. My general concern is to see how epistemological value is created, sustained, adapted, and generally put to work in the life of scientific software. The two trajectories of research software just described might be understood as distinct but mutually involved chains of activity. The goal of this project is to understand this dynamic: how the "unfolding" of scholarship, on the one hand, and software, on the other, relate to each other. I place the notion of *conspicuousness* at the center of this process of unfolding—it is the motive force, bringing computational objects into being. In the following sections I review, first, work that directly bears on the topic of scientific software development and, second, work that informs by understanding of conspicuous computing.

1.1 Review of Scientific Software Development

There are three broad areas of scholarship that relate to and inform this study of software development as a scientific knowledge practice: normative efforts to improve practices associated with scientific software (this work is mostly associated with software engineering, computer supported cooperative work, and information studies); efforts to conceptualize technical practices in the sciences (from science studies and history of science); and efforts

to explain and justify the epistemology of software, particularly simulation (from philosophy of science). Here, I briefly review these areas of work, drawing attention to how they relate to and inform the discussion of software development as a knowledge practice.

1.1.1 Improving Scientific Software

Much of the current research on scientists' software development practices is oriented toward its *improvement*. The slogan of the Software Sustainability Institute (a center for "research software engineering" in the UK), aptly characterizes the broad motivation of work in this area: "better software, better research" (Goble 2014). Software engineers have made extensive use of case study to understand and identify ways to improve local software development practices (Robinson, Segal, and Sharp 2007); they have found that traditional software engineering methods, like specification-driven development, do not work well for the "emergent requirements" of scientific software. Scientists often don't know precisely what they need from software in advance (Segal 2005). A common finding is that scientists lack the training and interest to adopt formal methods from software engineering (Segal and Morris 2008). Further, particular scientific domains come with their own peculiar challenges. Easterbrook and Johns' (2009) ethnographic study of Earth system model development, for example, shows how modelers evaluate the 'correctness' of their models. The authors note that testing models of the Earth is complicated by (among other things) the consideration that "a model's value as a scientific instrument doesn't always depend on its degree of isomorphism with the physical world" (65). Further, modelers tend to approach software development in scientific terms: "[Modelers] treat each model change [code revision] as an experiment. [...] They don't need 'finished' software to perform these experiments, but they continually experiment with the software itself to improve their understanding" (Easterbrook and Johns 2009, 70).

While case studies from software engineering are useful for demonstrating the specificity of practices in particular communities of scientific computing, the disciplinary context of software engineering somewhat constrains the elaboration of their significance. For example,

in their observations of Earth system modelers, Easterbrook and Johns keenly observe that, "it was hard to distinguish software development from other aspects of the scientific practice" (73). (Note the previous comparison of code changes to experiments). However, for the purposes of software engineering, the difficulty of distinguishing between 'doing science' and 'doing code' is a limitation (the authors literally note this detail in a section of their article called "Limitations"). The very multivalence that is peripheral and a source of difficulty for software engineers is one that I hope to put at the center of this study.

In information studies (IS), scientific software development is primarily addressed in relation to scholarly communication and "knowledge infrastructure" (Edwards et al. 2013). In this context, software is recognized as one part of a larger transformation in the social and technical conditions of science—or, more generally, in the ways knowledge is created, shared, and disputed. Software is associated with a host of scientific benefits and "epistemic virtues" (Daston and Galison 2007), each of which has a 'flip side,' a potential threat to the social role of scientific institutions. Key concerns in this area are *sharing* and *reuse*—terms that apply to both research data and research software (Borgman 2012). Specifically, code sharing is recognized to be necessary for both the "unprecedented levels of [...] collaborative innovation" in science (Howison and Herbsleb 2011, 513) and the reproducibility of computational research (Stodden, Leisch, and Peng 2014). To achieve the benefits of sharing research software, these studies seek to, first, understand the obstacles that prevent scientists from sharing their software and, second, identify policies and incentive structure that relevant institutions (typically funding agencies and publishers) can adopt to encourage scientists to make their software available.

A particular area of concern in IS that this project relates to is digital curation. Curation is 'care work' (Karasti, Baker, and Halkola 2006): it involves the collection, description, preservation, and ongoing maintenance of digital objects with the aim of supporting their value. Typically, "value" is understood as a capacity for reuse by an extended community (Mayernik et al. 2013). Software is increasingly recognized as a particular important yet precarious object in data curation work (Lynch 2014). This dissertation seeks to contribute to discussions around software curation by characterizing software development as a knowledge

practice and by identifying the forms of value associated with it.

1.1.2 Conceptualizing Scientific Software Practices

Social, cultural, and historical studies of scientific work provide a significant body of research relevant to understanding scientists' engagement with instruments. Much of this work characterizes laboratory work as 'messy', performative, tightly bound to local settings, and driven by immediate requirements to "make recalcitrant material do what it is supposed to" (Sismondo 2010, 108). Pickerings' account of science as a "mangle"—as an ongoing struggle with material agency—is representative of this body of work (Pickering 1995).

The instrument practices of physicists are particularly well represented in science studies[2]. I will preface the discussion of work on physics by first introducing Rheinberger's (1997) account of experimental systems, which he developed in the context of a history of protein biosynthesis. Rheinberger's characterization of the temporal structure of experimental systems—the "movement" of science—is a recurring reference in this dissertation (particularly in chapters two and four) and it is central to my understanding of 'conspicuousness', as described below.

Rheinberger decomposes experimental systems (defined as the smallest functional units of scientific knowledge production) into *epistemic things* and *technical objects*. Epistemic things are the incomplete objects of study; they are sources of contention and dispute as they are still in the process of being materially (ontologically) defined. If epistemic things are "question-generating machines," technical objects are "answering machines" (Rheinberger 1997, 312). Technical objects are akin to Latour's black boxes—they are the standards, instruments, and shared conditions of experimental science. Technical objects constitute historically specific spaces of representation within which the "scientific real" is produced. Thus, what differentiates technical objects from Latour's black boxes is the broader temporal (in fact, historical) structure within which Rheinberger conceptualizes the former. The

[2]Here, 'science studies' refers to both science and technology studies (STS) and history of science. Properly, these are distinct fields however there is extensive overlap between the two communities.

temporal dynamic of science, Rheinberger argues, involves the production of unexpected, novel events (new epistemic things) within experimental systems; as the former are materially defined they come to constitute new experimental conditions—which gives rise to new epistemic things, etc. This process characterizes the non-teleological 'movement' of science.

Knorr-Cetina takes issue with the degree of separation Rheinberger places between epistemic things and technical objects. "The equation of instruments with technological objects is highly problematic [...] in light of today's technologies, which are simultaneously things-to-be-used and things-in-the-process-of-transformation" (Knorr-Cetina 1997, 10). Because the instruments of contemporary science often demand constant investigation and modification, they are not stable conditions of production. Technical conditions are *also* epistemic objects—this is particularly true in the case of physics. The example of particle detection in high energy physics illustrates Knorr-Cetina's point: "The detector is an apparatus that is self-created and assembled within the experiment. Nonetheless, the behavior of this apparatus, its performance, blemishes, and ailments are not self-evident to the physicists. These features must be learned and the project of understanding the behavior of the detector spells this out" (Knorr-Cetina 2009, 56). The work of understanding the detector does not end. The detector ages, its operational conditions evolve, it is modified, and with each modification the effort to understand the detector is renewed. Commercial software is another example, as programs are treated as stable commodities while also undergoing constant updating and revision (Knorr-Cetina 1997; Knorr-Cetina 2008).

Merz (1999) builds on Knorr-Cetina's work to argue that the simulation codes used in high energy physics are "multiplex" with respect to the epistemic thing/technical object distinction. To *some* physicists *some* of the time, they are treated as "question-generating" machines, while in other situations they are treated as stable instruments with assumed properties. The tendency to treat codes as either epistemic objects or technical things largely coincides with the distinction between contexts of development and use. While there are exceptions—developers are also users, and there are categories of use in which the provisionality of code is epistemically relevant—the negotiation between various roles of research software often culminates in a negotiation between users and developers.

Compared to experimental work, scientific software development (particularly work with computational models) is a relatively new area of inquiry in science studies. The framing of the functional roles of software in science—here traced to Rheinberger by way of Knorr-Cetina and Merz—is a central theme in recent work on this topic (Sundberg 2009; Spencer 2015). I return to these concerns—particularly to the question of the roles of research software in the case of plasma simulation—in chapter four.

Beyond the nexus of Rheinberger, Knorr-Cetina, and Merz, Collins' work with the gravity wave detection community focuses on the difficulties of transferring knowledge about the construction and use of instruments between laboratories. The central issue Collins identifies is that much of scientists' understanding of how to build and operate instruments is *tacit knowledge*, knowledge that cannot be explicated. The only way to acquire the deep understanding associated with tacit knowledge is through socialization in the community of experts who possess it. Tacit knowledge is intimately connected to expertise, according to Collins. Expertise can only be acquired and maintained through social interaction (Collins and Evans 2008).

One of the notable aspects of Collins' argument about tacit knowledge and expertise is the role of computation in his conceptualization of these terms. In his various presentations of the topic, Collins consistently uses computing to demonstrate the difference between tacit and explicit knowledge. Algorithmic representation is the prototypical example of explicit knowledge, and the limits of computational representations are used illustrate the limits of explicit knowledge. In his typology of expertise, *ubiquitous* expertise (such as the ability to use a human language) only becomes apparent when we try to program it into a computer. "It was modernism in general and the computer revolution in particular that made the explicit seem easy and the tacit seem obscure" (Collins 2010, 7). Tacit knowledge becomes a particularly pressing problem when we expect computers to be able to replicate human expertise.

1.1.3 Justifying Scientific Software

Philosophy of science has had a thriving discussion on a range of issues associated with simulation and computational modeling in recent years (Winsberg 2010; Parker 2013). In these discussions, the material particulars of code work (i.e., the production and manipulation of computational models) are frequently drawn into consideration (Knuuttila and Voutilainen 2003; Gramelsberger 2011). Broad questions in the philosophy of simulation concern the epistemological status of computational models (i.e., in what sense are claims derived from simulations *justified*?) and the relationship of simulation to experimentation, theory, and representation. Winsberg (2018) provides a recent treatment of the debates in philosophy of simulation. The narrow aspect that I review here is how software development enters into these discussions as a philosophically relevant practice.

One of the conventional arguments for justifying claims derived from simulations is that simulation codes are rooted in *theoretical principles*—the results from a code are seen as correct to the degree that the code is faithful to an underlying theory. This view can be schematically described as a chain of representation: a particular scientific object (e.g., 'plasma') is the referent of a theory; the theory is the referent of the code; the code's results are valid if the the theory is valid. A host of practical difficulties interfere with this "theoretically principled" view of simulation (Winsberg 2010). Simulations are epistemologically interesting because the knowledge derived from them is often based on *unprincipled solutions* to practical problems. Often these practical problems occur in the confrontation with the material constraints of computational hardware and software. First, a simulation that "follows most straightforwardly from the theoretically principled model" (*ibid.*) will often be far too slow to be useful. Simulators reduce the computational load by simplifying the model or by replacing certain slow, theoretically principled parts of the code with faster, unprincipled approximations. For example, plasma simulations typically include far fewer electrons and ions than a 'real' plasma.

Another set of problems arises from discretization. Discretization is the process of expressing differential equations (which relate continuous rates of change over infinitesimal

intervals) as difference equations (which relate rates of change over finite intervals). A differential equation and a corresponding difference equation are not equivalent to each other. Discretization is necessary for making continuous functions (e.g., those used in many physical theories) calculable on digital computers, however it often introduces artifacts caused by the propagation of rounding errors and 'numerical instabilities' (i.e., unphysical or 'unreal' behaviors). When numerical instabilities are identified, they are sometimes dealt with by introducing "cooked up" simulation components—features that deviate or even go against theoretical principles (Winsberg 2010, 15). The design of these components is informed by a mix of physical intuition and technical expertise—it is *not* characterized by an exclusive commitment to the mathematical model.

Gramelsberger (2011) argues that the move from differential equations to computationally tractable difference equations is a rupture: it "gives rise to divergent symbolic epistemologies of equations and algorithms" (297). Following a detailed analysis of a portion of a climate model responsible for atmospheric precipitation (the `cloud.f90` file), Gramelsberger characterizes this new symbolic form of research: "The design for the choreography of coded procedures is not provided by the mathematical model; it is a matter of 'good modelling' on the part of experienced modelers who draw on observational and experimental knowledge. It is the choreography of coded procedures—imitating the flow of natural processes, e.g. in clouds—that gives the knowledge, employed in the mathematical model in a static way, a new symbolic form" (300).

Gramelsberger's emphasis on the manipulation of code itself ("the choreography of coded procedures") reflects a broader move away from representationalism toward an "artefactual" view of models (including computational models) as "epistemic tools" (Knuuttila 2011; Knuuttila and Boon 2011). In this context, the knowledge derived from simulations is not *explained* by their representational power (i.e., their accuracy or fidelity to a particular theory or phenomena). Instead, it is explained as an achievement of material constraints and the ongoing, active construction and manipulation of those constraints in a research setting. (It is important to note that this is *not* an argument against simulations as representations; it means the knowledge derived from simulations is not justified by a privileged relation-

ship between the simulation and the target. The critique is toward *representationalism*, not representation *per se*). Epistemic tools are "concrete artefacts, which are built by various representational means, and are constrained by their design in such a way that they enable the study of certain scientific questions and learning through constructing and manipulating them" (Knuuttila 2011, 267). In short, the encounter with computational constraints is a central part of the epistemological value of simulation. As explained in the next section, the confrontation with material constraints is at the center of the notion of 'conspicuousness.'

1.2 What is Conspicuous Computing?

My use of the term *conspicuous* is drawn from Dreyfus' (1991) reading of Heidegger. Heidegger described conspicuousness as one kind of disturbance in the relationship between a person and the "equipment" involved in purposeful action. Typically, our relationship to equipment is not a relationship at all (i.e., between *distinct* things that are prior to the relationship). Equipment tends to disappear in our 'use' of it—it is a transparent "something-in-order-to" (*ibid.*, 62) without definite qualities of its own. The classic example is hammering a nail: in the act of hammering, I am not occupied by the determinate characteristics of the hammer or the nail—or, in fact, of myself (*ibid.*, 65). The distinctiveness of things only comes about in the disturbance of the "absorbed coping" that characterizes much of our experience in the world. Conspicuousness is such a disturbance. Dreyfus associates it with moments of malfunction: "When equipment malfunctions [. . .] we discover its unusability by the 'circumspection of the dealings in which we use it,' and the equipment thereby becomes 'conspicuous' " (*ibid.*, 71). We don't start doing ontological work—noticing things with discernible aspects or properties—until we encounter them as a disturbance in our coping with the world.

Heidegger's phenomenology has been influential in Human Computer Interaction (HCI) for theorizing, precisely, 'interactions' between people and computers (Dourish 2004). For Winograd and Flores (1986) Heidegger's account of being in the world provided a philosophical basis for rejecting the "rationalistic tradition" of computing and cognitive science,

which emphasized representations and "detached theoretical understanding" as the basis for relating computation and human intelligence. The transparency of equipment in purposeful action formed a new foundation for thinking about and designing useful tools. In this context, as Dourish notes (2004), the classic example of the hammer and the nail might be replaced by the mouse and the cursor. What the interface designer ought to aspire to is not the production of an accurate representation of the cursor in the mind of the user, but instead, a relationship to the cursor in which the mouse 'disappears' as a discernible entity. (In fact, according to Heidegger, the cursor would also disappear within the broader "in-order-to" that characterizes the "equipmental whole" of absorbed coping. Behind every piece of equipment is another instance of equipment (Dreyfus 1991, 62)).

The point to emphasize is that, in this framing of HCI, conspicuousness (as a disturbance in our absorbed coping) is a *threat* as it breaks the 'flow' of interaction that designers usually hope to sustain in the tools and products they create. Software development and the more formalized discipline of software engineering, as Frabetti (2015) argues, are similarly oriented toward the problem of sustaining software's transparency and instrumentality. The work of identifying user needs, drafting specifications, writing code, debugging, and testing is typically oriented toward the goal of creating an experience of seamlessness in which software 'just works.'

The reason to put a term like conspicuousness at the center of an investigation of scientific software practices is that work from science studies has emphasized the *importance* of disturbance and 'malfunction' in research. "Our everyday notion of an object [...] would seem to contradict the features of objects that scientists and other experts encounter" (Knorr-Cetina 2008, 88). The productivity of conspicuousness has already been noted in the sense that it brings the world to our attention (in fact, it brings it into existence). Rheinberger notes that "[a]s a rule, new developments [in science] are at best an irritation at the point where they first appear" (Rheinberger 1997, 177). The experience of conspicuousness is, therefore, a central feature of scientific work—it is, in a sense, its motive force.

An additional point of reference is Knorr-Cetina's conception of the "unfolding ontology" of epistemic objects. Knorr-Cetina modifies Rheinberger's approach by rejecting the sepa-

ration between epistemic things and technical objects. She replaces it with a tension *within* epistemic objects themselves. Importantly, computers and computer programs are primary exemplars of epistemic objects—they are "both present (ready-to-be-used) and absent (subject to further research)" (Knorr-Cetina 1997, 10). The most important feature of epistemic objects is that they are necessarily incomplete; they "unfold" indefinitely.

> "[T]he defining characteristic of an epistemic object is this changing, unfolding character – or its lack of 'object-ivity' and completeness of being, and its non-identity with itself. The lack in completeness of being is crucial: objects of knowledge in many fields have material instantiations, but they must simultaneously be conceived of as unfolding structures of absences: as things that continually 'explode' and 'mutate' into something else, and that are as much defined by what they are not (but will, at some point have become) than by what they are." (Knorr-Cetina 2008, 89)

Conspicuousness, then, should also be understood as the non-identify of an object with itself, which is important for the emergence of both software and knowledge. To the extent that scientific software matches the trajectory of inquiry itself, it is frequently conspicuous in this sense—in fact, it *must* be so if it is to function as a setting in which new scientific insights emerge.

These observations from science studies closely relate to work from media studies that characterizes media as conditions of understanding (Mitchell and Hansen 2010). Media studies, particularly since the 1960s, has emphasized that media are not ideologically neutral and that media processes are intransitive operations. The schematic of knowledge production that follows from Kittler's claim that "[m]edia determine our situation" (Kittler 1999)—i.e., "media form the infrastructural basis, the quasi-transcendental condition, for experience and understanding" (Mitchell and Hansen 2010, vii)—closely corresponds to Rheinberger's elaboration of scientific objects as graphematic articulations within experimental conditions.[3]

[3]There are important differences between Kittler's media theory and Rheinberger's historical epistemology. Where Kittler, following Foucault and McLuhan, saw history as a series of paradigm shifts between

"Articulations of graphemes, or systems of signification, within the limits of an experimental setting, constitute the objects of science" (Rheinberger 1997, 106). The experience of 'irritation' that characterizes scientific work is fundamentally an experience of the material obduracy (i.e., agency) of systems of signification. This idea is closely related to theories of digital materiality, which emphasize the constraints operative in information technologies (Blanchette 2011; Kirschenbaum 2008). The obduracy of computation is one important source of conspicuousness.

Just as computer technologies trouble the distinction between technical things and epistemic objects—between the conditions of knowledge and the objects of knowledge—media theorists have found that computation violates and supersedes many conventional ideas about media. For example, computers don't *remediate* old media (in the sense of television adopting conventions from film) so much as *simulate* old media—and *all* media, at that (Winthrop-Young 2010). Among the efforts to develop a media theory appropriate to the challenges of computation and software, Chun's (2011) arguments on code as fetish is particularly relevant here as it helps to clarify what Knorr-Cetina described, above, as the unfolding ontology of software. Building on the use of the term in Marx's critique of capitalism and in psychoanalysis,[4] the fetishism of code involves the "false causality" linking code and execution (*ibid.*, 50). A computer is an inherently mysterious technology—its operations elude our perceptual capacities—yet software restores an experience of visibility and control (i.e., *logos*) to the otherwise enigmatic operations of the machine. In this case, to say that code is fetish is not simply to impugn it as ideology—the argument is not that software is the false consciousness that masks the truth of execution in hardware. Rather, the point is that the various dialectics of computation—hardware and software, interface and algorithm, source and execution—are configurations of both visibility and invisibility, knowledge and ignorance. This fact grounds the power and appeal of computation as a 'universal machine.'

media conditions, Rheinberger characterizes the historicity of science as a patchwork of experimental conditions that accommodates both continuity and rupture.

[4]In Marx's critique of capitalism, the commodity-form of value is fetishistic because, through it, relations between people are objectified as relations between things. In psychoanalysis, the fetish is a genital substitute that enables non-reproductive pleasure.

"[The computer's] combination of what can bee seen and not seen, can be known and not known [...] makes it a powerful metaphor for everything we believe is invisible yet generates visible effects" (*ibid.*, 17). A plasma simulation code models plasma, but software as media, stands in for all such processes of standing-in-for, for all contours of knowledge and ignorance. (No matter where plasma physics leads, we *assume* simulation can follow). Software as fetish clarifies how our ignorance of software is an enabling condition. The non-teleological "unfolding" of software is rooted in this aspect.

Noting the tension between the productive and the disruptive aspects of conspicuousness, the intuition that motivates this project is that our understanding of research software benefits from a description of software development that puts *the productivity of disruption* at the center, particularly as an epistemologically relevant term. Given the paucity of approaches to software development as anything other than the iterative implementation computational procedures, the aim of this study is to consider scientific software development as an ongoing encounter with conspicuousness.

1.3 Research Design

This study uses ethnographic and historical methods to describe software development practices in a computational physics lab. In particular, it seeks to elaborate lab members' collective experiences of conspicuousness, as conceptualized above, in order to understand its forms and roles during the co-emergence of software and knowledge in the context of plasma simulation. The central question is whether there are *different kinds* of conspicuousness occurring and interacting in the lab's work of building and using software.

As the preceding discussion demonstrates, my understanding of conspicuousness in rooted in phenomenology. As a 'disturbance' in an individual's relationship with equipment, it is essentially a personal experience. However, to the extent that researchers collectively orient their experiences and expectations (e.g., in a lab meeting discussion of how a simulation code is expected to behave), conspicuousness takes on a specifically 'cultural' aspect, and it becomes available for ethnographic analysis. Further, it is precisely this collective experience

that concerns me here because the problems of knowledge are of a social order (Shapin and Schaffer 1985, 332).

Ethnographic research methods have been widely used to understand the local meanings and significance of software practices in particular settings. These investigations span fields such cultural anthropology (Kelty 2008; Coleman 2013), science studies (Merz 1999), computer-supported cooperative work (Howison and Herbsleb 2011), and empirical software engineering (Segal and Morris 2008). While this work is clearly motivated by a variety of concerns, the general appeal of ethnography for studying software practices is that it provides insight into tacit knowledge and undocumented expertise.

My historical analysis characterizes the genealogies of the epistemic things and technical objects encountered in the field. Plasma simulation codes often involve extensive relations of ideas, actors, objects, and practices that evolve over time through extension, recombination, and disassociation (Akera 2007). In these genealogies the histories of computing (particularly high performance computing) and plasma physics are intimately related. Analyzing these transformations can lead to greater understanding of the dynamics through which epistemic things and technical objects relate to each other (e.g., in the case of the "charge-conserving current deposit" discussed in chapter four).

The choice to study a computational physics lab was informed by several criteria. First, the software developed at the site should be directed toward a wider scholarly community—it shouldn't be an interim step of purely local significance and use. Additionally, software development should be a central, established, and ongoing activity at the field site. These criteria were intended to identify established projects that are more likely to have deeply rooted understandings of how software development and scholarly work relate to each other. An additional appealing feature of plasma simulation is its extensive history, which provides an opportunity to study changes in software practices over time. For example, recent expectations from funding agencies that software should be made publicly available significantly affected the lab's relationship to its simulation codes (as discussed in chapter three).

Over the course a year (2015-2016), I attended weekly lab meetings (43), conducted

semi-structured interviews (14 in total, lasting about an hour each), and collected literature relevant to the labs' ongoing work. I also attended a three-day workshop, in which users and developers of one of the lab's more widely used codes met to discus new feature and future developments. In addition, I was given limited access to the lab's version control system through which I was able to review source code, read documentation, and observe ongoing discussions associated with the development work.

A condition of my access at the field site was that I would protect the identities of participants and that of the lab itself. All names used in this dissertation are aliases. I refer to the lab simply as 'the lab.'

1.4 The Site

The field site was a computational physics lab that specialized in plasma simulation. Before describing the setting in more detail, let me address a more basic question: What is a plasma and why might physicists be interested in simulating them[5]? Plasma is gas composed of charged particles (electrons and ions) interacting through electromagnetic forces. Colloquially it is "gas that glows." Plasma occurs in an immense variety of natural and artificial contexts. In fact, most of the matter in the the universe (observable matter) is expected to exist as plasma in the vast space between galaxies—the so-called intergalactic medium. Plasmas are often associated with extreme environments that are hard to interact with: the insides of nuclear reactors, the Earth's ionosphere, and stars, for example. This feature partly explains the popularity of simulation in plasma physics. Further, plasma behavior is inherently complex because the constituent electrons and ions are affected by the electromagnetic field of the plasma, which is affected, in turn, by the moving particles. As a result of this feedback and collective influence, traditional analytic techniques are not well suited to the task of describing plasmas. While the fundamental electromagnetic laws governing plasma phenomenon are very well understood (Maxwell's equations provide the main analytical description), a "full" description of a plasma would require solving a set of

[5]Like "code", physicists sometimes to treat "plasma" as a countable noun rather than a mass noun.

differential equations for each plasma particle—a computationally intractable problem, even in 'simple' cases (Verboncoeur 2005). Computer simulation plays a prominent role in plasma physics because it offers an alternative to traditional analytic techniques and because many plasma phenomena of interest are difficult or impossible to create in the lab or to observe in nature (Dawson 1983).

The plasma simulation lab was located in a US research university; it consisted of five senior faculty (some with positions in a partnering institution), several postdocs, and a handful of graduate students—fourteen people in all (though this number fluctuated some over the course of my fieldwork). The lab also hosted several visiting researchers. The simulationists' workweek was punctuated by two lab meetings, on Monday and Friday mornings. Meetings were held in a spacious conference room in the university physics department. In most respects the Monday and Friday meetings were the same. If there was no scheduled talk or pressing issue to discuss, as was often the case, meetings were a time for the head of the lab, Bruce, to receive updates and generally coordinate the lab's ongoing projects. The meetings differed thematically, however, in that Monday's were "about physics" and Friday's were "about code development."

A typical Monday meeting might include a practice talk by a graduate student or postdoc for an upcoming conference, or a less formal presentation of work in progress. Presentations followed a fairly common structure. First, there was the matter of the research area and the motivation for the work. Two areas of research dominated much of the lab's work: fusion energy research and plasma-based particle acceleration. In the former, simulations are used to study operational conditions of fusion energy devices (particularly inertial confinement fusion); the latter concerns the design of next generation particle accelerators, such as those used in high energy physics. Additional areas of research included astrophysical phenomena and plasma-based propulsion.

Lab meeting presentations typically included a review of numerical techniques relevant to the physics of interest—here the simulationists emphasized computational constraints in their evaluation of numerical approaches. Plasma simulation involves difficult tradeoffs

between processing time and detail in the simulated plasma.[6] Unlike general purpose computing, plasma simulations (and other kinds of high performance scientific computing) can easily consume the processing capacities of the largest supercomputers. Whereas general software developers often choose to sacrifice some performance for greater compatibility, simulationists are rarely willing to make such a sacrifice—indeed, plasma codes are often tailored to *particular* supercomputers, such as those managed by the National Energy Research Scientific Computing Center at Lawrence Berkeley National Lab. (Managing the computer time, measured in CPU hours, allocated to the lab by national computing centers was among the routine tasks of lab meetings).

The central objects of discussion in Monday's presentations were often the images (sometimes animations) derived from simulation data. Plasma simulation codes produce enormous datasets that are of little direct value; the value of the data only emerges with subsequent analysis and visualization. The interpretation of simulation results typically entailed visual comparison of plots based on different data sources—simulations were compared to other simulations (i.e. alternative numerical models), to experimental results, and to theory. In the short, one of the dominant activities of Monday meetings was the collective consideration and discussion of visual representations of plasmas, produced by various means. While most members of the lab were present for these discussions, and many participated, Bruce was the primary interlocutor. As the head of the lab, Bruce closely managed the researchers' work, particular the graduate students'. Presentations typically concluded with Bruce's suggestions for improving the simulation setup, or sometimes with a closing remark: "I'm going to have to think about this." (A specific example of such a discussion is included in chapter two).

A somewhat different social ordering was on display in Friday's meetings, which focused on technical aspects of simulation code development. Like the twin lab meetings, the simulationists tended to differentiate themselves between those who "just do physics" and those

[6]For example, "spectral" codes use the Fast Fourier Transform (FFT) algorithm to solve the Poisson equation with very high precision. However, the FFT algorithm can impose high communication overhead between nodes of parallel computers; as a result, simulationists sometimes prefer less precise techniques that are more accommodating to parallel processing.

who mainly worked on code development. (To be clear, everyone in the lab was a credentialed physicist or was working toward such a credential). For example, Bruce was known as someone who just did physics: some lab members speculated, more in amazement than disparagement, that Bruce had never touched a line of code in his long and prestigious career as a plasma simulationist. Other senior members of the lab were known mainly for their development work—they "really [didn't] do physics any more" as their time was significantly devoted to maintaining and improving the lab's codes. These "super technicians" (Latour and Woolgar 1986) often received credit for their work as co-authors on papers in which their codes were used. Trevor, a veteran of plasma simulation with over forty years of experience in high performance scientific computing, was one such figure. Friday meetings were times for people like Trevor (and sometimes more technically inclined graduate students) to present tutorials and demonstrations of tools and techniques. The topics of discussion included code documentation practices, tools for testing Fortran codes, new frameworks for developing new codes, and general planning of ongoing code work.

Plasma simulation codes vary widely. They are differentiated by their numerical approaches, the computer architectures they are optimized for, and the research communities they serve (among other considerations). The lab specialized in the development of Particle-in-Cell (PIC) codes, considered to be one of the most detailed simulation techniques because they model plasma at the particle level. PIC codes are among the most computationally demanding plasma models. The majority of the lab's codes were written in Fortran, one of the oldest high-level programming languages, now a mainstay in high performance computing (HPC). While Fortran might be considered an obsolete language in the world of general-purpose software development, simulationists (and other computational scientists) continue to use the language because of the extensive body of legacy scientific programs written in Fortran (some plasma codes have been in continuous development for decades) and because the programming language is very well suited to large-scale numerical computation. As the simulationists explained it, it takes less effort to write a high performance plasma code in Fortran than it does to write an equally performative code in another language (like C or C++).

The Friday meeting was a recent addition to the lab's weekly schedule. As I describe in chapter three, it was introduced as a response to changing norms in scientific software development. The lab was several decades old at the time of my fieldwork, and for much of its history code development was not directly supported by funding agencies. Research was funded but not code development itself. This has changed in recent years, particularly through initiatives associated with cyberinfrastructure and open science policies. Funding agencies increasingly expect that funded software will be made publicly available (i.e., with an open source license). The changing status of research software as a more recognized form of scholarly production was experienced by members of the lab as pressure to adopt 'open' and 'professional' approaches to software development. Bruce's comment in one of the meetings—"we have to start thinking about these [codes] as products"—captures the transformation in the lab's approach to software development. One of the ongoing discussions I observed in lab meetings concerned the question how (and whether) the lab should release its codes and expand the community of users around it them. This aspect is discussed in chapter three.

1.5 Overview of the Dissertation

The term 'conspicuousness' is, if you will, conspicuously absent in the three main chapters that make up this dissertation. I return to the theme of conspicuousness in the conclusion. Here I will sketch the central chapters by relating their arguments to the concern of conspicuousness.

The second chapter conceptualizes the lab's simulation development as a memory practice (Bowker 2005). Memory practices concern the ongoing production of a useful past; the kind of past that simulationists require is one that provides a structure of expectation—a way to differentiate between productive and regressive encounters with the unexpected. The form of conspicuousness that I consider in this chapter is the experience of the unexpected. I discuss the memory practices of cybernetics and software engineering to illustrate distinct, historically specific strategies for managing the unexpected behavior of complex systems. Further, I illustrate how lab members managed the unexpected behaviors of their codes, and

how this work constitutes the temporal coherence of simulation development. The 'productivity' of the unexpected is a local accomplishment and it is not teleological. This conclusion has implications for contemporary efforts to formalize scientific software development as an ongoing, cumulative activity.

Conspicuousness is not only a local experience. The norms associated with scientific software development—such as the contemporary expectation of openness—are evolving. Chapter three concerns the lab's ambivalence to prevailing norms of research software development associated with cyberinfrastructure and open science policies. These policy initiatives emphasize that code should move beyond its community of developers (i.e., as shared 'infrastructure' or reproducible code). While the lab adopted certain aspects of open source development, its most well-known codes remained closed. I use the notion of a user-developer regress to explain the lab's reluctance to fully embrace open access policies. A user-developer regress arises when the value of software is related to the potential for code to move beyond its community of developers, yet the expertise needed to use the code is stubbornly fixed to its community of developers. The central argument of the chapter is that the vision of research software offered by funding agencies does not coincide with the lab's (more conspicuous) relationship to software (i.e., in which codes are acknowledged as unfolding and provisional).

In chapter four, I differentiate five categories of simulation code developed in the lab: production codes, reference codes, frameworks, classroom codes, and toy codes. The axis of difference that this categorization describes (which cuts across rather than along divisions between fields of study) resembles Rheinberger's (1997) distinction between epistemic things and technological objects. I argue that production codes fit within Merz's conception of simulations as "multiplex and unfolding." However the intentions associated with reference codes, frameworks, and classroom codes are more aligned with the role of technical objects. Whereas the trajectory of production codes closely matches the research process, the latter categories involve practices such as classroom demonstration, benchmarking, and the rapid development of new codes. Toy codes stand out as a difficult and mercurial concept, which suggests they are also multiplex, though in a manner distinct from production codes. The

implication is that reference codes and frameworks, in particular, correspond to efforts to stabilize plasma simulation techniques as technical objects.

1.6 Bibliography

Akera, Atsushi. 2007. "Constructing a Representation for an Ecology of Knowledge Methodological Advances in the Integration of Knowledge and Its Various Contexts." *Social Studies of Science* 37 (3): 413–41. doi:10.1177/0306312706070742.

Blanchette, Jean-François. 2011. "A Material History of Bits." *Journal of the American Society for Information Science and Technology* 62 (6): 1042–57. doi:10.1002/asi.21542.

Borgman, Christine L. 2012. "The Conundrum of Sharing Research Data." *Journal of the American Society for Information Science and Technology* 63 (6): 1059–78.

Bowker, Geoffrey C. 2005. *Memory Practices in the Sciences*. MIT Press.

Chun, W.H.K. 2011. *Programmed Visions: Software and Memory*. MIT Press.

Coleman, E. Gabriella. 2013. *Coding Freedom: The Ethics and Aesthetics of Hacking*. Princeton University Press.

Collins, Harry. 2010. *Tacit and Explicit Knowledge*. University of Chicago Press.

Collins, Harry, and Robert Evans. 2008. *Rethinking Expertise*. University of Chicago Press.

Daston, Lorraine, and Peter Galison. 2007. *Objectivity*. Zone Books.

Dawson, John M. 1983. "Particle Simulation of Plasmas." *Reviews of Modern Physics* 55 (2): 403–47. doi:10.1103/RevModPhys.55.403.

Dourish, Paul. 2004. *Where the Action Is: The Foundations of Embodied Interaction*. MIT Press.

Dreyfus, Hubert L. 1991. *Being-in-the-World: A Commentary on Heidegger's Being and Time, División I*. MIT Press.

Drucker, Johanna. 2009. *SpecLab: Digital Aesthetics and Projects in Speculative Computing*.

University of Chicago Press.

Easterbrook, Steve M., and Timothy C. Johns. 2009. "Engineering the Software for Understanding Climate Change." *Computing in Science & Engineering* 11 (6): 64–74. doi:10.1109/MCSE.2009.193.

Edwards, Paul N., Steven J. Jackson, Melissa K. Chalmers, Geoffrey C. Bowker, Christine L. Borgman, David Ribes, Matt Burton, and Scout Calvert. 2013. "Knowledge Infrastructures: Intellectual Frameworks and Research Challenges."

Fletcher, Pamela, and Anne Helmreich. 2012. "Local/Global: Mapping Nineteenth-Century London's Art Market." *Nineteenth-Century Art Worldwide*, 1850–1939.

Frabetti, Federica. 2015. *Software Theory: A Cultural and Philosophical Study.* Rowman & Littlefield International.

Goble, Carole. 2014. "Better Software, Better Research." *IEEE Internet Computing* 18 (5): 4–8.

Gramelsberger, Gabriele. 2011. "What Do Numerical (Climate) Models Really Represent?" *Studies in History and Philosophy of Science Part A*, Model-based representation in scientific practice, 42 (2): 296–302. doi:10.1016/j.shpsa.2010.11.037.

Howison, James, and James D. Herbsleb. 2011. "Scientific Software Production: Incentives and Collaboration." In *Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work*, 513–22. CSCW '11. New York, NY, USA: ACM. doi:10.1145/1958824.1958904.

Karasti, Helena, Karen S. Baker, and Eija Halkola. 2006. "Enriching the Notion of Data Curation in E-Science: Data Managing and Information Infrastructuring in the Long Term Ecological Research (LTER) Network." *Computer Supported Cooperative Work (CSCW)* 15 (4): 321–58. doi:10.1007/s10606-006-9023-2.

Kelty, Christopher M. 2008. *Two Bits: The Cultural Significance of Free Software.* Duke

University Press.

Kirschenbaum, Matthew G. 2008. *Mechanisms: New Media and the Forensic Imagination*. MIT Press.

Kittler, Friedrich A. 1999. *Gramophone, Film, Typewriter*. Stanford University Press.

Knorr-Cetina, Karin. 1997. "Sociality with Objects Social Relations in Postsocial Knowledge Societies." *Theory, Culture & Society* 14 (4): 1–30. doi:10.1177/026327697014004001.

———. 2008. "Objectual Practice." In *Knowledge as Social Order: Rethinking the Sociology of Barry Barnes*. Vol. 83.

———. 2009. *Epistemic Cultures: How the Sciences Make Knowledge*. Harvard University Press.

Knuuttila, Tarja. 2011. "Modelling and Representing: An Artefactual Approach to Model-Based Representation." *Studies in History and Philosophy of Science Part A*, Model-based representation in scientific practice, 42 (2): 262–71. doi:10.1016/j.shpsa.2010.11.034.

Knuuttila, Tarja, and Mieke Boon. 2011. "How Do Models Give Us Knowledge? The Case of Carnot's Ideal Heat Engine." *European Journal for Philosophy of Science* 1 (3): 309. doi:10.1007/s13194-011-0029-3.

Knuuttila, Tarja, and Atro Voutilainen. 2003. "A Parser as an Epistemic Artifact: A Material View on Models." *Philosophy of Science* 70 (5): 1484–95. doi:10.1086/377424.

Latour, Bruno. 2013. *An Inquiry Into Modes of Existence*. Harvard University Press.

Latour, Bruno, and Steve Woolgar. 1986. *Laboratory Life: The Construction of Scientific Facts*. Princeton University Press.

Lynch, Clifford. 2014. "The Next Generation of Challenges in the Curation of Scholarly Data." In *Research Data Management: Practical Strategies for Information Professionals*, edited by Joyce M. Ray, 395–408. West Lafayette, Indiana: Purdue University Press.

Mayernik, Matthew, Tim DiLauro, Ruth Duerr, Elliot Metsger, Anne Thessen, and G.

Choudhury. 2013. "Data Conservancy Provenance, Context, and Lineage Services: Key Components for Data Preservation and Curation." *Data Science Journal* 12 (0). doi:10.2481/dsj.12-039.

Merz, Martina. 1999. "Multiplex and Unfolding: Computer Simulation in Particle Physics." *Science in Context* 12 (02): 293–316. doi:10.1017/S0269889700003434.

Mitchell, W. J. T., and Mark B. N. Hansen, eds. 2010. *Critical Terms for Media Studies*. University of Chicago Press. doi:10.7208/chicago/9780226532660.001.0001.

Parker, Wendy. 2013. "Computer Simulation." In *The Routledge Companion to Philosophy of Science*, edited by S. Psillos and M. Curd, 2nd Edition, 135–45. London: Routledge.

Pickering, Andrew. 1995. *The Mangle of Practice: Time, Agency, and Science*. University of Chicago Press.

Rheinberger, Hans-Jörg. 1997. *Toward a History of Epistemic Things: Synthesizing Proteins in the Test Tube*. Stanford, Calif.: Stanford University Press.

Robinson, Hugh, Judith Segal, and Helen Sharp. 2007. "Ethnographically-Informed Empirical Studies of Software Practice." *Information and Software Technology* 49 (6): 540–51. doi:10.1016/j.infsof.2007.02.007.

Segal, Judith. 2005. "When Software Engineers Met Research Scientists: A Case Study." *Empirical Software Engineering* 10 (4): 517–36. doi:10.1007/s10664-005-3865-y.

Segal, Judith, and Chris Morris. 2008. "Developing Scientific Software." *IEEE Software* 25 (4): 18–20. doi:10.1109/MS.2008.85.

Shapin, Steven, and Simon Schaffer. 1985. *Leviathan and the Air-Pump*. Princeton University Press Princeton, NJ.

Sismondo, Sergio. 2010. *An Introduction to Science and Technology Studies*. 2nd Edition. Wiley.

Spencer, Matt. 2015. "Brittleness and Bureaucracy: Software as a Material for Science."

Perspectives on Science 23 (4): 466–84. doi:10.1162/POSC_a_00184.

Stodden, Victoria, Friedrich Leisch, and Roger D. Peng. 2014. *Implementing Reproducible Research*. CRC Press.

Sundberg, Mikaela. 2009. "The Everyday World of Simulation Modeling the Development of Parameterizations in Meteorology." *Science, Technology & Human Values* 34 (2): 162–81.

Verboncoeur, J. P. 2005. "Particle Simulation of Plasmas: Review and Advances." *Plasma Physics and Controlled Fusion* 47 (5A): A231. doi:10.1088/0741-3335/47/5A/017.

Wing, Jeannette M. 2006. "Computational Thinking." *Communications of the ACM* 49 (3): 33–35.

Winograd, Terry, and Fernando Flores. 1986. *Understanding Computers and Cognition: A New Foundation for Design*. Intellect Books.

Winsberg, Eric. 2010. *Science in the Age of Computer Simulation*. University of Chicago Press.

———. 2018. "Computer Simulations in Science." In *The Stanford Encyclopedia of Philosophy*, edited by Edward N. Zalta, Summer 2018. Metaphysics Research Lab, Stanford University.

Winthrop-Young, Geoffrey. 2010. "Software/Hardware/Wetware." In *Critical Terms for Media Studies*, edited by W. J. T. Mitchell and Mark B. N. Hansen. University of Chicago Press. doi:10.7208/chicago/9780226532660.001.0001.

CHAPTER 2

Simulation Development as a Memory Practice

2.1 Abstract

This chapter explores the conceptual foundations and problems of software in the scholarly record by considering software development as a scientific memory practice. I discuss the memory practices of cybernetics and software engineering to illustrate distinct, historically specific strategies for managing the unexpected behavior of complex systems. Building on Frabetti's deconstructive reading of software engineering and Rheinberger's portrayal of the temporality of experimental systems, I characterize the coherence of plasma simulation development as an effort to make software's unexpected behavior productive. The productivity of the unexpected in plasma simulation is a local accomplishment and it is not teleological. This conclusion has implications for contemporary efforts to formalize scientific software development as an ongoing, cumulative activity.

2.2 Introduction: A Scientific Software Literature

From its start, software has been a part of science. Programming as a kind of writing—as the production of a text, written in a 'high-level' programming language, that can be transformed or compiled into machine code—emerged in the context of scientific computing as an effort to unify the notational practices of science on the one hand and computation on the other. This is evident in the case of the programming language, Fortran, developed in IBM's Applied Sciences Division in the mid 1950s. Now considered to be among the first high-level programming languages, Fortran was conceived as a "Mathematical Formula Translating

System." Fortran is significant not only for its remarkable durability as a mainstay of scientific computing (Decyk, Norton, and Gardner 2007), but also because it helped put to rest debates over whether programs compiled from high-level languages can make the same efficient use of computational resources as 'hand coded' programs (programs written directly in machine code). Its principal architect, John Backus, reflected in 1979, "[i]t was our belief that if FORTRAN, during its first months, were to translate any reasonable 'scientific' source program into an object program only half as fast as its hand coded counterpart, then acceptance of our system would be in serious danger" (Backus 1998, 70). What is significant here is that Backus measured Fortran's success, at least in part, in terms of the language's ability to incorporate qualities of two kinds of scientific writing: the intuitive qualities of mathematical formulae, as the 'natural' language of science, and the performative qualities of "hand coded" scientific programs. There is an evident asymmetry in these forms of writing, however. Scientific programming clearly lacked the aspect of mathematical notation that constitutes a "tradition" of formal thought (Derrida 1978). Scientific programs may have been scientific texts, but they did not yet—and perhaps still do not—constitute a 'body' of thought.

By the 1960s, computational scientists had started to recognize software's precarity in the scientific record. In a prescient article, "The Publication of Scientific Fortran Programs", Roberts (1969) noted that "[a] great deal of scientific effort is now being spent on the construction of computer programs [...], but much of this effort is wasted at present because the average program has a very limited range of application and is quite difficult for anybody except the author or his [sic] close colleagues to understand or to use." Roberts looked to scholarly publishing—"a remarkably efficient mechanism for ensuring the continuous growth and dissemination of a body of universal and accurate information"—as a model for guiding the production, evaluation, and distribution of scientific programs. He further outlined a set of recommended programming principles, covering aspects such as modularity and documentation, that he argued would make scientific programming a more cumulative activity. In later writing, Roberts (1971) suggested systems of "open publication" and the establishment of "a scientific software literature" to supplement traditional journal publications. "The ex-

istence of a high-quality open scientific program literature," Roberts argued, "should serve as a stimulus to the whole computing industry, just as the regular scientific and mathematical literature of books and journals does for technology."

Despite this long history of scientific programming, only recently has something like Roberts' vision of a "scientific software literature" managed to take hold among practitioners in scientific computing.[1] In recent years, efforts on a variety of fronts have converged on a set of practices, values, and debates oriented toward the problem of elaborating scientific software as a coherent body of practice. Included within this area of concern, which I broadly characterize as the formalization of research software development, are efforts such as: training scientists to write better, more sustainable software (cf. UK's Software Sustainability Institute and the Software Carpentry initiative); efforts to ensure that research software developers receive *credit* for their contributions, particularly through citation (cf., the Journal of Open Research Software); efforts to *fund* research software development; and efforts to promote the *integrity* of scientific software (software reproducibility); efforts to collect, preserve and *curate* scientific software. This constellation of practices is closely related to the discussions of 'open science' and knowledge infrastructure (cyberinfrastructure) discussed in chapter three. What unifies these various projects is a concern for norms and forms of scholarly software development. In short, while software has long be produced in science (for as long as there has been software), we can recognize these discussions as steps to formalize software as a scientific product.

If software development is emerging as an increasingly formalized area of scientific practice, then what kind of 'memory' does it constitute? Where, for example, does software stand in conceptual relation to the notion of the 'scientific record', a term traditionally associated with the accumulated knowledge embodied in scientific texts? My aim in this chapter is to explore the conceptual foundations and problems of software in the scholarly record by considering software development as a scientific memory practice (Bowker 2005). In particular,

[1]There are few instances of what we might now call 'software curation' from the late 1960s. One notable example: the journal, *Computer Physics Communication*, working in association with the Queen's University in Belfast, established the International Physics Program Library in 1969. (It is now known as Computer Physics Communications Program Library).

I will focus on simulation development as a central form of scientific software development. Memory practices concern the technical, social, and formal mediation of the past. They constitute the past as something *useful* in the present. Roberts' hope of fashioning scientific software development into a more cooperative, cumulative activity by establishing a "scientific software literature" is an argument for such a practice. However, it remains to be considered whether the analogy to literature (perhaps it is more than an analogy) appropriately describes software's status as scientific memory. What kind of memory is constituted in software?

To quickly sketch the terrain of the problem, consider different temporalities and relationships to the past associated with two familiar conceptions of software: software *as tool* and software *as text*. Berry (2011) introduces the tool/text opposition through his distinction between an external, user-oriented perspective of software—one that calls for distant reading of *applications*—and an internal, design oriented perspective that calls for close reading of *code*. We can also relate these terms to distinct idioms of scientific practice: performativity and representationalism, respectively[2] (Pickering 1995). In a performative mode, software's tool-like aspects are most prominent. Here, software is a source of material agency the contours of which are not necessarily known in advance (to the user or the developer). The performative 'understanding' of a tool is not representational in nature—it is not derived from descriptions of the tool or analytical investigations of the tool. It is a "real-time" understanding that emerges through interaction and use. Further, the refinement of a tool is iterative in nature. It involves a process of "tuning", to use Pickering's analogy of the radio

[2]The distinction between the representationalist and performative idioms, as characterized by Pickering (1995), has been significantly troubled by more recent work in science studies. For example, Barad's (2007) "agential realist" account of science (in fact, it is an entire metaphysics) is explicitly performative and explicitly anti-representationalist, however the contours of these terms do not follow Pickering's. As Barad notes, Pickering's account of performativity "ignores important discursive dimensions of scientific practices, including questions of meaning, intelligibility, significance, identify formation, and power, which are central to poststructuralist invocations of performativity and feminist accounts of technoscientific practice" (Barad 2007, 411n18). Performativity, for Barad, refers not just to situated interaction with non-human agencies ("tuning"), but to the observation that subjects and objects are actively produced through discursive practices. In this view, all scientific practices, *including* the production and use of representations, are performative—which is to say, representational work brings things into being. Like Pickering, Barad's concern with representation is not representation *per se*, it is *representationalism*, "the belief in the ontological distinction between representations and that which they purport to represent"; "that which is represented is held to be independent of all practices of representing" (Barad 2007, 46).

dial. Performativity also entails a particular relationship to the past: through refinement and tuning, traces of past events inhere in tools. Thus, the 'memory' associated with them resides in the temporal continuity of practices in which they are enrolled. Following Bowker, it is a form of non-conscious memory. This memory becomes recognizable through the analytic framework of material culture, or through an archaeology of effective techniques. On the other hand, a representational understanding of a tool, in which it is approached as a object of analysis and reflective thought, is only required in moments of breakdown, when the tool ceases to be effective *as a tool*.

Software's textuality—and software development's emergence as a kind of literary practice (Knuth 1984)—is an historically specific response to the necessity and difficulty of understanding software in a representational mode. This argument will be presented in more detail below, but the broad idea is that the field of software engineering emerged from the perceived unsuitability of the performative approach in managing risks associated with large, complex systems. To make software broadly reliable and usable, it needed to be made legible and communicable (again, recall Roberts' argument for a software literature). Through the formal techniques of software engineering, the textual forms of software development multiply. We encounter a system of documents (specifications, source code, bug reports) all of which contribute to the understanding of a program. The temporal emergence of software (as text) is iterative, but not in a performative sense. Rather, it involves a cycle of repeated *description* through specification and *implementation* in source code. Software engineering thus constitutes software as a representational object, and software development as a representational activity. To summarize the distinctive relationships to memory of software-as-tool and software-as-text, in the former software is not made *as* memory (it is just *technics*), while in the latter there is an overt relationship to memory (it is *mnemotechnics*) (Frabetti 2015).

The distinction between the performative (tool-like) and representational (text-like) aspects of software need not be drawn sharply, once and for all. I am not aiming at a metaphysical argument about software's 'nature.' Rather, I regard the relationship between performative and representational idioms as an evolving contour and part software's histor-

ical specificity (cf. Mackenzie's (2006) argument that contemporary software methods have a more performative aspect). In fact, I will argue that to fully appreciation software development as a memory practice, we need to do away with the apparent opposition between these terms.

My argument in this chapter is as follows. The formalization of research software development has involved the adoption of norms and strategies from software engineering, a field dedicated to the systematization of the software development process. Advocates of research software "sustainability," for example, frequently draw on normative frameworks from software engineering (Goble 2014). Following Frabetti's (2015) deconstructive reading of software engineering's foundational texts, I characterize the field as a collection of inscriptional practices built atop several internal contradictions—contradictions that constitute the field as a domain of practice. A central tension in software engineering is that developers' ignorance of the systems they create is both a *threat* to be minimized and an essential resource that *propels* the developers' work. Software engineering aims to eliminate the unexpected from the work of programming, but it is also oriented toward the productive potential of the unexpected (e.g., through creative leaps or surprising shifts in how a system is used). The double role of the unexpected—as progress and regress—has particular significance in the case of scientific software, as it is an important basis for software's epistemological value. Scientists would not need simulations, for example, if they knew exactly how they would behave in advance.

The kind of memory that simulation developers require is one that allows them to make the unexpected behaviors of their codes productive. As a memory practice—as an activity of constituting a past that is useful—simulation development involves the production and *incorporation* of unexpected events in software. A simulation code is an object nobody entirely understands—it is "epistemically opaque" (Humphreys 2009)—however its behavior is never wholly unexplainable. It is a crafted source of formalized ignorance and material agency *from which* new understandings might emerge. The unexpected events that the code generates beg to be explained. They may be incorporated (i.e., committed to the past) as refinements to the code, as changes in an embodied understanding of the code,

or as published 'results.' To be productive, they need to be re-presented. The difficult negotiations associated with the use of simulations happen between the occurrence of the unexpected and its incorporation within a space of representation. That is when quick hacks are performed to make the unexpected thing go away, when people are sought who may be familiar with the unexpected thing, when colleagues are made to recognize that the thing is (in fact) unexpected, and when blame is placed on those who caused the unexpected thing. That is *also* when credit and priority are given to the people who 'discovered' the unexpected thing. In short, that is the moment in which memory practices operate. As Rheinberger noted in relation to experimental systems, "the new is not the new at the beginning of its emergence" (1997, 185). The events subsequently regarded as progress or regress do not initially appear as such, but they do appear within representational systems that are performatively reinforced and transformed in the effort to discern those events as either progress or regress.

The elaboration of scientific software development as a memory practice provides a conceptual foundation for understanding contemporary efforts to formalize software development as scientific practice. In particular, it clarifies software's status within the scientific record. This has implications for emerging knowledge management practices in digital scholarship such as software preservation and curation.

This chapter relates three main texts—Bowker's (2005) conception of memory practices, Frabetti's (2015) deconstructive reading of software development, and Rheinberger's (1999) articulation of the temporal structure of experimental systems—to conceptualize the temporal arrangements through which research software emerges as a form of knowledge. After outlining key features of memory practices in the first section, I discuss the memory practices of cybernetics and software engineering to illustrate distinct strategies for managing "the unexpected" within complex systems. Cybernetics and software engineering are important as two historically specific responses to the kinds of complexity that emerge in programming. I conclude by outlining some of the ways simulations emerge through encounters with the unexpected. While the discussion aims toward a degree of generality, I introduce specific scenarios from plasma simulation to illustrate these dynamics.

2.3 Memory Practices

Bowker's (2005) *Memory Practices in the Sciences* is significant for its elaboration of scientific memory in terms of technologies for information processing and formal systems that structure the scientific record. Bowker provides an understanding of scientific memory as a technologically mediated practice, against which we can consider the case of software development, generally, and simulation development in particular. Straightforwardly, memory practices concern useful descriptions of the past. They are activities, processes, strategies, and techniques through which such descriptions are produced, made use of, and carried into the future. The subtleties of memory practices emerge by considering the full scope and spectrum of ways that *a useful past* has been conceptualized and materially constituted. As this suggests, memory practices are historically specific and they are intimately related to technical conditions.

To illustrate, consider the different kinds of 'records' associated with laboratory notebooks, on the one hand, and published accounts of laboratory experiments on, on the other. As Shankar (2007) notes through a study of scientists' personal record-keeping practices in the lab, the scientific record emerges through the production of documents that are both personally meaningful to the individual scientist and broadly reliable for the purposes of the scientific community. The end result is "a document that is paradoxically wholly personal and yet intrinsically professional." Published accounts, on the other hand, are widely recognized as *idealized* versions of what happened during a scientific investigation. This idealization has a purpose. As Bowker notes: "Scientific [publications] are written not to record what actually happened in the laboratory, but to tell the story of an ideal past in which the protocols were duly followed: the past that is presented should be impregnable [...]. It takes a great deal of hard work to erect a past beyond suspicion" (Bowker 2005, 7). The point of the comparison between lab notebooks and publications is not to uphold the latter as a white-washed fraud and the former as the messy reality. Rather, the idealized report and the internally-held institutional memory of the lab notebook correspond to distinctive memory practices. They constitute the past as a useful resource in different ways, however, their real

value emerges through their articulation. Memory practices are not isolated collections of activities. Collectively they form "memory regimes" to create a shared—and to a certain degree *unitary*—conception of a "continuous, useful past."

Common technologies and formal tools for recording, structuring, and processing information are conditions of possibility for relating memory practices to one another and for forming memory regimes. As Shankar notes, laboratory notes must be personally meaningful to the scientist but they must also lend themselves to becoming "enmeshed in larger meanings that dovetail with expectations of the broader scientific community" (Shankar 2007). The "poetics and pragmatics" of laboratory recordkeeping resides in the personalized strategies for negotiating these twin demands. Bowker further expands the scale of the commensurability afforded by common formal systems to the relationship between disciplines. Interdisciplinarity, Bowker suggests, is possible to the extent that disparate fields share a common knowledge infrastructures within which research objects are constituted. "We do not as a society have a series of separate and separable discourses about the past [...]. Rather, I would argue, the boundaries between disciplines are porous precisely because the same kind of information-processing technology is being used in each case" (Bowker 2005, 136). (This is not a deterministic argument, Bowker argues, because the history of information processing cannot be isolated from the 'social' and the 'cultural.')

The central "material substrate" of contemporary memory, and thus the ultimate material condition of commensurability in the present, is the digital computer. The primary form of computational information processing that Bowker considers is the database. (The emphasis on data storage and data organization is understandable considering the fields that Bowker focuses on—biological sciences, bioinformatics, and biodiversity—fields heavily oriented toward the representation of the past and present of life on Earth). Software and software development receive relatively little attention in this elaboration of scientific memory. The computer program is mainly discussed as a piece of "ideal machinery"—as a metaphor for understanding reality (e.g., the universe is a simulation) that structures our conception of the past.

A crucial aspect of memory practices is that they are not strictly concerned with the *con-*

scious recovery of the past (i.e., in the mind of an individual). Bowker associates forms of non-conscious memory with rules and constraints that limit interpretation and action. Rules or constraints are forms of externalized memory. Technical forms—particularly classification systems and standards—are central features of memory practices in this respect. Classification systems constitute descriptive matrices, limited ontological spaces that constrain what can and cannot be named and remembered. The basic thrust of memory practice, as a way of thinking about science, is that technologies for recording and processing information have a constitutive power derived from the constraints they impose on scientists' objects of study. This power is 'onto-epistemic' in nature, as the distinction between the object produced in a technologically produced space of representation is not distinguishable from the object of knowledge. I return to this point below in my discussion of Rheinberger's understanding of the materiality of experimental systems as "graphematic articulation" (Rheinberger 1997, 106). The primary object that Bowker is concerned with is 'the past'. Importantly this is both the past as an object of study (e.g., the Earth's past) and the past of sciences' efforts to elaborate that object.

Of the historical memory regimes that Bokwer considers, the case of cybernetics warrants special considerations for several reasons. First, histories of computing and cybernetics are intimately related, so it is worth looking to the latter to understand something of the former. Cybernetic systems are also closely related to simulations—they are both closed systems in which complex behaviors emerge through feedback. As a result, the way cyberneticists conceptualized memory and its role in the study of complex systems serves as a useful point of comparison for thinking about contemporary simulation practice. As we will see, however, the memory practices of cybernetics are historically distinct. They have largely been interpreted as an example of knowledge production in a performative idiom (Pickering 2010).

2.4 The Performative Memory of Cybernetics

Cybernetics is a sprawling, transdisciplinary field oriented toward the study of 'general' systems, systems in which the human-machine distinction is eroded or entirely erased. The cyberneticists of the post-war period elaborated a set of concepts—feedback, stability, control, communication, formal theories of information, and others—for understanding the behavior of such systems. These concepts were informed by, and came to influence, a variety of fields such as biology, psychology, cognitive science, anthropology, computer science, and business management. A full accounting of the projects of cybernetics is well beyond the scope of this chapter. What concerns me here is cybernetics' relationship to memory and performativity, which Pickering (2010) and Bowker (2005) describe in quite similar terms.

The systems that cyberneticists conceived of were promiscuously heterogeneous yet fundamentally *closed*. Everything relevant to the behavior of the system was to be found within the system itself. Ross Ashby's homeostat—a system that self-stabilized, through a process of negative feedback—is a paradigmatic example of such a closed system (Pickering 2010, 101). Ashby, who had a background in psychiatry, regarded the homeostat as a model of the human brain because of its ability to adapt to changing conditions. Importantly, Ashby contended that such a system needs no understanding of its own past to achieve equilibrium: stability is achieved through an iterative process of responding to the conditions of the moment. The temporal pattern of the homeostatic feedback loop involves a series of 'steps' in which the present continuously emerges from itself. (As we will see simulations involve this same temporal pattern). Information from past events may be needed, however it is not needed *as* a representation of the past. The past is only relevant as an aspect of the 'state' of the present. As Bowker notes, memory, according to Ashby, "is a metaphor needed by a "'handicapped" observer who cannot see a complete system, and "the appeal to memory is a substitute for his inability to observe" (Bowker 2005, 100, quoting Ashby (1956)). Cybernetics thus removes *conscious* memory as a necessary term for understanding complex behavior (in machines or humans).

An important aspect of Bowker's argument about the memory practices of cybernetics is

that the 'internal' temporarily of the objects of study (the homeostatic mechanism) and the 'external' temporarily of the discipline itself (the process of studying homeostatic mechanism) are like the faces of a mobius strip. The 'twist' from the temporality of the object to the temporality of the study of the object is apparent in Ashby's suggestion that conscious memory is just as unnecessary to the cyberneticist as it is to the cybernetic mechanism. As Ashby wrote":

> "Ordinarily, when an experimenter examines a machine he makes full use of knowledge 'borrowed' from past experience. If he sees two cogs enmeshed he knows that their two rotations will not be independent, even though he does not see them actually rotate. This knowledge comes from previous experiences in which the mutual relations of similar pairs have been tested and observed directly. Such borrowed knowledge is, of course, extremely useful, and every skilled experimenter brings a great store of it to every experiment. Nevertheless it must be excluded from any fundamental method, if only because it is not wholly reliable: the unexpected sometimes happens; and the only way to be certain of the relation between parts in a new machine is to test the relation directly." (Ashby (1956), quote from Bowker (2005), 101)

In cybernetics, memory is eliminated on multiple levels. This multiplied destruction of memory leads Bowker to describe the memory regime of cybernetics as the regime of the "empty archive." What I would like to emphasize here is the important role of *the unexpected* and the impossibility of reliably understanding complex systems by descriptive means, as elements of Ashby's argument against memory. Because the homeostat doesn't construct representations of its environment or past, it embodies a nonrepresentational approach to knowledge. Along these lines, Pickering regards cybernetic experiments as "ontological theater"; they are rooted in a performative epistemology that suggests an approach to science without a "detour" through representation (i.e., direct involvement with material agency) (Pickering 2010). (On the other hand, it is not quite right to characterize cybernetics as an anti-representational field of practice. After all, Ashby valued the homeostat as a *model* of

the brain).

The temporality of the cybernetic mechanism is closely related to the temporality of simulation. Simulations, such as those created by plasma scientists, start at a time of zero (t=0), and time progresses through a series of discreet steps. The 'state' of the simulation (i.e., the charge, position, and momentum of the simulated particles) are all that the simulation requires to calculate the next step. In this sense the simulation does not keep a record of its own past. (in practice, simulationists will often record the successive 'states' of simulation—or certain aspects of it—for later analysis, but the point here is that simulation only needs the present to calculate the immediate future. And it is only the *immediate future*, the next time step, that the simulation is concerned with). In this sense a simulation does not require a 'memory' of the past. Like the temporality of cybernetics, simulation is a case in which future states can only be determined by iterating through a sequence of moments.

Despite the similarities between cybernetic mechanisms and simulations, when we look to simulation practices for something like Bowker's 'twist' of the mobius strip—the turn that connects the temporal patterning of its object of study to the temporal patterning of the investigation—we do not find the same unequivocal renunciation of memory. Certainly, simulation work involves extensive tacit know-how (as discussed more in chapter three), however the 'archive' of simulation is far from empty. Plasma simulationists, for example, hold computational models like particle-in-cell (PIC) in high regard precisely because they are "very well understood." What they mean is that PIC codes have an extensive record of effective use, and that many of the obstacles and pitfalls associated with the PIC model (i.e., sources of unwanted numerical effects) have been thoroughly described in literature. Much of this 'description' and analysis of simulation codes occurs through the mathematical analysis of the PIC model as a 'space' of physical phenomena. So while the effective use of a PIC code requires a deep familiarity with the code (in an instrumental mode), the broader practice of resolving 'the unexpected' in simulation codes is critically informed by a rich scientific memory and a variety of representational tactics. Plasma simulation is not simply "ontological theater", to use Pickering's phrase.

How are we to understand this apparent contradiction or tension between the performa-

tive (memory-less) approach to knowing code and the analytic approach, rooted in representational tactics (like mathematical analysis of a simulation code)? In order to understand this aspect of simulation development, it's helpful to look to another domain of practice, software engineering. As we will see in the following section, this contradiction—between system and description—is not unique to computational science. In fact, it is a constitutive, durable tension that strikes at the conceptual foundations of software engineering. In the following section, I discuss the origins of software engineering in order to understand how the field has managed these tensions (between system and description). Software engineering, as a field, represents an important body of practice for working with code. Its methods and strategies have also played an important role in the formalization of research software development. For example, initiatives such as Software Carpentry and the Software Sustainability Institute, which seek to educate researchers in good coding practices, root their understanding of 'high quality' code in relation to a set of values articulated from software engineering (Goble 2014; G. Wilson et al. 2014).

2.5 Software Engineering's Formalization of Software

The field of software engineering, I argue, emerged in response to the perceived insufficiency of performativity as a strategy managing large, complex systems in the late 1960s. (Interestingly, this recognition occurred around the same time that Richards (1969), writing in the context of scientific computing, suggested that scientific programming should constitute itself as a 'literature'). Software engineering's inaugural moment came in 1968 with the first of two NATO Conferences on Software Engineering, held in Garmisch, Germany (Mahoney 2004). (There was a second conference the following year, in 1969). At the first NATO conference, software experts gathered to define their inchoate field as a discipline of engineering. As the conference proceedings explained, "the phrase 'software engineering' was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering" (Naur and Randell 1969).

The NATO Conferences were motivated by a sense of "crisis" pervading the software development community in the late 1960s. This sense of crisis involved several aspects. In particular, developers felt that they lacked the techniques and practices to build the large, complex systems they were increasingly asked to build as demand for software grew. Project were frequently over-budget and not completed on time. As R. Graham of Bell Labs remarked: "Today we tend to go on for years, with tremendous investments to find that the system, which was not well understood to start with, does not work as anticipated. We build systems like the Wright brothers built airplanes—build the whole thing, push it off the cliff, let it crash, and start over again." (Naur and Randell 1969, 54). The crisis also entailed a confrontation with code's "unavoidable fallibility" and the sense that software was a new source of social risk that compounded as the use and complexity of software systems multiplied. What should be clear from these statements is that the programmers at the NATO conference and the cyberneticists described above confronted similar problems with respect to the complexity of the systems they worked with—"the unexpected sometimes happens." However, for the software engineers (at least those at the NATO Conferences) the unexpected was a source of fallibility and risk that needed to be reduced and if possible eliminated. As we will see, the approach they took to resolve software's unavoidable fallibility—while never free of performative aspects—was significantly oriented toward representational tactics. In particular, software engineering constituted software development as a practice for sustaining memory of a system's past through the formalization of *privileged* representations that structured the development process.

The "software crisis" of the 1960s was precipitated by the risks associated with the growing use and complexity of computer applications on the one hand, and the lack of formal strategies for understanding and managing complex systems on the other. The solutions that programmers sought were methodological in nature. The strategies for developing software at an 'industrial' scale that emerged from the NATO conferences, which were further elaborated in the 1970s and onwards, conceptualized the "software lifecycle" as a series of stages, such as specification, design, coding, and testing (Mahoney 2004). In some instances the process was broken into as many as thirteen steps. As Frabetti (2015) emphasizes, these

stages involve different forms of writing that are differentiated by their relationship within a temporal structure. "[B]y separating problem from solution, while subsequently relating them through a series of written texts, the Garmisch conference report invests 'writing' with a central role in the organization of the time of software development" (Frabetti 2015, 82). A specification, for example, entails an enumeration of the requirements that a system should fulfill—a collection of user "needs." These requirement are then used to produce a design, another kind of document describing the individual components or modules that the system should consist of. The design is then implemented as a text written in a high-level programming language. The program is finally tested, which produces more documents in the form of bug reports. After testing, problems in the original specifications are often identified and the process repeats. Software engineering, in this mode, is a formalized system of inscription and re-inscription, involving a specific temporal structure. In its most basic form this process involves a repeated process of *description* and *implementation*. The system and its description evolve in lockstep, and the differences between them prompt further steps in the systems 'progress.' The important point is that software engineers enroll the linearity of writing to imagine the development process as a *predictable* path between the identification of a 'need' and its fulfillment as a system. This structure of inscription provides the basis for defining software's instrumentality. Further because this inscriptional arrangement constitutes a temporal structure—one used to organize time and make the development process predictable—software engineering centers memory (i.e. the specification as an evolving representation of 'needs') as a central concern for achieving this instrumentality.

Paradoxically, as Frabetti shows, the programming experts who attended the NATO conferences also acknowledged the *impossibility* of rationalizing software development as a linear and predictable process. As one conference participant put it: "program construction is not always a simple progression in which each act of assembly represents a distinct forward step and that the final product can be described simply as the sum of many sub-assemblies" (Naur and Randell 1969). The central concern here is that a software system cannot be completely specified in advance—the 'software lifecycle' is, at least potentially, an infinite loop, without necessary progress or regress. Even more confounding, programmers recog-

nized that the distinction between system and description, between conceptualization and realization is arbitrary or even illusory. As Naur notes in the conference report:

> "The distinction between design and production is essentially a practical one, imposed by the need for a division of the labor. In fact, there is no essential difference between design and production, since even the production will include decisions which will influence the performance of the software system, and thus properly belong in the design phase. For the distinction to be useful, the design work is charged with the specific responsibility that it is pursued to a level of detail where the decisions remaining to be made during production are known to be insignificant to the performance of the system." (Naur and Randell 1969, 31)

The subtle point here is that the presumed hierarchy between the distinct forms of writing associated with conception and realization—a representational hierarchy used to organize the time of software development and structure the work force—does not hold up in practice. In representational terms, Naur is noting the nonequivalence of design and implementation *and also* the non-priority of the former over the latter for determining the "the performance of the software system." If the design document is undermined as the privileged referent of the implementation, development is no longer a linear progression. Rather it becomes a process of "shuttling back and forth between different spaces of representation," each of which impose its own set of material constraints and affordances (Rheinberger 1997, 108). (Naur is also suggesting that, ideally, the distinction between design and implementation would dissolve in the same way that high-level programming languages unify the notational practices of mathematics and the language of the machine's operation). Examples of such "shuttling back and forth" are readily available in software development practices. The notion of *reverse engineering*—"the process of analyzing a subject system to identify the system's components and their interrelationships and to create representations of the system in another form or at a higher level of abstraction" (Chikofsky and Cross 1990)—aptly characterizes the inversion of the relationship between the forms of writing that structure 'normal' development.

It follows that software engineering's (impossible) formalization of programming is both necessary and inadequate for sustaining software's instrumentality. Following a deconstructive approach, Frabetti refers to these tensions in the foundational concepts of software engineering as *points of opacity*. The preceding discussion illustrates two such points: first, the impossibility of arranging software development as a linear, risk-free process from specification to implementation (because problems in the original specification appear only after the system has been implemented) and, second, the non-equivalence of the design as a privileged referent for the implementation. An additional point of opacity concerns the double role of the unexpected. Developers' ignorance of the systems they create is recognized as both a *threat* to be minimized and an essential resource that *propels* their work. "The original ignorance of what the system does is constitutive of the system" (Frabetti 2015, 90).

These contradictions are important for clarifying software engineering's status as a representational practice because they demonstrate disjunctions between terms (or traces) that the field treats as interchangeable. "A deconstructive reading of software is the opposite of a functional reading. For a computer professional, the point where the system 'undoes itself' is a malfunction, something that needs to be fixed. From the perspective of deconstruction, in turn, it is a point of revelation, on in which the conceptual system underling the software is clarified" (Frabetti 2015, xxi).

2.5.1 'Opacity' of Simulation

There is an important resonance between Frabetti's deconstructionist opacity and the notion of the "epistemic opacity" from philosophy of science (Humphreys 2009). In Humphreys' definition, "[a] process is essentially epistemically opaque to X if and only if it is impossible, given the nature of X, for X to know all of the epistemically relevant elements of the process" (618). Applied to computation, epistemic opacity refers to an inherent constraint in a person's ability to understand (in a representational sense) the complete computational process. It is seen as a constraint derived from the 'inhuman' scales of time and space within which computational processes unfold. Simulations are opaque in this sense because "no human

can examine and justify every element of the computational processes that [they] produce" (Humphreys 2009, 618). Humphreys argues that the epistemic opacity of computer simulation characterizes its epistemological and methodological distinctiveness; it is the source of a new set of issues for the philosophy of science. As in the case of the functionalist reading of software, epistemic opacity is presented as an inherent and unavoidable *problem*, a source of difficulty, for both the simulationist and the philosopher of science. For the simulationist it is a practical problem, for the philosopher, an epistemological one: How can reliable knowledge emerge from a fundamentally unknowable process?

The philosophical conception of epistemic opacity echo's Ashby's concern for the impossibility of descriptive understanding of a cybernetic systems. Such knowledge, he argued is unreliable because "the unexpected sometimes happens." Ashby's 'solution' to this difficulty, as characterized by both Bowker and Pickering, was found in a move to a performative idiom: analytical or descriptive knowledge of a system is not a reliable basis for predicting its future behavior; one must abandon the expectation of knowing how a system will behave in advance; the only reliable understanding is the "real-time" understanding of direct interaction. In relation to the representational practices in software engineering, the epistemic opacity of computing is analogous to the statement that there is no single representational space within which the entirety of a computational system can be understood by the engineer. The multiplication of inscriptional techniques associated with software development might be understood as an effort to manage this opacity by providing a range of representational tactics with distinct affordances. (The basic opacity remains as none of these representations is privileged).

Bringing the discussions of cybernetics and software engineering together, we see two historically specific encounters with complex systems. It would appear that they involve quite distinct strategies or responses to 'the unexpected.' In the case of cybernetics—at least in the example of Ashby—we see a rejection of conscious memory and representation as strategies for understanding cybernetic mechanisms. In the case of software engineering, we see an increased commitment to representation in the multiplied forms of writing for managing the development process. However, I have also noted contradictions associated these 'moves.'

While software engineering aims to structure programming as a linear process of writing, Frabetti's 'points of opacity' demonstrate that this goal (ultimately the instrumentality of software) is an impossible achievement. As we will see in the next section, this notion of opacity plays an important role in characterizing the temporal arrangement of simulation development.

2.6 Making the Unexpected Productive

From the preceding discussions, we saw that the memory practices of cybernetics and software engineering are strongly oriented toward the difficulties of complex systems: "the unexpected sometimes happens," as Ashby put it. For the remainder of this chapter, I turn to the case of plasma simulation and how members of the lab approached the unexpected as a source of problems and opportunities. By approaching plasma simulation as a memory practice, I aim to show how simulationists organize their work as an ongoing, cumulative process—how they constitute a past that is useful in the present, and productive of a future. 'The unexpected' continues to play a central role, as it did in the cases of cybernetics and software engineering. Its role is further complicated, however, by the fact that simulations are, in a sense—a sense that needs to be elaborated—*meant* to produce 'the unexpected.' Part of the scientific value of simulations, like experimental systems, is that they are generative of novel events (Rheinberger 1997). However, this novelty (or unexpectedness) is often not unanticipated. The questions to unravel then are: how is the structure of expectation (i.e., a distinction and relationship between the expected and the unexpected) produced? What is the role of scientific 'memory' in this production? And how does the structure of expectation relate to the kinds of novelty and interaction with material agency that characterize experimental work? The productivity of the unexpected, I argue, is a local achievement—it is not a teleological productivity of 'science itself.'

2.6.1 'We've seen this for years'

One of the most common ways simulationists encounter the unexpected is in the difficulty of getting codes to work as expected. In simulation "a great part of the effort is in getting the simulation to behave properly at all" (Kennefick 2000). The difficulty of getting codes to behave is not just that the codes and the parameters used to manipulate them are difficult to understand and often poorly documented. The 'results' of the simulation—principally graphical plots (sometimes animated) showing the evolution of the simulated plasma—are often ambiguous. Many of the lab meetings I observed were dominated by efforts to interpret visualizations of simulated data. Typically, this involved a group discussion of a particular plot, or set of plots. Discussions focused on the question of "what's happening" in the plot (i.e., what's being simulated? what's causing this feature?) and they often moved toward recommendations for improving the simulation to make a particular features more prominent and compelling. Visualizations were explicitly talked about as elements of an argument with more or less persuasive power.

In the following lab meeting excerpt, Sarah has presented her latest effort to simulate a certain plasma dynamic, the "hosing" instability, but she is having difficulty producing the expected behavior, a certain kind of oscillation. Sarah stands at one end of a conference table in front a group of plots (some animated) projected on the screen behind her. Sarah and Liu have been working together—Sarah more as a 'user' and Liu more as a 'developer'—and they respond to questions from Ben about whether or not "hosing" is occurring in the simulations.

> Sarah: Going back to the idea that I would like to see something like "hosing" [...] I increased the density to more intense fields, but probably not enough. And what I get when I started with a tilt, which is not visible in the beginning [...] I still get just a bunch [of particles] sucking to the wall. And I tried doing just a normal bunch without tilt, a little bit off axis, and I get this [pointing to image] but then I get it going, jiggling around. That one I'm still looking at, trying to understand [...]. So I still didn't get any evidence of hosing, but I did

get, like, a detail that is a problem for sure.

[. . .]

Ben: Why don't you say there's hosing, because it looks like there's hosing toward the end?

Sarah: This is the centroid . . . I had this in the movie but it's [not here].

Liu: That's the particle hitting the wall.

Ben: It hit the wall, it stuck to the wall, and then toward the end it starts to oscillate [like hosing].

Sarah: I think what happens, when you see the movie—there's a line, and then it spikes, and as it spikes, there's something that goes behind, and then it spikes more. And they have this tail that goes . . . it doesn't oscillate like something we would expect from something called "hosing."

Ben: Except for the last few time steps.

Liu: It's not oscillating. [. . .] It's not a beam anymore.

Sarah says she has not found "hosing" but something more ambiguous: "a detail that is a problem for sure." It is not clear what this 'detail' consists of—what its cause is, whether it is a physical or non-physical effect. In fact, Ben is not convinced that it is *not* hosing. Liu, the developer, participates in clarifying what's happening in the plots. Once Ben is convinced that there is no "hosing", the conversation moves to the issue of how Sarah should proceed in her work—how she might she go about dealing with the "detail." In this part of the discussion, Bruce (the lab PI) is the main interlocutor.

Bruce: I've got to think about this, because we've seen this for years. [. . .]

Sarah: I think what happens is when it hits the wall, it gets some weird behavior on itself, the bunch, that prevents it from jiggling around.

Bruce: Oh, that—we wrote a paper in [a scientific journal] on that. [...]

Bruce: Look at the paper by [former student]. [...] Another thing to try by the way, is to take [Lars'] diagnostic for this data, and decompose the fields into m equals zero, m equals one, and m equals two.

What I want to emphasize in this exchange are the references to (Bruce's) memory, published texts, and the 'diagnostic'—the resources that Bruce draws on and suggests to Sarah to make sense of the ambiguous results. Bruce's personal memory (in addition to other senior members of the lab, with decades of experience in plasma simulation) was an important source of continuity for relating current research to the past. Bruce's comment, "I've go to think about this," was a common refrain in lab meetings; as suggested here, it signaled an effort to reflect on past research to inform current grad students' efforts—and perhaps a desire to move the conversation along after the discussion has come to an impasse. This is not to say that the continuity of the lab's work is solely an achievement of Bruce's prodigious memory: it is important to acknowledge the occurrence of the ambiguous "detail", in relation to which the things "we've seen [...] for years" are retrospectively 'thought about.'

This exchange illustrates the lively use of plasma simulation's past—in fact the perpetual reconstitution of that past as it is related to the present—in the process of making determinations about what is expected and unexpected. The collective scrutiny of images unfolds as an effort to relate the ambiguous aspects of the plot ("some weird behavior on itself ... that prevents it from jiggling") to particular published accounts and personal experiences ("we wrote a paper on that"). In the process, Sarah's results and the broader discourse of plasma simulation are related to each other and organized as an ongoing activity—both are actively constituted in the process. Rheinberger, following Bachelard, refers to this historical dynamic as recurrence (Rheinberger 2005). Recurrence refers to the integration of the past

and the present, particularly in the sense that the past is a condition for scientific objects produced in present, and the present is applied retroactively to rethink the past. Scientists are spontaneous historians, as Bruce demonstrates heres.

The 'diagnostic' tool that Bruce suggests Sarah make use of introduces another set of considerations. One of the most important memory practices associated with plasma simulation concerns the use of mathematics to define how a code should and should not behave. As touched on already, plasma simulationists hold the PIC model in high esteem because it is "very well understood". The understanding, here, refers to awareness of the model's shortcomings and common problems to look out for (e.g., not enough particles, coarse grid size and time-step).

Plasma simulation literature consists significantly of efforts to identify and analyze difficulties associated with various numerical methods. A common strategy is to derive a description of the 'numerical' plasma from an analysis of the system of difference equations used in the code. Equipped with such a description, the simulationist is better able to resolve the conditions under which the 'numerical' plasma deviates from a 'real' plasma (i.e., as described by differential equations). A good example of this tactic involves the dispersion relation—an equation that describes the propagation of waves in a given medium. By deriving the *numerical* dispersion relation for a particular computational model (e.g., a one dimensional electrostatic code), the simulationist acquires a mathematical description of how waves propagate in the code, which can be compared to "real plasma" (Lindman, 1970). Essentially, the simulationists treats the code (or the difference equations the code implements) as a specific kind of medium that can be analyzed as any other medium within which waves propagate (i.e., mathematically).

The derivation of numerical descriptions of simulated plasmas is a central part of what plasma simulationists do. One graduate student described this peculiar kind of work as being "like physics, except that it's unreal." The sense of 'unreality' here is conceptually related to the confusion between implementation and description in the case of software engineering. In the same way that 'reverse engineering' seems to invert the privileged representational status of 'higher' abstractions (description) over 'lower' abstractions (implementation), the

analytical description of an "unreal" (numerical) model seems to upend the (naive) representationalist understanding of simulation practice in which the numerical model is produced as a representation of a set of differential equations. By associating the numerical difference equations of the code with the unreal and the differential equations of physical theory with the real, simulationists themselves assert this structure. However, in the derivation of a *numerical* dispersion relation—the physicists invert this relation as the computational model itself falls under description.

What I mean to illustrate here is that in Bruce's suggestion to use a diagnostic tool there is an embrace of the sensibility of "shuttling back and forth between different spaces of representation" that we also identified with the effort to understand the complexity of software.

2.6.2 Disputing the Unexpected

In the lab meeting excerpt just discussed, Ben expressed some initial confusion toward Sarah's claim that she "didn't get any evidence of hosing." On occasions, this sort of confusion—confusion about whether or not the code is behaving in a certain (bad) way—can rise to the level of a sustained dispute between lab members. In the following example, the code itself is used to demonstrate its own defect. The experience of the unexpected is not always shared. (This example is also discussed in the next chapter).

> Paul thinks he has found a problem in Fordham's code. The simulation he is running uses so-called thermal bath boundary conditions—the idea is that the space outside the simulation 'box' should be modeled as an extended plasma with a constant temperature. As particles move around in the simulation box, some will inevitably approach the edge and escape. Each time this happens, a new particle should be *added* to replace the departing particle. However, Paul notices that, often, the new particles immediately leave the simulation box—they are moving in the wrong direction. The problem, Paul thinks, is that the velocities are selected from a "full-Maxwellian" distribution instead of a "half-Maxwellian";

this causes a non-physical plasma temperature at the edge of the simulation box. Paul brings the issue to Fordham's attention, who wrote the code, but Fordham doesn't think there's anything wrong. The simulation data that Paul shows Fordham doesn't convince him that anything is amiss. Fordham believes the full-Maxwellian is the right approach, but to satisfy Paul he adds an option that will allow him to run the code with in a 'half-Maxwellian mode.' This is how things stand for a while—there is an option to use one approach or the other. Frustrated that Fordham doesn't recognize the problem, Paul works to identify simulation scenarios where the problem with the full-Maxwellian approach is fully apparent and undeniable. Eventually he succeeds. Fordham adopts the 'half-Maxwellian' approach and removes the 'full-Maxwellian' option.

This is an example of a problem in the implementation of a computational model that leads to unwanted, non-physical effects. What is significant is the extent of the effort to convince Fordham that there was a problem in the first place. The (mistaken) full-Maxwellian approach seemed to Fordham to be the common sense way to implement a thermal boundary condition. Further, the non-physical effects that it gave rise to were only noticeable in very specific situations, which is why it took Paul some time to identify a scenario in which the negative effects were undeniably apparent. Perhaps the most interesting aspect of this story, though, is Fordham's provisional decision to get around the substantive issue at the heart of the dispute (which approach is right?) by adding an option to the code's configuration. The introduction of the option illustrates how codes can evolve in ways that are *not* rooted in theoretical principles.

2.6.3 The 'Historiality' of the Unexpected

Physicists often use simulations to investigate and generate new theoretical insights of physical phenomena. Consider the following account from an interview with a member of the lab recognized as the lead developer for the one the lab's more widely used codes.

Fordham: [...] we did some work [with colleagues] on the Kelvin-Helmholtz

> instability. Basically, they were doing the simulations and they started seeing some phenomena that didn't seem to be predicted by normal theory, and they realized how the thing was actually being generated. They developed a theory for it, and basically they went back to the code and re-did a number of simulations to test out their theory with a number of different parameters. The two things matched, so we felt pretty sure that everything was okay and we could go and publish the thing. That's actually a very central part of what we do. [...]. There's a close interplay between guys developing theory and the guys running the codes to test this type of scenario.

In this account (and it is important to note that it is a *retrospective* account), one group of physicists—"the guys developing theory"—worked with another group—"the guys running the codes" (also the developers)—to study potential effects of the Kelvin-Helmholtz instability (KHI) in a specific physical scenario. (KHI is a frequently observed vortex dynamic in fluids. Jupiter's Red Spot is a well-known example). Fordham says that the code exhibited behavior that "didn't seem to be predicted by normal theory." This is an account of the emergence of something productive and novel, something scientifically meaningful perhaps, to consider more closely.

As a retrospective telling of an apparently successful use of his code, Fordham's account might be said to suffer from a problem that historians of science often encounter: "the recent is made into the result of something that did not so happen so. And the past is made into a trace of something that had not (yet) occurred." (Rheinberger 1997, 178). In considering this scenario, my intention is not to disclose a gap between the Fordham's account and the facts of 'what really happened.' Rather, by thinking of simulation development as a memory practice, the goals is to consider how the past is 'made' in this instance—and to understand the simulation developers' role in the production of that past.

Rheinberger's argument about the "time structure" and "historiality" of experimental systems, such as those implicated in the history of protein biosynthesis, is an important point of reference for thinking about the temporality of simulation development. Following

Ilya Prigogine, Rheinberger argues that time is not an *attribute* of experimental systems—experiments do not exist in universal time—rather, time is local and operational within experimental systems.

> "Research systems [. . .] are characterized by a kind of differential reproduction by which the generation of previously unknown things through unprecedented events becomes the reproductive driving force of the whole machinery. As long as this movement goes on, we may say that the system remains "young." Being young, then, is not a result of being located near zero on the time scale; it is a function—if you will—of the very functioning of the system. The age of the system is measured by its capacity to produce differences that count as unprecedented events and keep the machinery going" (Rheinberger 1997, 180).

A field of experimental science, according to Rheinberger, is an "ecological patchwork" of systems with different reproductive cycles that are (ideally) coupled so that epistemic objects and tacit knowledge can move between them. I do not mean to overlook the important distinctions between experimentation and simulation. The benefit of this conception is that it helps to describe how simulation is coupled to other activities (e.g., "the guys developing theory" and "the guys running the code"). While the code in Fordham's account is 'old' in a traditional sense—it's been used for decades—it continues to be 'young' in the sense of producing new insights that are relevant for theoreticians.

One expression of this 'coupling' is the ambiguity over who is "doing the simulations." Fordham fluctuates between "we" and "they" in the account. Are the theorists *using* the developers' code or are the developers doing the simulations *for* the theorists? I understand this ambiguity as an indication that the categories of *user* and *developer*—considered here as roles from software engineering, crucial to conceptualizing software as a 'product', and which often are projected onto interactions with software—are not wholly appropriate for contexts such as this one. Software engineering, you might say, endeavors to make software 'old' (in Rheinberger's sense) as quickly as possible; research software development does not. As I describe in more detail in the next chapter, in the case of plasma simulation, developers

recognize themselves to be the "best users" of their own codes. More significantly though, the developers are closely involved in the process of deciding that "everything was okay and we could go and publish the thing." In other words, the developers' expertise of the code was crucial for identifying the code's behavior as "a [new] phenomena" rather than a spurious, non-physical behavior or bug. These two observations are related. The ambiguity between use and development and the role of the developers in effectively 'using' the code demonstrate the non-instrumentality of the simulation code in this instance. The code has produced something new, but the code itself is not sufficient for interpreting that event as progress or regress, nor is simply any user. It takes the developers' embodied expertise of the code to adequately interpret the codes' behavior. The encounter with the unexpected is made productive, but this productivity is a local accomplishment.

2.7 Conclusion

Let me return to the lingering possibility of research software as a 'literature,' in the sense that Richards invoked that term. How do the aspects of plasma simulation just described inform the question of research softwares' formalization as a category of scientific production? Can research software development be organized as an on-going, cumulative process or are researchers doomed to the experience of the Wright Brothers: to "build the whole thing, push it off the cliff, let it crash, and start over again" (Naur and Randell 1969, 54)? Frabetti's deconstructive reading of software engineering and Rheinberger's portrayal of the temporal coherence of experimental systems (which, I have argued, also informs the coherence of simulation work) present a common response to this question. The answer is something like: yes, but only to the extent that such an "overarching chronotopic paradigm" (Rheinberger 1997, 181)—for example, software engineering's linearization of programming—is able to perpetuate itself under erasure, as a durable and necessarily impossible fantasy. As it happens, these fantasies do endure—the vision of "cyberinfrastructure" (Atkins 2003) presented in the next chapter falls under this category.

Perhaps the more important question is not whether research software engineering can be

constituted this way but whether it ought to be. Bowkers' ultimate argument with respect to memory practices is a warning—the move toward commensurability and standardization, which characterizes much of contemporary computationally-driven science, forecloses as many possibilities as it opens up. It's important to retain the possibility of an unruly past. As we've seen in this chapter (in fact, it is a continuing theme of the dissertation), the coherence of the lab's software development practice is thoroughly local. If software can be a 'literature' then perhaps our conceptualization of software development should adopt an expansive sense of that term.

2.8 Bibliography

Ashby, William Ross. 1956. *An Introduction to Cybernetics.* Chapman & Hall and University Paperbacks.

Atkins, Daniel. 2003. "Revolutionizing Science and Engineering Through Cyberinfrastructure: Report of the National Science Foundation Blue-Ribbon Advisory Panel on Cyberinfrastructure," January.

Backus, J. 1998. "The History of Fortran I, II, and III." *IEEE Annals of the History of Computing* 20 (4): 68–78. doi:10.1109/85.728232.

Barad, Karen. 2007. *Meeting the Universe Halfway: Quantum Physics and the Entanglement of Matter and Meaning.* Duke University Press.

Berry, David M. 2011. *The Philosophy of Software: Code and Mediation in the Digital Age.* Basingstoke, Hampshire; New York: Palgrave Macmillan.

Bowker, Geoffrey C. 2005. *Memory Practices in the Sciences.* MIT Press.

Chikofsky, E. J., and J. H. Cross. 1990. "Reverse Engineering and Design Recovery: A Taxonomy." *IEEE Software* 7 (1): 13–17. doi:10.1109/52.43044.

Decyk, V.K., C.D. Norton, and H.J. Gardner. 2007. "Why Fortran?" *Computing in Science Engineering* 9 (4): 68–71. doi:10.1109/MCSE.2007.89.

Derrida, Jacques. 1978. *Edmund Husserl's Origin of Geometry: An Introduction.* U of Nebraska Press.

Frabetti, Federica. 2015. *Software Theory: A Cultural and Philosophical Study.* Rowman & Littlefield International.

Goble, Carole. 2014. "Better Software, Better Research." *IEEE Internet Computing* 18 (5): 4–8.

Humphreys, Paul. 2009. "The Philosophical Novelty of Computer Simulation Methods."

Synthese 169 (3): 615–26. doi:10.1007/s11229-008-9435-2.

Kennefick, Daniel. 2000. "Star Crushing: Theoretical Practice and the Theoreticians' Regress." *Social Studies of Science* 30 (1): 5–40. doi:10.1177/030631200030001001.

Knuth, D. E. 1984. "Literate Programming." *The Computer Journal* 27 (2): 97–111. doi:10.1093/comjnl/27.2.97.

Mackenzie, Adrian. 2006. *Cutting Code: Software and Sociality*. Peter Lang.

Mahoney, Michael S. 2004. "Finding a History for Software Engineering." *IEEE Annals of the History of Computing* 26 (1): 8–19.

Naur, Peter, and Brian Randell. 1969. "Software Engineering: Report on a Conference Sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany, 7th to 11th October 1968." Garmisch, Germany: Nato.

Pickering, Andrew. 1995. *The Mangle of Practice: Time, Agency, and Science*. University of Chicago Press.

———. 2010. *The Cybernetic Brain: Sketches of Another Future*. University of Chicago Press.

Rheinberger, Hans-Jörg. 1997. *Toward a History of Epistemic Things: Synthesizing Proteins in the Test Tube*. Stanford, Calif.: Stanford University Press.

———. 2005. "Gaston Bachelard and the Notion of 'Phenomenotechnique'." *Perspectives on Science* 13 (3): 313–28.

Roberts, K. V. 1969. "The Publication of Scientific Fortran Programs." *Computer Physics Communications* 1 (1): 1–9. doi:10.1016/0010-4655(69)90011-3.

———. 1971. "Computers and Physics." *International Center for Theoretical Physics, Computing as a Language of Physics*, 3–26.

Shankar, Kalpana. 2007. "Order from Chaos: The Poetics and Pragmatics of Scientific Recordkeeping." *Journal of the American Society for Information Science and Technology*

58 (10): 1457–66. doi:10.1002/asi.20625.

Wilson, Greg, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, et al. 2014. "Best Practices for Scientific Computing." *PLoS Biol* 12 (1): e1001745. doi:10.1371/journal.pbio.1001745.

CHAPTER 3

Opening Plasma Codes

3.1 Abstract

In this chapter I consider the lab's ambivalence to prevailing norms of research software development associated with cyberinfrastructure and open science policies. These policy initiatives emphasize that code should move beyond its community of developers (i.e., as shared 'infrastructure' or as reproducible research). While the lab adopted certain aspects of open source development, its most well-known codes remain closed. The chapter's main argument is that the vision of research software offered by funding agencies did not coincide with the lab's. I use the notion of a user-developer regress to explain the lab's reluctance to fully embrace open access policies. A user-developer regress arises when the value of software is related to the potential for code to move beyond its community of developers yet the expertise needed to use the code is stubbornly fixed to its community of developers. The user-developer regress manifests itself in disputes between users and developers over whether the code is 'working'.

3.2 Introduction

Stylish black signs guide me through unstylish beige corridors in the University's math and science complex. They deliver me to a room described as a "visualization portal" where an international group of plasma simulationists is gathering for a workshop on a code of collective concern, a code I'll call PULSE. The "portal" is a presentation space replete with the tropes of 'high tech' scholarship. With steel gray walls and acoustic tiles,

one imagines presenters standing dramatically in the projector light, washed by animated graphs and three dimensional models. At the back of the room are racks of A/V equipment with blinking LEDs and a bank of computer monitors attended by dedicated technical staff. It seems like a lot of technology for a few Powerpoints, I think. On the other hand, an overwhelming technical presence is appropriate for this gathering, even if the equipment is a somewhat meager substitute for the higher performance computer clusters PULSE often runs on. Before heading to my seat, I pay my respects to the machine with a nod to its attendant.

The workshop is the first formal meeting of the extended community of PULSE users and developers. Representatives from over a dozen institutions are in attendance. The workshop also marks the release of a major new version of the code. Over the past several years developers from several institutions have worked on 'modernizing' the code: they have rewritten it, making use of paradigms like object oriented programming; they have restructured the codebase to make the implementation of new extensions more straightforward; and they have taken advantage of quality assurance strategies like unit testing and continuous integration to limit the introduction of bugs. They have done all this because the community of PULSE users has steadily grown since the code was initially developed in the late 90s, and because the quality of the code—the ease with which it can be extended and maintained—was beginning to suffer as PULSE was extended to new problems. This meeting, then, is an important milestone. It represents not just the 'maturity' of PULSE as a piece of software but the emergence of a community organized around its use and improvement.

The head of one of the labs leading PULSE's development, Bruce, opens the workshop with a slide he says he often uses in talks about the code. The slide shows a few bullet points describing the code in terms that are both technical and scientific—"massively parallel", "fully relativistic". On one side are logos of the institutions and funding agencies involved in its development; there is a colorful, three dimensional model of a simulated plasma that looks like a struggle between an exploding rainbow and a fiery tangle of yarn. The slide also includes a plot of how PULSE performs on one the newest high performance computers available to researchers in the US. It shows the code's "speed up" (a factor of 1 to 1000)

in relation to the number of computer 'cores' used in the simulation (from 1000 to over a million). The line goes up nearly linearly, falling just below a dashed line called 'optimal'. The plot is meant to show that the code's parallelization scheme is efficient, that it can "scale up" to larger simulations without losing performance due to the overhead of parallelization itself.

In many respects, PULSE is a model case of scientific software development as envisioned in recent decades by national funding agencies. As part of its cyberinfrastructure initiative, the National Science Foundation (in particular) has aimed to "revolutionize" science and engineering through coordinated funding of projects that build and integrate shared resources used in computationally-driven scholarship (Atkins 2003). Software has played a critical role in this vision, and the agency has incentivized software "that works easily at scale, encourages reuse, and efficiently promotes innovation while retaining reliability" (National Science Foundation 2012). These are ambitions that PULSE's developers have pursued in their efforts to modernize and improve the code, to expand its user base, and to do so without sacrificing performance.

In other respects, however, PULSE's development deviates from the visions of scientific software development articulated by funding agencies, science policy advocates, and champions of "open science". Directly in the center of Bruce's slide, we see that PULSE is "open access through MoU". Bruce explains that the code is *not* distributed with an open source license. Not even an executable binary (without the accompanying source code) is available for public download. Instead, as Bruce explains, PULSE is only available to those who have signed a memorandum of understanding (MoU) and received access to the online code repository (on GitHub). MoUs are frequently used in large scale scientific collaborations, particularly in physics, to define mutual expectations between partnering institutions (Schroeder 2008). Often, they are used to define the *resources* (financial, computational, etc.) that collaborating institutions will contribute to a project, as well as expectations around publishing results derived from the use of those resources. For example, the Worldwide Large Hadron Collider Computing Grid, the computing infrastructure used to process data from the LHC, is constituted through such an agreement (Bird 2011). MoUs, then, are

quasi-legal instruments often used to manage the design, access, and use of resources used in large scale scientific research.

Understood as a condition of access to a complex scientific instrument and object of considerable investment on the part of multiple institutions, PULSE's MoU is perhaps not surprising. However, the MoU is serving as a gatekeeper to a resource that is not limited in the same way that traditional, physical scientific instruments are. Unlike particle accelerators and telescopes, software is conventionally regarded as a 'non-rival' resource—your use of a computer program does not interfere with my ability to use that same program. This non-rivalrous quality is an important aspect for arguments that software—like other forms of information—should be understand as a public good (Benkler 2006). PULE's "open access" policy suggests a gesture toward the norms of "open science"—an approach funding agencies have encouraged in order to achieve sustainability (Stewart, Almes, and Wheeler 2010) and reproducibility (Stodden 2010)—without fully committing to it.

A degree of ambivalence toward the vision scientific software development offered by funding agencies and policy advocates was also apparent in my conversations with members of the lab.

> Bruce: My friends [colleagues in experimental plasma physics] buy a laser and they get all this money from a funding agency and they build this huge facility, and they're not being told by the funding agency 'you now have to let anyone in the world come and use your facility... and they can bring their own equipment and I want you guys to figure out how they can add it in there and integrate it, because we funded it and it should be an *open* lab.' But software ... because the funding agencies have seen from the commercial side, from Linux and others, how this has grown, they've seen that this is the way software should be treated.

Bruce regrets what he sees as a double standard in the expectations associated with traditional scientific instruments and research codes like PULSE. What is notable is that he considers the very aspects of software that funding agencies value (resource sharing and ease of integration) as unimportant—in fact he suggests they are sometimes an imposition

on his work. Regardless of whether Bruce is correct in his characterization that funding agencies are eager to carry-over practices of openness that have succeed on "the commercial side", funding agencies do present a "way software should be treated". As Bruce's comment suggests, the simulationists do not accept this vision as inherently good—in certain respects they are even critical of it.

This chapter considers the simulationists' work amidst the background of changing normative conditions of scientific software development. Rather than accept the visions of software development offered by funding agencies and policy advocates—visions of 'open science', for example—as inherently good, I consider these prevailing norms as values that are selectively enacted and performed. This approach follows work in science and technology studies that regards issues such as 'openness' as dynamic, situated, and dependent on local goals, constraints and opportunities (Levin and Leonelli 2017; Kelty 2012).

Historically, computational physics has enjoyed a fair degree of autonomy when it comes to determining its technical practices. This autonomy is understandable given the particularities of the computational tools physicists make use of. The computational resources that simulationists require are not as ubiquitous as the consumer-oriented devices of personal and mobile computing; they are geographically concentrated in supercomputing centers, and access to them is carefully managed, often by national agencies like the NSF and the DOE. Plasma simulation codes are frequently tailored to *particular machines*, and developers strive to ensure that their codes make efficient use of the computers on the "top 10" list—a regularly updated list of the world's most powerful supercomputers as measured in floating operations per second (flops). General and broad compatibility is not a major concern. Further, empirical studies of software development have noted scientist's disregard for general purpose software engineering methods (Howison and Herbsleb 2011). Plasma simulation codes are often written in Fortran, a language that has been quite slow to incorporate 'modern' programming conventions like object orientation and which other programmers would regard as antiquated. Even the *actual* language of scientific computing is distinct—simulationists have the unique habit of treating "code" as a countable noun ('the lab has several codes'). This grammatical treatment appears to be quite specific to scientific computing.

These observations illustrate that plasma simulation—and high performance scientific computing (HPSC) more generally—is a rather autonomous community of practice, fairly well insulated from broader norms and standards of computing. If we take this cultural distinctiveness as a sign of disciplinary autonomy—i.e, the degree to which the field can define its own methods and objects of analysis—then plasma simulation and the wider field of HPSC appears to be a rather well delineated and "strong" academic territory (Becher and Trowler 2001). Its authority in this sense is secure. This image of HPSC, and computational physics in particular, as a "strong" community—strong *because* it is insular—comports with arguments from the history of science that the adoption of shared standards and norms is only necessary in fields with insecure borders (i.e., to shore-up objectivity) (Porter 1996). Secure communities like high energy physics are based largely on informal knowledge, on trust in personal relationships, and non-standardized measurement practice (Porter 1996, 245; Traweek 1992).

In recent decades, this autonomy has shown signs of breaking down as wider norms and practices associated with scientific computing have become more prominent fixtures in science policy. In the case of plasma simulation, codes have grown much larger, more complex, and the level of investment required to build and maintain them has grown. Funding for code development is increasingly accompanied by expectations that the resulting code will be broadly reusable. Physicists also turned to professional practices to manage increasingly complex software projects. Nevertheless, the broad vision of research software offered by funding agencies did not coincide with the lab's.

This chapter is organized as follows. In the next section I review the "visions" of scientific software development offered by funding agencies and policy advocates. This discussion focuses on cyberinfrastructure and open science. In the cyberinfrastructure vision, software exist in a continuum between narrowly focused, discipline-specific tools and general purpose, shared resources (National Science Foundation 2012). Importantly, the different parts of the continuum are meant to be "integrated" so that more particular tools take advantage of shared resources wherever possible. The cyberinfrastructure model incentivizes the production and use of reusable and sharable tools, resource, components, etc. The open science

vision of software seeks to reduce or eliminate barriers to the public disclosure of research codes. Open science advocates recognize research codes as features of methodology that must be *communicated*, and made available to the "organized skepticism" (Merton 1942) of science. The integrity of computational science thus depends on the 'openness' of its research codes. Crucially, both cyberinfrastructure and open science locate the scientific value of software in its capacity to move beyond an immediate community of developers. They both feature the expectation that scientific code will be taken up by an expanding base of users, either as shared infrastructure or as the content of scientific communication.

In the final part of the chapter, I turn to the lab's mixed attitude toward prevailing normative visions of scientific software development. Many of the practices I observed in my fieldwork accord with emerging norms: the lab developers moved toward models of open source development (for some but not all of its codes); they adopted practices associated with "sustainability" (modernizing code to make it more maintainable); and certain categories of code development that fit into the vision of cyberinfrastructure were emphasized. My argument is not that the lab's activities were purely an *effect* of science policy. In many ways, the changing policy landscape has benefited the lab in that software development is increasingly recognized as a fundable activity.

Nevertheless, the lab remained ambivalent toward these visions. The paradox of the instrumental—that instruments are conditions for scientific knowledge production and yet is also, themselves, somewhat unknowable—is a central concern for plasma simulationists. 'Opening' such communities is problematic in that openness demands a different relationship to instruments in which the tension of the instrumental is collapsed. I identify a paradoxical relationship—the *user-developer regress*—that describes a central difficulty in opening plasma simulation codes. A user-developer regress arises when the value of software is related to the potential for code to move beyond its community of developers yet the expertise needed to use the code is stubbornly fixed to its community of developers. The user-developer regress manifests itself in disputes between users and developers over whether the code is 'working.' When users claim that the simulation is 'wrong' (i.e., it fails to produce the expected results), developers can respond that they aren't sufficiently familiar with the code

(i.e., they aren't simulating the experiment correctly). Users, developers feel, simply don't have the authority to make claims about whether the code is right. The user-developer regress, I argue, is not just an obstacle to open science and cyberinfrastructure policy. It is a a product of prevailing norms associated with the use of code as a form of scientific knowledge. One consequence of the user-developer regress is that simulation developers sought approaches to 'openness' in which the community of users would not expand too rapidly.

3.3 Policy Visions of Scientific Software

In this section I present the vision of scientific software development as articulated in science policy literature, particularly in material associated with *cyberinfrastructure* and *open science* initiatives. My purpose is not to argue for or against these initiatives. It is to present *cyberinfrastructure* and *open science* as historically specific constellations of values and norms concerning the role of software in science. Cyberinfrastructure and open science policies say something about "the way software should be treated"—they present conceptions of what software is, what its scientific value consists of, and how practices associated with it should be shaped to maximize the benefits to science. Cyberinfrastructure and open science are extensively interrelated 'movements' but they are also sufficiently distinct. They relate to slightly different aspects of the traditions of scientific progress. My discussion emphasizes how each of these policy visions treats the various 'abilities' of software—accessibility, reusability, sustainability, reproducibility—as scientifically valuable.

3.3.1 Cyberinfrastructure

The term 'cyberinfrastructure' is a product of the late 1990s, and its contemporary usage is strongly linked to a 2003 report, "Revolutionizing Science and Engineering Through Cyberinfrastructure" (Atkins 2003). The so-called Atkins report envisions scientific research accelerated by computing and information technology. For these opportunities to be realized, the report argues, significant investment in a new kind of infrastructure is called for:

> "The term *infrastructure* has been used since the 1920s to refer collectively to the roads, power grids, telephone systems, bridges, rail lines, and similar public works that are required for an industrial economy to function. Although good infrastructure is often taken for granted and noticed only when it stops functioning, it is among the most complex and expensive thing that society creates. The newer term *cyberinfrastructure* refers to infrastructure based upon distributed computer, information and communication technology. If *infrastructure* is required for an *industrial* economy, then we could say that *cyberinfrastructure* is required for a *knowledge* economy." (Atkins 2003, 5, emphasis in original).

The Atkins report presents cyberinfrastructure as the basis of a new ecology of knowledge production in which the pace of scientific discovery is increased by leveraging computational capacities, such as scalability, the exponential growth rates of storage and processing, and the ability to use and share remote resources (computing facilities, digital libraries, communication networks, etc.). The report aims at a transformation in the "the conduct of science" (49) as a whole. The report conceives of science as a vast technically mediated process in which both efficiencies and discoveries are produced by expanding capacities and integrating computational tools used in different domains. (It is important to note that the term, cyberinfrastructure, is fairly specific to science policy in the US context. "E-science" designates a similar set of initiatives in the UK, for example).

Software plays a critical role in cyberinfrastructure policy. While the Atkins report is careful to distinguish between domain-specific software applications (e.g., simulation codes tailored to particular research problems) and general purpose tools (e.g., the parallelization systems and compilers used across simulation codes), the science-enhancing qualities of software are not exclusive to the latter. Indeed, the logic of scientific progress through software evolution presented in the report is one in which domain-specific tools are repurposed for new research areas: "There is an exciting opportunity to share insights, software, and knowledge, to reduce wasteful re-creation and repetition. Key applications and software that are used to analyze and simulate phenomena in one field can be utilized broadly. This will only take place if all share standards and underlying technical infrastructures" (12).

The scientific value of software, then, is that it can be easily shared, adapted, repurposed to new research areas. Importantly, these opportunities hinge on integration with lower-level systems through the use of shared standards. A key feature of the cyberinfrastructure vision of software is the "belief that software ought to evolve toward a shared platform, with components that are reused as widely as possible as both end-users and component producers coalesce around particular pieces of software" (Howison et al. 2015).

The National Science Foundation's conception of the role of software in cyberinfrastructure is further clarified in the report, "Vision and Strategy for Software for Science, Engineering, and Education" (National Science Foundation 2012). This report is part of the NSF's larger initiative, "Cyberinfrastructure Framework for 21st Century Science and Engineering". As the report states, "The NSF vision is to facilitate software infrastructure that works easily at scale, encourages reuse, and efficiently promotes innovation while retaining reliability." (National Science Foundation 2012, 4). The report presents a three-tiered model of the scientific software ecosystem:

- "Software Elements: targeting small groups that will create and deploy robust software elements for which there is a demonstrated need that will advance one or more significant areas of science and engineering.
- Software Frameworks: targeting larger, interdisciplinary teams organized around the development and application of common software infrastructure aimed at solving common research and industrial problems, resulting in sustainable community software frameworks serving diverse communities.
- Software Institutes: establishing long-term hubs of excellence in software infrastructure and technologies, research and application communities of substantial size and disciplinary breadth." (National Science Foundation 2012, 5–6).

These categories are meant to guide the NSF's funding of cyberinfrastructure projects. Each category is successively associated with a larger community, and the software elements are integrated with larger, more general components developed by software institutes. At

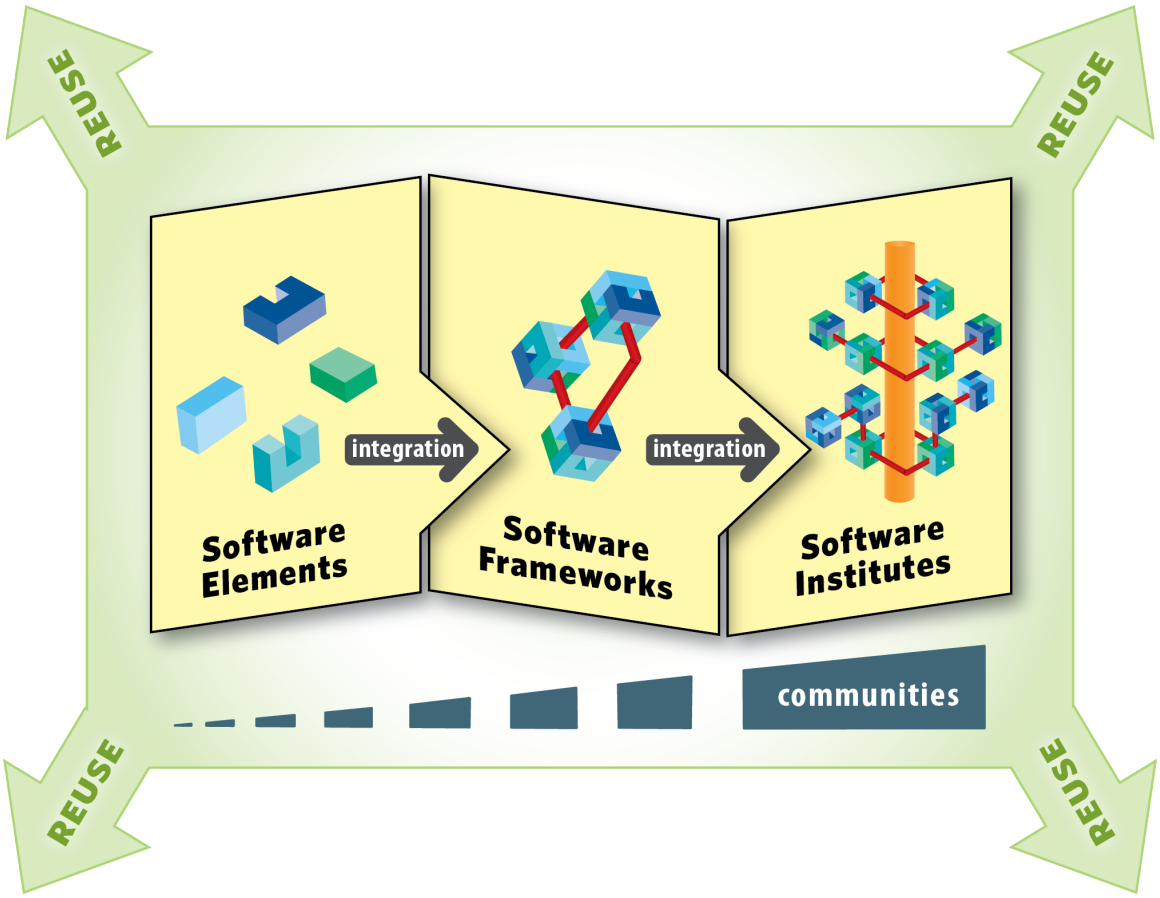

Figure 3.1: The NSF's (2012) three-tiered model of cyberinfrastructure software.

each level, reusability is encouraged. Further, open source approaches are explicitly embraced in the NSF's software vision. The activities that the NSF will invest in, according to the report, include "[w]idespread adoption of open source models for software development and dissemination that include high quality documentation and accepted engineering practices, leading to software that is accessible, understandable and reusable" (9).

3.3.2 Open Science

Whereas cyberinfrastructure refers to a fairly well delineated set of policy initiatives, *open science* can be linked to a broad array of scientific practices and norms. In certain respects it has more extensive historical roots. Concerns for "openness" in science are myriad, covering the public accessibility of scientific publications, the public disclosure of knowledge claims (that they may be openly debated), the cooperative nature of scientific inquiry, and the possibility for public participation. In the expansive sense, open science is the "conceptualization of science as the pursuit of 'public knowledge' and an object of public-minded patronage" (David, Den Besten, and Schroeder 2006). Open science is often presented as integral to the ethos of modern science; it is articulated in relation to scientific norms identified by Merton (1942). Merton argued that the integrity of science is the result of institutional norms. He identified four of them: *communalism* (common ownership of scientific products), *universalism* (claims are evaluated by common criteria, regardless of the individual who makes them), *disinterestedness* (science is for the public good, not personal benefit), and *organized skepticism* (claims are evaluated by the community). David (2006) recognizes Merton's norms as a primary expression of the relatively recent emergence of openness in science. For example, the imperatives of communalism and organized skepticism imply an expectation that findings and methods are fully disclosed and widely accessible. Research from science studies since the 1970s—particularly from the tradition of laboratory ethnography—presents ample critique of Merton's norms, arguing that they do not describe scientists' actual practices and motivations (Latour and Woolgar 1986). Yet "open science" persists as a powerful vision of how science *ought* to be shaped.

In recent years *open science* has become more narrowly associated with the policies and advocacy positions of open access, open source software, and open data—movements that seek to reform the conditions of reuse for scholarly publications, code, and data, particularly by leveraging internet technologies. In this context, *open science* refers to practices and guidelines that can be applied across the disciplines, and that support values such as transparency, accountability, integrity, accessibility, and reusability (Willinsky 2005; Levin and Leonelli 2017). Open science policies might be regarded as an effort formalize Merton's normative structure of science—a structure that he recognized explicitly as *informal*. (The formalization of the scientific reward system in open access policies is recognizable in the fact they do not remove all barriers limiting distribution of a publication: the expectation of attributing authorship remains. Thus ensuring that the priority of the original work is at least recognized).

A central animating concern of open science—one with particular relevance for scientific computing and code sharing policies—is *reproducibility* and the sense that contemporary science is experiencing a "reproducibility crisis" (Baker 2016). The authority of modern science is often seen to rest on the ability to replicate laboratory conditions and experimental results. Repetition is seen as the essential practice for evaluating claims and settling disputes. However, surveys of scientific literature from a range of fields often find low rates of reproducibility, leading to the concern that a significant portion of published research findings are false (Ioannidis 2005). Failures in reproducibility are attributed to a variety of causes—selection bias, pressures to publish, reluctance to share methodological details; solutions to the problem often include a call for greater transparency in reporting methods, and data sharing (Munafò et al. 2017).

STS research has contributed significantly to understanding challenges associated with reproducibility. Collins' (1985) work on gravitational wave physics, for example, is important for demonstrating the obstacles and limits of replicating experiments. Collins notes that in cutting-edge research areas much of the knowledge required to make experiments 'work' is tacit—tacit knowledge cannot be explicated and it can only be 'transferred' through social interaction. Scientists need to spend time in the 'life world' of the lab where the experiment

works. Tacit knowledge is a problem for reproducibility for two reasons—it means the expertise needed to repeat an experiment is stubbornly fixed to particular groups of people, and failures in reproducibility can always be blamed on an inability to precisely recreate the original experimental conditions. Collins refers to the latter problem as the "experimenter's regress"—a term that I will return to in the discussion below to explain the challenge of opening plasma simulation codes.

Advocates of open science argue that computational methods ease some of burdens associated with scientific reproducibility while introducing others. Stodden (2010) argues that computational science avoids some of the problems associated with "tacit knowledge" in traditional scientific research: "The rules of the game have changed with the increased use of computational tools in science, that afford, in theory, precise replication of published results by independent parties." Because software knowledge is necessarily explicated, some of the problems associated with tacit knowledge are not as prevalent, she argues. Stodden presents a vision of scientists communicating and evaluating hypothesis through code: "Imagine the ability to routinely inspect code and data and recreate others' results: Every step taken to achieve the findings can potentially be transparent. Now imagine anyone with an Internet connection and the capability of running the code being able to do this."

Scientific computing also introduced its own challenges with respect to open science. Scientists are often not inclined to make their codes available to colleagues. Further, they are not trained in 'best practices' that facilitate the development of high quality code—code that is well documented, tested, and easy to debug (G. Wilson et al. 2014). As as result, research codes are often difficult or impossible to understand for anyone other than the developer. Software may be 'open', but if it is of poor quality in these respects then it is not an effective form of scientific communication (Hey and Payne 2015). Projects associated with "software sustainability" aim to give scientists the tools they need to write more maintainable code (Goble 2014).

It should be clear from this discussion that there are many points of overlap between the open science and cyberinfrastructure discourses. They are not strictly distinct visions. Software sustainability is a common theme as both movements recognize the poor quality

of scientific code to be a problem for broader reuse. Open science advocates also argue that open science requires an "infrastructure" (particularly around publishing) that will make reproducibility more broadly realizable (Stodden and Miguez 2014). An important point of overlap for my argument is the expectation that code should move beyond the immediate community of developers. This vision concerns the way software moves into the wider scientific community—in fact I would argue that it concerns the *kind* of communities that should form around software to begin with. Both cyberinfrastructure and open science emphasize sharing, but they have distinct ways of conceptualizing what is shared (e.g sharing a road versus sharing a text).

Finally, both cyberinfrastructure and open science involve a dialectic between the *opportunities* associated with reusability and resource sharing and the *challenges* that prevent the realization of those opportunities. Much of the contemporary social research around scientific software development adopts this structure; it seeks to identify obstacles and ways to overcome them (e.g., incentive structures that promote sharing). The argument that I present below does not fint into this dialectic. My approach to cyberinfrastructure and open science is informed by perspectives from STS that grapple with the tradeoffs and forms of social order associated with these visions. For example, Kelty (2012) argues that "open" expert communities are also "closed" to the extent that they constitute a cohesive domain of practice around shared concepts, objects, and goals. This idea is of a piece with Shapin and Schaffer's insight that "solutions to the problem of knowledge are solutions to the problem of social order" (Shapin and Schaffer 1985, 332). The important distinction between expert communities is not whether they are open or closed but how they balance and configure openness and constitutive closure. As we will see, the lab stands at a cross-roads with respect to the social organization of computational science.

3.4 Simulationists' Ambivalence to Openness

The remainder of this chapter turns to the situation in the lab and how lab members reconciled the prevailing norms of scientific software development with their own research agendas.

A danger in this discussion is to present science policy as an 'external' force that lab members merely responded to. My argument does not hinge on a causal account of science policy's *effect* on the lab's 'internal' software development practices. I regard cyberinfrastructure and open science as reflections of prevailing norms associated with scientific computing. The lab's practices, I argue, need to be understood in relationship to these norms, not in opposition to them.

Consider, for example, the change in the lab's meeting schedule. During my fieldwork, the lab typically met twice per week—there was a meeting on Monday that focused on "physics" and a meeting on Friday that focused on "code development". Monday's meeting often included practice conference presentations and updates on ongoing research projects. Friday's meeting, on the other hand, were oriented toward technical training and questions of code design—topics included documentation, code testing, design discussions, and code review. As one interviewee explained, "[we used to have] a physics meetings only, once a week. The code development meetings arose because of [a grant], and the need to make open source codes available." While the the change in the lab's meeting schedule is clearly related to the conditions of the grant and "the need to make open source codes available", it also reflects an organic desire within the lab to improve the quality of its code. Senior physicists in particular, recognized the growing complexity of the simulations and the need to adopt more professional practices (consistent documentation, code testing, etc.) to streamline the development process. Regression testing—a technique for checking that *new* features in the code don't break existing functionality—was a particular concern. The point here is that the change in the lab's meeting schedule was not simply a 'response' to open science; it also reflected a shift in practices to accommodate the growing complexity of plasma codes.

On many fronts the lab embraced the vision of open science and cyberinfrastructure. Developers worked to create framework codes and 'trusted components' that were general purpose and that could be used to quickly make new codes—efforts that accorded with the tiered approach to cyberinfrastructure described in the previous section. They also sought to make their software more sustainable by adopting professional practices, as just described. Finally, several new codes were released under open source licenses during the year of my

fieldwork. However, there remained one important exception. The lab's oldest and best known code, PULSE, was 'opened' only in a limited sense. As explained in the beginning of the chapter, the lab chose to make it available through MoU. The reluctance to fully 'open' PULSE, as I show next, stemmed from the developers' concerns that the considerable credibility (or "brand recognition," as one interviewee put it) the code had achieved in the community would be threatened if the code were widely distributed.

In a hard-line open science perspective, one might say that if a researcher is not willing to open their code to public scrutiny, then whatever value it appears to have (its "brand recognition") is not justified. But that's not fair in this case. PULSE, has been used in dozens of studies in a variety of research areas. Its success in the past occurred through informal cooperation between trusted colleagues. In the case of PULSE, the lab sought to to maintain this style of user-developer interaction. Simply put, it takes considerable expertise to "get good results" with a plasma simulation code—"good results" are results that represent the codes best approximation of the experimental setup. The degree to which a particular simulation corresponds to a particular experimental setup can be a matter of dispute. In particular, developers may take issue with users' claims that they are simulating what they think they are simulating. In the case of PULSE, the developers were worried that 'misuse' of the code along these lines might negatively impact the credibility that the code had achieved.

3.4.1 Crafting Input Decks

The challenges associated with the use of simulation codes are illustrated through considerations of the "input deck", the basic interface through which simulationists define the simulation's inputs (the phrase "input deck" is somewhat of anachronism, as it refers to a time when the code and it's input existed as a deck of punch cards). An input deck is a structured text file (see below) that defines the initial parameters for the simulation—it is the primary interface by which users interact with the code and configure the simulation. A typical input deck defines values pertinent to the particle-in-cell method—grid dimensions,

boundary conditions (i.e., how to deal with the edges of the simulation space), number of particles, initial particle distribution and velocity, etc.—as well as computational details, such as how many "nodes" to use. There are potentially dozens of configuration options to consider, and they can interact with one another in ways that are difficult to predict, even for experts. Producing an input deck, particularly, for real-world simulations can be difficult. One lab member described the process as error-prone—even as a senior physicists with deep familiarity with the code, he admitted he would probably make a mistake if he had to write an input deck from scratch. Instead, he prefers to start with a deck that works and modify it as needed. An excerpt of an input deck is shown below (this is for a 'test' run, not a real-world example):

```
!-----nodes and boundary conditions------
node_conf
{
  node_count(1:2) =  1,1,
  do_periodic(1:2) =  .true., .true.,
}
!-----grid config----------
grid
{
  nx_p(1:2) =  100, 100
  coords = "cartesian",
}
!-----time step and global data dump timestep----------
time_step_config
{
  dt     =   0.04d0,
  dumpn  =    10,
}
```

At the PULSE workshop, the discussion during a session on "best practices" focused on how to craft and distribute input decks. Workshop participants expressed some bewilderment regarding the general rules and strategies for designing input decks, however others were skeptical that a rule-based approach would be appropriate. Understanding input deck parameters, their effects and how they interact with each other is "an art", as one participant put. "There will never ever be a list of rules to follow to get the correct parameters. It will always be an art... It comes from experience". The discussion turned to strategies for sharing and communicating input decks so the "experience" of others could be built upon. Different ideas were considered: there should be a repository of input decks that recreate the simulations of 'classic' papers in the field; there should be a repository of *problematic* input decks that demonstrate spurious, non-physical numerical effects and other "traps"; researchers should publish their input decks to make their research reproducible. Many of these suggestion were met with counter arguments. In particular, more senior members of the workshop were concerned that easy access to input decks might encourage a use of the code that was extensive but not intensive.

> "You can sometimes be tempted to take an existing deck and just tweak one thing without really understanding the physics [of the computational model]"

> "There needs to be a balance between [resources for] learning the tool, quickly getting useful results, and being an expert that can get *good* results."

These comments—made in a public setting to colleagues—contrast somewhat to the admission, above, that even experienced simulationists often prefer to "tweak" existing input decks. The point is that, while "tweaking" (working iteratively from existing decks) was a common practice for both experts and non-experts, the experts worried that it was not sufficient for acquiring deep understanding: "The worry is that people mistakenly believe it's plug-and-play", as one workshop participant commented. These comments present a structure of expertise between the kind of understanding needed to quickly get useful results—in a sense, the ability to use the 'black box' effectively—and 'real' understanding of the code's internal computational model.

The expertise associated with one simulation code does not necessarily carry over to other codes. Plasma simulations codes vary extensively from one code to the next and the parameters that are used in one code are not like to have the same effect in another. One workshop participant described for example, that codes that use certain numerical approaches (those with "low order particle shapes") are known to have problems with "numerical heating" so simulations have to be carefully crafted to account for that effect; if you were to simply transfer parameters from such a code to a code with "high order particle shapes" then the simulation would show different physics. Because the understanding of just one code is difficult, effective comparison across multiple simulation codes are also difficult. One workshop member recalled that for some problem areas it has taken as long as ten years for the sufficient understanding of different codes to develop to the point where they can be compared.

3.4.2 'Developers are the best users'

How does one acquire "real understanding" of a code and the capacity to "get good results"? Unsurprisingly, lab members felt that, "as developers of the code, we are also the best users of the code." One way to approach this statement is to recognize software development as an effort to both create a code and *understand* that code. Simulation codes, like other complex scientific instruments, are "self-created" but not "self-evident" (Knorr-Cetina 2009)—accordingly, the development process necessitates a continuous inquiry into the system being produced. Practices such as benchmarking (discussed in the next chapter in relation to the lab's reference codes) is one aspect of this. But the main method through which developers come to understand what they have developed is *testing*. Here this term should be understood in both a conventional sense—the everyday strategies that simulationists deploy to make sure their code is working—and a technical sense—as in "integration testing", a term of art for managing the code development process so that tests are systematically and automatically performed as the code evolves. Simulation developers tend to have their own individualized ways of testing the code, often oriented toward the particular kinds of problems they are concerned with.

Fordham: [...] we all have some simple test cases that we try whenever we change the codes for any particular purpose. And for me in particular, I've been using the same exact test problem which was the first problem I ran when I came to [the university some years ago]. [...]. I can just look at it at a glance and I'll know exactly if there's something wrong with it.

Seth: Okay, what's the test case?

Fordham: It's basically two plasma clouds, one made up of electrons the other made of positrons, they collide, this is a 2D run, and they filament and do things. You get these very clear rings... Ring structures of magnetic field. And these islands and magnetic fields pushes away the plasma, so in those regions you only have magnetic fields and the plasma sits inside or outside the magnetic field...

Seth: So there's a particular behavior that you're very familiar with then you want to see that behavior happen in the simulated plasma.

Fordham: And it's a very quick test that I can do, and if I broke the codes, it will show up.

The essential quality of these tests is that the developer has a theoretically informed expectation of how the code *should* behave in a particular situation. The developer's understanding of the code is the product of repeated interactions with the code in which the link between the input deck and the "test problem" is reinforced. Through such repeated testing, the developer acquires a 'deep understanding' of the code. Developers also challenge each other's conceptions of the code. Disputes at the level of algorithm selection, for example, concern the degree to which the code functions as an effective model of a target system.

Such a problem arose in the implementation of PULSE's 'thermal bath' boundary condition. In a typical PIC simulation, particles continuously move in and out of the simulation window. While the part of the plasma outside the simulation window is not directly simulated, it must be modeled to some degree in order to determine when new particles should

enter the simulation window and what their properties should be (e.g. their velocity, charge, etc). There are different ways to model the simulation boundaries—in a thermal bath, particles leaving the simulation window are replaced with new particles with attributes (e.g. velocity and charge) selected some a probability curve. If the velocities and charges for the new particles are chosen correctly, the total energy of the plasma will be steady—i.e., the plasma will not be artificially driven to lower or higher energy levels. In PULSE's implementation of the thermal bath boundary condition there was issue with the distribution from which new particle velocities were selected—in certain, very rare instances, numerical non-physical effects occurred.

> Fordham: "[The thermal bath issue] was basically a discussion between me and [a colleague], which I lost. I should say that right now. And [the colleague] was absolutely right. [...] both of us had sound arguments and honestly, it was only when someone found a very simple test problem where if you run the code for a long time, you would see a really significant difference and you could say, 'No. What I expect the code to do is what [the colleague's algorithm] is doing and not what [my algorithm] is doing.' But before that, the way we settled it was we pretty much put the two options [in the input deck]. So for a long time you had two options there to do it one way or the other way, but when someone showed, 'Now listen, there is this test case, I think this is the correct one,' and, faced with that [the matter was settled]."

The thermal bath issue was not a 'bug' in the traditional sense because the problem didn't stem from an error in Fordham's implementation of his algorithm. The problem was rooted in the choice of the algorithm itself and whether that algorithm provides the "expected" representation of a thermal bath. In formal system engineering terminology, the problem was not one of *verification* (the algorithm may have been well implemented), it was one of *validation* (it failed with respect to some real world use of the code). As this example demonstrates, disputes over a code validity are highly *subjective*, particularly for simulations (Easterbrook and Johns 2009). It was only with some effort that Fordham's colleague was

able to convince him that his approach was mistaken. This example also demonstrates how parameters in input decks can multiply (the issue was temporary settled by having two option in the input deck) and how understanding of a parameter's effects can change over time.

Generally speaking, simulations are difficult to test because their very purpose is to give insights into phenomena that are difficult to understand analytically and/or empirically. Physicists would not need simulations if they always new what to expect from them. In addition to tests against theory, confidence in simulation emerges as the code is compared to experimental results and to other simulations. The plasma simulation lab remained in close contact with developers of other simulation codes and significant discrepancies prompted careful examinations of the underlying causes. Trust in simulations, like trust in experimental results, is the product of repetition. In the following, I turn to the problem of 'using' simulations and the regress-like situations that occur in disputes between users and developers over whether the simulation 'is working'.

3.4.3 Tensions Between Users and Developers

Short of being a developer, the simulationists I talked to acknowledged that the best way to understand a code was to spend time with its developers. During the PULSE workshop, for example, the PIs encouraged users to visit their respective labs in order to work out details of how to incorporate PULSE in their projects. This approach was particularly important for using newer aspects of the code, such as recent features for simulating effects of quantum electrodynamics—"Someone who knows the code needs to sit down for a few months to explain [these new features]—that's what's necessary for a group to get familiarity."

During the year of my fieldwork, the lab hosted two visiting researchers. One was a student with prior development experience, who came to the lab to develop a new code for his research project—he was taking advantage of the lab's general expertise in plasma simulation development. Another, who I'll call Sarah, was a student with less development experience. Sarah split her time between the lab and a group of experimenters at the same

university. As she described her involvement in the lab:

> Sarah: "I only have one slight difference from most people [in the lab], that is that my background is not computer science as much as it is physics. So I am maybe addressing physical problems and *using* the code more than *developing* it—more than coming to say, 'I'm going to add a new module to the code'... I came saying, 'I'm going to study this and this and [we] will probably have to change the code so that this and this will be solved'"

Sarah recognizes the distinction between developers and users to be one of "background", training, and the degree to which one's work foregrounds problems of "computer science" or "physics." (To be clear, while many members of the lab that I talked to had experience and some academic training in computer science and software engineering, few had completed degrees). Nevertheless, her conception of "using" a code includes close cooperation with the developers and the recognition that using a code often necessitates its modification. As we will see, this was a mode of use that the developers in the lab preferred.

It is important to note that Sarah split her time between two labs—the simulation group and another University group conducting experiments in Sarah's research area. Sarah's "use" of the code was directed by the needs of the experimenters—she ran the the code for them. "I am the interface that communicates all [the experimenters'] questions to [the simulation group], I also am the one who gives [the experimenters] the results, saying 'this is what I got from what you gave me.'" The experimenters use of the code was a conventional one for plasma simulation code—they used the code to help them feel more confident about the results they saw in their experimental setup. Experimenters would also use the code to quickly search the 'parameter space' of possible experimental setups to identify regimes in which a particular phenomenon or experimental effect was maximized.

Sarah's role as an "interface" between the experimenters and the simulationists was difficult at times because the experimenters often ran into difficulties with the codes, and they would sometimes voice their concerns in a manner that sounded, to the simulationists, like an aspersion. With some tact, Sarah illustrated the situation as follows:

Sarah: "Imagine that you are running your experiment. You get the output from the simulation and it looks different. And someone from the simulation group will say, 'well, in your case, we would have to pay attention to this detail, so we should have actually changed this in the code and maybe it would work better'. And the experimental person will so, 'OK, so the problem is the code. It's not physics and it's not *my* experiment. So *you're* code is wrong.' And if they are too—how can I say—careless in the way that they express this .... [the simulationist], who has been working on this code for many years, day and night will feel a bit resentful. So that's the personal part. But for the more practical part. ... If you're running an experiment, you're very focused on building the right set of lenses and the right setup. You are constantly thinking about this and worrying about your actual experiment. And you get your results. You do that analysis for a few months, and you try to think about it, physically and everything. So when you go to the simulation, since you never really had to look into the deeper layers of the code, or really had to look at the code, you can have this idea that you just say to the code, 'I have a plasma like this, I have a laser like this. Tell me what happens in the end'. And you just expect it to come. ... And maybe you don't realize all the work that is behind [the code], and you quickly assume things about the code. And if you're the kind of person that has the tendency to assume, you'll just say 'OK, the code doesn't work. Finished. Let's go. move on, do another thing.'. I guess [a simulationist] might label that as someone who doesn't know what they're talking about in the sense that they don't know what's going on with the code, or how the code works."

The difference between experimenter-user and simulationist-developer entails a difference in the objects in which one is professionally invested. As Sarah's comment illustrates, simulationists were sometimes frustrated with experimenters' instrumental treatment of the codes. However, Sarah was also sympathetic of the experimenters' concerns: they have their own craft, their own world of faulty equipment, contingencies, and justifications to attend to. If the simulation cannot be relied upon to help them feel more confident about their

experimental work then it is only a source of more troubles.

What we see in this story is a conflict over how to determine whether or not a "code is wrong." The experimenter regards any necessary code modification as evidence that the code is wrong, while the developer recognizes such changes to be an inherent part of simulation: "[if] we changed this in the code [...] maybe it would work." Code adjustments and 'tuning' of the code are a routine part of what it means to do simulation, from the developers' point of view. Disputes like these were sometimes discussed in laboratory meetings. Amongst themselves, simulationists responded quite heatedly to accusations that the code was broken or that it couldn't handle a specific condition. In particular, simulationists thought that users were not in a position to asses the quality of the code as a whole. Bruce, one of the senior members of the simulation group, put it this way:

> Bruce: "When people are using the code and they're not feeding back to those of us who have the bigger picture, then they'll say 'the code can't do this' or it's always over estimating this or it's giving them the wrong answer—when the reason could be a more fundamental reason and not that there's an issue with the algorithm in the code [...]. [There are people] using the code and they'll say this, and [we] just get....I mean you can't make a statement like that. Talk to us about what's going on, we may know why [the code works as it does].
>
> Seth: Is it a misuse of the code?
>
> Bruce: Or experimental parameters are not known well enough... or there *is* a numerical issue, but we know certain parameters are more sensitive to [numerical effects].

Only the developers, Bruce argues, are in a position to say why the code behaves as it does. The issue is not whether the code is error-free. It's that claims about the relationship between the experimental setup and the simulation setup is the domain of the simulationist's expertise. As Bruce further explained, this user-developer regress constituted a significant problem for making codes open source:

> Bruce: [Hypothetically speaking], you make the code open source. They publish with your code. They may not know how to use it. But that bad result, or piece of crap result, is associated with [PULSE]. So one of the issues many of us have when we give our codes away is this idea that you use our code and say "that's what [PULSE] produced", rather than—you know this argument—"guns don't kill people, people kill people"... I'm not buying that, but anyway...

In comparing simulation codes to guns—and, by extension, his lab to a gun company—Bruce finds himself caught uncomfortably between his belief that users should take more responsibility for the results they get with a code and his distaste for a slogan used in arguments for looser gun regulations. Through the analogy, Bruce finds himself in the position of the gun maker who would rather not have the actions of 'users' reflect on the 'brand' of its product. 'The code isn't wrong, only people's use of code is wrong', he wants to say.

While there is some risk of pushing the analogy between codes and guns too far (for example, the gun debate has little to do with whether or not guns are 'working' in a technical sense), it provides a useful point of comparison for thinking about how agency, responsibility, and blame are divided among technologies, users of a technology, and developers. If the lab is a gun maker, it is a strange kind of gun maker that would prefer to be involved in each use of its product—"[t]alk to us about what's going on", Bruce says. Indeed, this attitude toward its codes is distinct from broader social attitudes toward software, in general. "These codes are *research* codes," as Bruce commented. "They're not apps that we're sending you, [where] if it breaks then it's our fault. We're not getting payed enough to do that." Unlike commercial software and 'apps', simulation code require constant reflective use, and they resist instrumental use. Ultimately, the lab's argument for keeping codes like PULSE—codes with considerable "brand recognition"—closed was that the risks to the credibility associated with openness were too great. Wider cultural attitudes toward software as a 'black box' work against the kind of relationship that simulationists would prefer to foster. The salient quality of the comparison between simulation codes and guns is that it suggests a reversal of the assumed benefits of openness and accessibility.

3.4.4 The User-Developer Regress

Software is often valued for its portability, however, as the preceding description of disputes between plasma simulation developers and simulation users illustrates, the expertise required to effectively use simulation codes can be stubbornly localized.

The trouble of determining whether a given code is 'working', and the disputes that follow from this difficulty, parallel a similar set of concerns in experimental work, as identified by sociologists of science. Collins' notion of the experimenter's regress describes a similar paradox: often, the only criterion for determining whether an experiment is working is the experiment itself. This is particularly evident in 'frontier' research, such as gravity wave detection (at the time of Collins' writing): "What the correct outcome is depends upon whether there are gravity waves hitting the Earth in detectable fluxes. To find this out we must build a good gravity wave detector and have a look. But we won't know if we have built a good detector until we have tried it and obtained the correct outcome! But we won't know what the correct outcome is until... and so on *ad infinitum*" (Collins 1985, 84). As Collins articulated it more recently, "[t]he experimenter's regress starts with the observation that experiments, especially those on the frontiers of science, are difficult and that there is no criterion, other than the outcome, that indicates whether the difficulties have been overcome" (Collins 2005, 457). A consequence of this regress is that experiments can only function dispositively if the cycle is broken—i.e, by finding means other than the experiment itself to asses experiment's effectiveness.

Gelfert (2012) has proposed the "simulationist's regress" to describe the analogous concern, "where the best or only test of a simulation is its own disputed result." As with experimentation, disputes associated with this kind regress are prominent in simulations of novel phenomena (like black holes forming from collapsing binary neutron stars) or in situations where there is limited data for directly comparing to simulation results (like climate modeling). However, the disputes between plasma simulation users and developers described in this chapter do not stem from difficulties of "frontier" science or from a lack of data. The phenomena being simulated (e.g., laser wakefield acceleration) are not novel, their existence

is not disputed. Further, experimenters use plasma simulation code precisely to *break* the cycle of the experimenter's regress: hopefully the simulation will give them confidence that their experimental setup is 'working.' There is no problem, here, of missing data to compare the simulation results to. Disputes arise when the code fails to replicate fairly well-known, "expected" aspects of an experimental setup (e.g., the emittance of a beam generated in a wakefield).

Nevertheless, there is are regress-like situations that arises from the lack of an agreed upon procedure for setting up a simulation to accord with a given experimental arrangement. As we have seen, crafting input decks is an "art" that depends on experience and "deep understanding" of the code. Given the challenge of getting such codes "to behave properly at all" (Kennefick 2000), the question of whether the difficulties have been overcome is never far from the simulationist's mind. The user-developer regress consists of the observation that research codes are difficult for non-developers to use effectively (i.e., in a way that reflects well on the code) and yet they must be used extensively (and effectively) for the code to be recognized by the broader community. From the example of PULSE, one approach for 'resolving' the user-developer regress is to organize users as a 'closed' community, small enough for developers and users to build relationships of trust and the shared understanding that are required to use the code effectively.

3.5 Conclusion

Cyberinfrastructure and open science identify software's value in its ability to move beyond a community of developers. As infrastructure, software's promise rests in the efficiencies brought with greater resource sharing and the adoption of common standards across domains (Atkins, 2003). In open science, the broad availability of research software is a necessary condition of supporting reproducibility and the integrity of computational science (Donoho et al. 2009). Against the background expectation that successful software is software that is taken up by an extensive community of users, codes like PULSE are somewhat puzzling. The lab's decision to manage the growth of PULSE was precisely rooted in a desire to maintain

its value. The user-developer regress was used to explain the apparent paradox between developers and users. This discussion of plasma simulation development complicates the vision of open science by showing that accessibility and value did not coincided in the minds of the simulationists.

Another important conclusion from this chapter concerns the complex role of the developer in the *use* of a code. Developers preferred to be involved in the use of their codes and they played an important role in mediating and explaining simulation results. The user's (non-expert's) trust in the code was ultimately rooted in a social relationship with the developer. Importantly, the lab chose to release its *newer codes* (those without an existing community of users) with open source licenses. A question for future research is whether open simulation codes are more prone to the kinds of disputes associated with the user-developer regress. Generally, it seems reasonable to consider how the communities associated with these distinct categories of code will differ.

3.6 Bibliography

Atkins, Daniel. 2003. "Revolutionizing Science and Engineering Through Cyberinfrastructure: Report of the National Science Foundation Blue-Ribbon Advisory Panel on Cyberinfrastructure," January.

Baker, Monya. 2016. "1,500 Scientists Lift the Lid on Reproducibility." *Nature News* 533 (7604): 452. doi:10.1038/533452a.

Becher, Tony, and Paul Trowler. 2001. *Academic Tribes and Territories: Intellectual Enquiry and the Culture of Disciplines*. Society for Research into Higher Education & Open University Press.

Benkler, Yochai. 2006. *The Wealth of Networks: How Social Production Transforms Markets and Freedom*. Yale University Press.

Bird, Ian. 2011. "Computing for the Large Hadron Collider." *Annual Review of Nuclear and Particle Science* 61 (1): 99–118. doi:10.1146/annurev-nucl-102010-130059.

Collins, Harry. 1985. *Changing Order: Replication and Induction in Scientific Practice*. University of Chicago Press.

———. 2005. "Replication." In *Science, Technology, and Society: An Encyclopedia*, edited by Sal P. Restivo. Oxford University Press.

David, Paul A. 2006. "From Keeping 'Nature's Secrets' to the Institutionalization of 'Open Science'." In *CODE:Collaborative Ownership and the Digital Economy*, edited by Rishab Ghosh, 85–108. MIT Press.

David, Paul A., Matthijs Den Besten, and Ralph Schroeder. 2006. "How Open Is E-Science?" In *E-Science and Grid Computing, 2006. E-Science'06. Second IEEE International Conference on*, 33–33. IEEE.

Donoho, David L., Arian Maleki, Inam Ur Rahman, Morteza Shahram, and Victoria Stodden. 2009. "Reproducible Research in Computational Harmonic Analysis." *Computing*

in Science & Engineering 11 (1): 8–18. doi:10.1109/MCSE.2009.15.

Easterbrook, Steve M., and Timothy C. Johns. 2009. "Engineering the Software for Understanding Climate Change." *Computing in Science & Engineering* 11 (6): 64–74. doi:10.1109/MCSE.2009.193.

Gelfert, Alex. 2012. "Scientific Models, Simulation, and the Experimenter's Regress." In *Models, Simulations, and Representations*, edited by Paul Humphreys and Cyrille Imbert. Routledge.

Goble, Carole. 2014. "Better Software, Better Research." *IEEE Internet Computing* 18 (5): 4–8.

Hey, Tony, and Mike C. Payne. 2015. "Open Science Decoded." *Nature Physics* 11 (5): 367–69.

Howison, James, and James D. Herbsleb. 2011. "Scientific Software Production: Incentives and Collaboration." In *Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work*, 513–22. CSCW '11. New York, NY, USA: ACM. doi:10.1145/1958824.1958904.

Howison, James, Ewa Deelman, Michael J. McLennan, Rafael Ferreira da Silva, and James D. Herbsleb. 2015. "Understanding the Scientific Software Ecosystem and Its Impact: Current and Future Measures." *Research Evaluation* 24 (4): 454–70.

Ioannidis, John P. A. 2005. "Why Most Published Research Findings Are False." *PLOS Medicine* 2 (8): e124. doi:10.1371/journal.pmed.0020124.

Kelty, Christopher M. 2012. "This Is Not an Article: Model Organism Newsletters and the Question of 'Open Science'." *BioSocieties* 7 (2): 140–68.

Kennefick, Daniel. 2000. "Star Crushing: Theoretical Practice and the Theoreticians' Regress." *Social Studies of Science* 30 (1): 5–40. doi:10.1177/030631200030001001.

Knorr-Cetina, Karin. 2009. *Epistemic Cultures: How the Sciences Make Knowledge.* Har-

vard University Press.

Latour, Bruno, and Steve Woolgar. 1986. *Laboratory Life: The Construction of Scientific Facts*. Princeton University Press.

Levin, Nadine, and Sabina Leonelli. 2017. "How Does One 'Open' Science? Questions of Value in Biological Research." *Science, Technology, & Human Values* 42 (2): 280–305. doi:10.1177/0162243916672071.

Merton, Robert K. 1942. "The Normative Structure of Science." In *The Sociology of Science: Theoretical and Empirical Investigations*. University Of Chicago Press.

Munafò, Marcus R., Brian A. Nosek, Dorothy V. M. Bishop, Katherine S. Button, Christopher D. Chambers, Nathalie Percie du Sert, Uri Simonsohn, Eric-Jan Wagenmakers, Jennifer J. Ware, and John P. A. Ioannidis. 2017. "A Manifesto for Reproducible Science." *Nature Human Behaviour* 1 (1): 0021. doi:10.1038/s41562-016-0021.

National Science Foundation. 2012. "A Vision and Strategy for Software for Science, Engineering, and Education: Cyberinfrastructure Framework for the 21st Century."

Porter, Theodore M. 1996. *Trust in Numbers: The Pursuit of Objectivity in Science and Public Life*. Princeton University Press.

Schroeder, Ralph. 2008. "E-Sciences as Research Technologies: Reconfiguring Disciplines, Globalizing Knowledge." *Social Science Information* 47 (2): 131–57.

Shapin, Steven, and Simon Schaffer. 1985. *Leviathan and the Air-Pump*. Princeton University Press Princeton, NJ.

Stewart, Craig A., Guy T. Almes, and Bradley C. (eds) Wheeler. 2010. "Cyberinfrastructure Software Sustainability and Reusability: Report from an NSF-Funded Workshop." Bloomington, IN: Indiana University.

Stodden, Victoria. 2010. "The Scientific Method in Practice: Reproducibility in the Com-

putational Sciences."

Stodden, Victoria, and Sheila Miguez. 2014. "Best Practices for Computational Science: Software Infrastructure and Environments for Reproducible and Extensible Research." *Journal of Open Research Software* 2 (1): 21. doi:10.5334/jors.ay.

Traweek, S. 1992. *Beamtimes and Lifetimes: The World of High Energy Physicists*. Harvard Univ Pr.

Willinsky, John. 2005. "The Unacknowledged Convergence of Open Source, Open Access, and Open Science." *First Monday* 10 (8). doi:10.5210/fm.v10i8.1265.

Wilson, Greg, D. A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven H. D. Haddock, et al. 2014. "Best Practices for Scientific Computing." *PLoS Biol* 12 (1): e1001745. doi:10.1371/journal.pbio.1001745.

CHAPTER 4

Commonplaces of Plasma Simulation

4.1 Abstract

The physicists differentiated between several kinds of simulation codes: production codes, reference codes, frameworks, classroom codes, and toy codes. These categories cannot be differentiated on a purely 'algorithmic' basis because the same basic model (Particle-in-Cell) is found across them. This chapter treats the categories as a set of commonplaces the lab used to differentiate its development activities. In particular, the categories are used to explore the relationship between objects of science and the technical conditions through which those objects are produced and studied. I argue that the lab's production codes fit within Merz's conceptualization of simulations as "multiplex and unfolding", however the intentions associated with reference codes, frameworks, and classroom codes are more aligned with technical objects. Collectively, the categories described the poetics and pragmatics of plasma simulation.

4.2 Introduction

A physics student's first encounter with plasma simulation might look something like Figure (4.1), which shows the interface of an application used in 'laboratory' assignments in an undergraduate plasma physics course. The simulation is a simple one—it shows a single charged particle (the dot in the center) moving through a magnetic field. Lines designate the contours of the magnetic fields. The program models the trajectory of the charged particle as a rotation around a central path, called the "guiding center." The screen space

is mostly devoted to the three-dimensional scene in which the particle moves in response to the magnetic field. In the top-right corner we see a collapsible window with settings for controlling the simulation. The student can arrange the scene with magnetic field structures commonly used in physics instruction (e.g., infinite wire, Helmholz coil, or dipole); they can adjust the field current as well as the particle's initial position, velocity, and charge/mass ratio. In the top-left corner there is a small graph showing the number of frames per second at which the whole scene is rendered, a subtle reminder of computational constraints. There is a button to stop and start the simulation at the bottom.

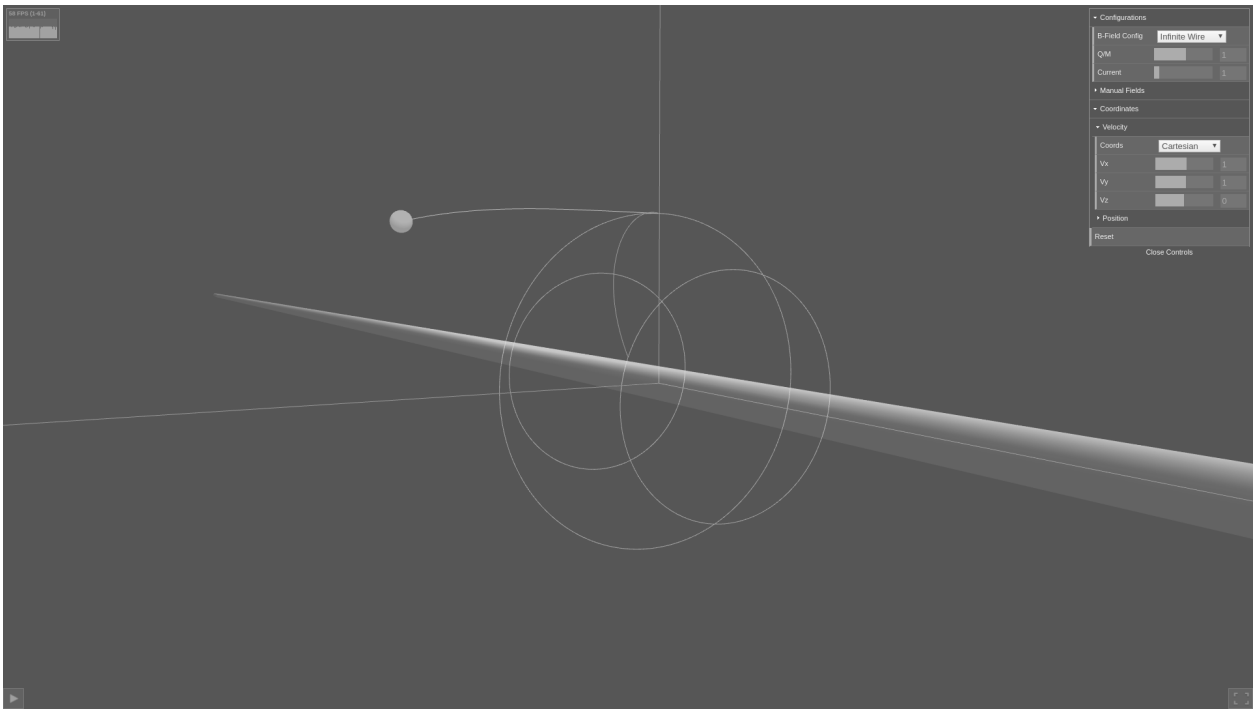

Figure 4.1: Lars' Guiding Center Simulation (fieldnotes)

The graduate student who made the program, Lars, gave a demonstration during one of the group's weekly meetings. Bruce, the lab PI, asked Lars to demonstrate a few classic "drift" configurations:

Bruce: Show me E-cross-B

Lars changes some settings, and the particle begins to trace a cork-screw pattern.

Bruce: What about grad-B?

Again, Lars changes some settings, the particle starts on a new trajectory... A few exchanges later, the red dot is suddenly not behaving as expected—it appears to be moving at quite a clip away from where it should be [it's unclear what I'm seeing]. Lars looks confused and maybe a little embarrassed, unsure if it's a bug or if he has accidentally mis-configured something. He suggests that the charged particle might have accelerated too much and departed the field area.

Sean [a senior member of the group] (laughing): That's the story of fusion!

Later, Bruce comments that, indeed, the program is useful because students can push the parameters to theoretical limits, to the point that the program breaks

This lab meeting interaction might be used to illustrate several well-known aspects of scientific modeling and simulation practice. Perhaps the most recognizable of these is the physicists' capacity to recover something of value even in a moment when the representational accuracy of the simulation appears to be uncertain. Lars' momentary confusion (being unsure if the particle's behavior was the result of a bug or his own misconfiguration of system) was not met with overt mistrust in his code. On the contrary, it was met with Sean's (admittedly joking) recognition of the 'the story of fusion' in the particle's strange behavior and Bruce's observation that such 'breakages' are good ways for students to get a sense of the limits of physical theories. This feature of scientific work—the idea that the utility of a representation is not rooted in its accuracy, truth, or realism—is well accounted for in philosophy of science (Hacking 1983; Wimsatt 2007). It is also an insight that has influenced recent work on scientific modeling and simulation, work that replaces representationalism with an 'artifactual' approach that emphasizes productive ambiguity, the generativity of material constraints, and the multiple functional contexts within which models are deployed (Knuuttila, Merz, and Mattila 2006; Knuuttila 2011). "Instead of accepting the apparent misrepresentation as a defect—which results from taking accurate representation as a criterion of knowledge—one might consider the motivations and possible cognitive gains of

the purposeful misrepresentation that is characteristic of modelling" (Knuuttila 2011, 270). While the strange behavior during Lars' demonstration was not a "purposeful misrepresentation", it would become so in the scenario that Bruce suggests: pushing the program to the point that it breaks as a way for students to explore theoretical limits. Further, if a model's representational capacities are an achievement of their construction and manipulation, then Sean's joke about fusion is an expression of such a practice. The joke is premised on a way of associating the behavior of the 'phenomenon' to the behavior of the simulation. This relationship is not a given. Like an acquired sense of humor, it is an achievement rooted in a certain category of experience. In this case, the experience of building and working with simulations.

There is another set of considerations that follow from this example, however. What, for instance, should be made of the pedagogical aspect? Lars' demonstration was one of the few instances I observed in which a simulation was actually carried out *in* a lab meeting. Typically, simulationists work with codes in the confines of their private work spaces—it's not an activity that is exposed to public scrutiny. The intention of using the code to illustrate aspects of plasma physics to undergraduate students is relevant to our understanding of "misrepresentation" in this instance. In the demonstrative context of a classroom, unexpected behaviors may not be so generative. Further, Lars' program is unlikely to lead to a publication—at least not a research finding in his field. It may, at most, be acknowledged in documents like CVs, course syllabuses, websites, and (mostly importantly) reports to funding agencies. Even in the lab, *classroom codes* like this one were secondary in comparison to the "real" codes, so called *production codes*, that the physicists devoted most of their attention to. Unlike classroom codes, production codes are painstakingly tailored to fully exploit the capabilities of the world's fastest supercomputers, and they are oriented toward rather specific areas of research. They certainly do not run on a PC's web browser and they most certainly do not have user-friendly graphical interfaces (c.f., the previous chapters' discussion of 'input decks'). Production codes are *serious*. classroom codes are not.

Does this mean classroom codes, or other secondary categories of code, are epistemologically irrelevant within simulation practices? This chapter argues to the contrary. While

production codes were unquestionably central to the lab's work, that privileged position was achieved by bracketing a variety secondary concerns (such as undergraduate instruction) and designating supporting codes to satisfy those concerns. Rather than focus exclusively on the charismatic megafauna of plasma simulation—production codes—this chapter aims at a more eclectic characterization of the ecology of software practices in the lab. The object of analysis is the five-part categorization: production code, reference codes, framework code, classroom code, and toy code. By tracing the distinctions between these terms, I show that they collectively constitute an informal set of commonplaces that structured simulationists' relationship to software. Further, these terms strongly relate to efforts of from funding agencies to make scientific computing more 'open.'

The code categories were explicitly invoked by lab members—most were used in documents (with the exception of toy codes), to organize lists of the labs' software. Toy codes are a less clearly defined category that emerged during interviews. I include toy codes because, like classroom codes, reference codes, and frameworks, they are an overlooked category of simulation work. Production codes—codes used in research—are (understandably) the focus in most case studies of simulation practices from science studies and philosophy of science. It is also important to emphasize the *local* quality of this categorization. I have not observed these terms used outside the context of the lab (although, as we will see, the use of terms like production, reference, and framework, have general meanings in software development). That is not to say that the distinctions I describe are not applicable to other settings.

The axis of difference that this categorization describes (which cuts across rather than along divisions between fields of study) resembles Rheinberger's (1997) distinction between epistemic things and technological objects. Knorr-Cetina (1997) and Merz (1999) have questioned the separation between these terms, arguing that epistemic things are sometimes also technical objects. Merz's argument is particularly important as it focuses on the problem of "the kinds of objects simulations are." Merz argues that 'event generators,' a kind of simulation used in high energy physics, are "multiplex": their epistemic roles vary from context to context. This conceptual framing—Rheinberger's distinction between epistemic things and technical objects and Merz's subsequent elaboration of simulation codes

as multiplex—continues to be relevant for understanding community divisions in scientific computing (Sundberg 2009; Spencer 2015). To make sense of the lab's code categorizations, I argue that the production codes fit within Merz's conceptualization of simulations as "multiplex", however the intentions associated with reference codes, frameworks, and classroom codes are more aligned with the role of technical objects. Whereas the trajectory of production codes occurs in the context of research itself, the use of these latter categories involves practices such as classroom demonstration, benchmarking, and the rapid development of trusted components. Toy codes stand out as a difficult and mercurial concept, which suggests they are also multiplex, though in a manner distinct from production codes. Finally, we can understand these categories as involving distinct memory practices, as described in chapter two.

In the following section I elaborate Merz's notion of simulation codes as multiplex and explain the consequences of this argument for Rheinberger's understanding of the temporal structure of experimental systems. I also provide a brief description of the Particle-in-cell (PIC) model that is relevant for the final discussion of the individual code types.

4.3 The Multiplexity of Simulation Codes

The question to consider in this section whether the 'multiplexity' of simulation codes undermines the temporal structure of computational science constituted by the distinction between epistemic things and technical objects (Merz 1999). In other words: Does Rheinberger's characterization of the "movement" of scientific inquiry, which is premised on that distinction, lose its coherence if technical objects (like simulation codes) are also epistemic things?

To set the background, it is helpful to underscore the role of Bachelard's historical epistemology in Rheinberger's argument. Bachelard characterized scientific 'progress' as a dialectic between conceptualization and materialization (Rheinberger 2005b). This is not a teleological argument—'progress' for Bachelard "means permanent goings-on, a continuous movement of differential reproduction rather than a movement toward a preconceived end" (*ibid*, 314).

Bachelard's approach to science stands apart from others because it emphasizes practices of construction rather than representation (or re-presentation *as* construction). This emphasis is encapsulated in the term *phenomenotechnique*, which is meant to indicate the throughly constructed nature of scientific objects. Natural phenomena are not *given* (i.e., by nature). They have an inherently technical character—and because they have a technical character, they have an inherently historical character. "Scientific objects are always transformations of earlier scientific objects and thus intrinsically historical entities" (*ibid.*, 325).

These considerations carry over into Rheinberger's own account of the temporal structure of experimental systems, in which "sufficiently stabilized epistemic things turn into the technical repertoire of the experimental arrangement" (Rheinberger 1997, 29). New technical objects constitute new conditions for novel epistemic things. The distinction between epistemic things and technical objects is a gradient—a potentiality—that propels the experimental system. While "exchange," "hybrids," and "blending," are possible, Rheinberger insist that "the distinctness is clearly perceived in scientific practice"; further, this distinctness is important as it "it helps to understand the occurrence of unprecedented events and with that the essence of research" (*ibid.*, 30-31).

However, as already described in the introduction to the dissertation, the technical conditions of contemporary science are often unstable—they demand constant care and revision, and each revision renews the effort to understand what they have become (Knorr Cetina 2001). Knorr-Cetina suggests a modification of Rheinberger's temporal dynamic in which the driving tension of research shifts from the distinctiveness of epistemic things and technical objects to the non-self-identity of epistemic objects. Epistemic objects have an "unfolding ontology."

> "[T]he defining characteristic of an epistemic object is this changing, unfolding character—or its lack of 'object-ivity' and completeness of being, and its non-identity with itself. The lack in completeness of being is crucial: objects of knowledge in many fields have material instantiations, but they must simultaneously be conceived of as unfolding structures of absences: as things that

> continually 'explode' and 'mutate' into something else, and that are as much defined by what they are not (but will, at some point have become) than by what they are." (Knorr-Cetina 2008, 89)

Merz (1999) argues that event generators—programs used by high energy physicists to simulates particle 'events' in a collider—characterize this unfolding ontology. Event generators are never finalized. They are always "provisional" (particularly for their authors), however users may treat them instrumentally, as black boxes. As Merz notes, in contrast to Rheinberger, "this dynamics has no direction in time. Event generators do not turn from one kind of object into another when time passes; they can be at the same time question-generating devices in one setting, and part of 'the technical repertoire of the experimental arrangement' in another setting" (*ibid*; quoting Rheinberger, 1997).

Before moving on to a detailed discussion of the lab's categories for simulation codes and their relative status as epistemic and technical things, I need clarify the general features of the computational model at work across these categories. While not all the lab's codes were Particle-in-cell (PIC), most were.

4.4 The Particle-in-cell Algorithm

In a 1965 introductions to plasma simulation techniques, Buneman and Dunn write: "The physical laws that govern most of the dynamic behavior of plasmas were known nearly 100 years ago–they are Newton's laws of particle motion and Maxwell's field equations. Yet plasmas are poorly understood media. They have been called capricious and unpredictable. [...] The simple reason why we have failed to understand plasmas is that they contain too many simultaneously interacting particles." The problem of predicting how plasmas behave is a version of the many-body problem: while the interaction between two charged particles is simple to compute, the computational complexity grows exponentially as more particles are added to the system. (Plasma particles interact through electromagnetic forces that span considerable distance). The Particle-in-Cell (PIC) technique was developed as a way to address the challenge of "too many simultaneously interacting particles."

In a PIC simulation, the 'particles' do not directly interact with one another; instead, they are 'pushed' by electromagnetic forces that are only defined on a grid (see figure). After every particle has been pushed, the field values on the grid are recalculated ('solved') and the process repeats. The reason this is a simplification is because the resolution of the electromagnetic field is determined by the grid size, which is typically significantly courser than the resolution of the particle positions. Thus, the intuitive explanation for the term 'particle-in-cell' is that each particle is only affected by forces interpolated from values defined at adjacent grid intersections.

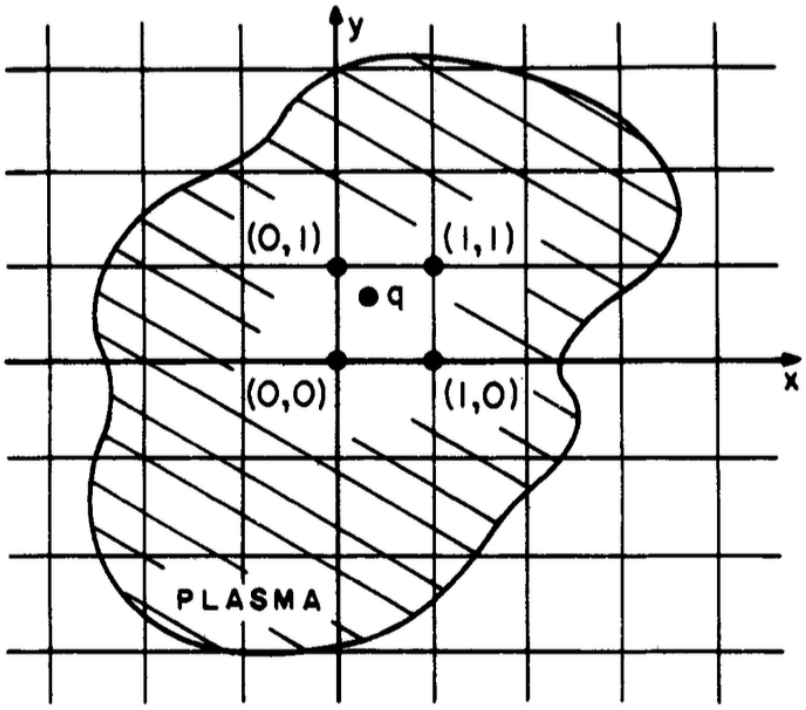

Figure 4.2: Particle-in-Cell Grid

The basic loop of a PIC simulation involves four steps:

- interpolate the electromagnetic forces from the grid to the particle positions
- update the positions of the particles
- calculate current and/or charge densities on the grid

- calculate the electromagnetic forces on the grid

The four phases are typically associated with two code components: the first two are handled by the "pusher" (because the particles are pushed) and the second two by the "solver" (because field values are solved and 'deposited' on the grid).

The PIC algorithm is more accurately understood as a class of algorithms that follow the basic structure described above. Different PIC codes are differentiated by a host of factors, some of which include: the number of dimensions and the spatial geometry (e.g., 2D, 3D, or radial coordinate systems), the subset of Maxwell's equations included in the solver (some codes just use electrostatic forces), and optimizations for specific computational architectures.

4.5 Production Codes

Production PIC codes are those that are actively used in research (they are also simply 'research codes'). When people speak generally of 'simulation codes'—whether in the context of computational physics, philosophy science, or science studies—they are typically talking about production codes. The distinction between production codes and the other categories discussed here is aptly characterized using Latour's definition of the scientific instrument as "any set-up, no matter what its size, nature and cost, that provides a visual display of any sort in a scientific text" (Latour and Woolgar 1986, 68). Unlike reference, framework, and classroom codes, the data produced with production codes is used to generate graphs and visualizations that appear in published research. (There is some overlap with toy codes in this area). Production codes are a privileged category—without question, lab members were primarily concerned with the development and use of production codes. Accordingly, the following discussion is more involved than the discussion of the other categories.

Production codes are the lab's most visible software—they have "brand recognition" as one interviewee put it. The circulation of imagery produced with these codes, in journals and conference talks, is central to supporting the 'brand' of the code and conveying the extent of

the codes use. (Framed journal covers featuring visualizations generated with the lab's codes decorated the physicists' offices). What does a production code's 'brand' consist of and how does it become defined? It is not rooted in the code itself—i.e., in access to the program or its source code. Indeed, many of the lab's most respected codes are not publicly available. Productions codes are named and these names index the people, institutions, research areas, experiments, numerical approaches, and computational techniques associated with them. Over time, the value of a production code can shift as the research areas it is associated with change, as numerical approaches find new applications, as new numerical approaches are integrated into the code, and as new computational techniques become relevant. The authority of a code is in some ways rooted in its pedigree—its relationship to older codes, numerical techniques, and research areas.

Production codes can be conceived as relations of ideas, actors, objects, and practices that evolve over time through extension, recombination, and disassociation (Akera 2007). In the following section I present a historical example illustrating how a particular simulation technique, "the charge-conserving current deposit", emerged a relevant simulation practice and how its relevance changed in response to shifts in high performance computing. This example shows how production codes articulate relationships between research areas, numerical approaches, and computer architectures, and how these configurations can change over time. It also shows how a particular technique can change in significance, moving from a narrow concern associated with a particular type of phenomena to become a more broadly applicable computational approach. In the unfolding of production codes, new technical objects can be stabilized.

4.5.1 An historical example: ISIS and the charge-conserving scheme

The plasma simulation code ISIS was developed in the early 80s by members of the Intense Particle Beam Theory Group at the Los Alamos National Laboratory. It was announced to the community at the International Conference of Numerical Plasma Simulation in 1984 and an abstract describing the program was published the same year in the *Bulletin of the*

American Physical Society (Gisler, Jones, and Snell 1984). The abstract includes the following explanation for the code's novel "charge-conserving scheme": "Because of causality problems in doing Poisson corrections in time-dependent relativistic simulations, we use a charge-conserving scheme to assign particle current density to the finite-difference grid." To explain: Charge conservation (Gauss's law) is a problem for PIC codes[1] and there are different strategies to account for it. One is to periodically correct the field to ensure that charge is conserved (i.e., "Poisson corrections"). Typically, such a correction involves a global and *instantaneous* modification of the field values stored on the grid (discussed above). This strategy doesn't work well when the plasma particles are moving at a significant fraction of the speed of light, which is the case in simulations of particle beams. It's a problem because field 'updates' are not supposed to propagate faster than the speed of light; if they do "causality problems" can result. The approach taken by the developers of ISIS was to make sure Gauss's law (the law of charge conservation) is satisfied at the start of the simulation and then to compute the charge densities in such a way that charge is guaranteed to be conserved as the simulation moves forward (i.e., "charge conserving current deposit"). In such an approach, the field modifications are "local" rather than "global" and causality problem are mitigated. The main disadvantage of the charge-conserving scheme, as Gisler et al. (1989) note, is that "it is cumbersome to code." The developers of ISIS determined that this technique was worth the extra coding effort as it would enable more precise simulation of particle beam interactions. These terms—the charge-conserving scheme, the specific challenges associated with simulating particle beam physics, and the "cumbersome" code associated with the technique—constitute an initial stable relationship that was subsequently modified in significant ways.

ISIS was originally written to study diodes (devices used to produce electron/ion beams), however it was later taken up in a range of research areas. It is important to emphasize, however, that the extension of the code to new types of plasma phenomena entails an interpretive

[1] "Perhaps the simplest method for computing the charge and current densities in a PIC code is to treat the particles as idealized point sources of charge and use linear interpolation [...]. The assignment of charge by this method is continuous as the particle moves through the grid. [...] As the code runs, discrepancies between the charge and the divergence of the electric field could develop" (Marder 1987, 49).

effort. The subsequent work of Gisler et al. (1989) on astrophysical jets included an appendix in which the authors 'introduce' ISIS to the new research community: "Particle-in-cell computer codes are widely used in the fusion community, the the particle beam community, and space physics community for the simulation of collisionless plasmas. Since such codes are not yet in wide use in the astrophysics community a brief account of the workings of ISIS may be in order here" (143). The authors' argument involves an extension of the simulation across many order of magnitude—from diodes to space jets. Their argument is that space jets can be conceptualized as very large diodes, so simulations suited to the former are also suited to the latter.

The second, more important extension involves the shifting role of the underlying charge-conserving scheme. The charge-conserving schemed shifted from a strategy for dealing with relativistic plasmas to a general purpose strategy for computing plasmas on parallel processors. As explained above, the technique was originally meant to address causality problems associated with simulating relativistic plasmas. However, by the early 90s, the charge-conserving scheme had taken on a new role due to the complementarity between its 'local' field solver and the computational requirements associated with parallel processing. As the scientific community looked to parallel and distributed processing to achieve greater computational performance, they also began to evaluate simulation techniques in terms of their ability to take advantage of these new architectures. As Villasenor and Buneman (1992) noted, "[c]omputers of the immediate future demand minimization of data paths and therefore call for local methods. [...] Methods which allow one to update fields purely from local data should therefore receive renewed attention." A key challenge associated with parallel processing is minimizing data exchange between processors. Global field solvers (such as those based on Poisson corrections) are problematic from a parallel processing perspective because each processor needs to send and receive data from a larger number of processors, which can significantly impede performance.

While ISIS's charge-conserving field solver satisfied the requirements of parallel processing, ISIS itself was not a parallel code. It was a serial code (Mori 2002). ISIS continued to be used into the early 2000s—it became known as the "the standard Los Alamos PIC

code" (Carlsten 2002)—however its charge-conserving scheme had a more lasting influence. Throughout the 90s and early 2000s several parallel PIC codes were developed (Buneman, Neubert, and Nishikawa 1992); some of these were directly based on the "cumbersome code" of ISIS's solver. For example, in the development of one new parallel code (PEGASUS), ISIS was "stripped (a major endeavor) but the basic particle push, current deposit, and field solver routines were kept" (Mori 2002, 14).

The story of ISIS's charge-conserving scheme is the story of how a technique initially oriented toward the representation of a particular kind of phenomenon (relativistic particle beam interactions) became a general purpose technique at the center of efforts to scale plasma simulations through parallel processing. Along the way, the meaning of the technique shifted from a narrow, conceptually-specific concern to a technical one. The charge-conserving scheme was detached from the historical continuity of the production code, ISIS. It became a general computational resource used in *other* production codes. In the following section, I turn the question of how the lab's productions codes evolve within the context of individual research projects.

4.5.2 Unfolding Production Codes in Graduate Research

Much of the conceptual development of the lab's production codes was carried out by PhD students. A typical dissertation in the lab involved a sequence of events, roughly: (1) identifying an open research problem (typically with some direction from the lab PI), (2) developing or adapting a numerical approach to address the problem, (3) implementing that approach by extending the functionality of an existing production code (or in rare instances, making a new code from scratch), (4) running the simulations and (5) defending (and hopefully publishing) the results. The activities of coding and research went hand-in-hand in this regard. As one interviewee suggested (perhaps with some exaggeration and over simplification), if the code for a project already exists then the research is not original.

While much of the lab's research entailed code work, not all the lab's code work counted as research. This is especially clear in the discussion of the other code types below. Further,

the day-to-day tasks of testing, debugging, and generally *maintaining* code were not the kind of work that students were *expected* to do as researchers. Nevertheless, some graduate students devoted a significant amount of time to such work. The distinctions within coding practices are expressions of the separation between technical and scientific qualities of the simulation code. In the context of production codes, the issue I want to focus on here is the relationship between physics research and simulation code development— i.e., the process through which simulation codes are moved forward as technical phenomena.

The norms of academic research structured the development process in ways that are important to highlight. For example, code development at national labs follows a different dynamic than academic research code development. As Bruce put it:

> "So, that is how research happens... At a national lab, you tell people: this is what we need, and you need to do it. At university, you, as an advisor, have many different ideas that you think are important—although some may be more important in your own mind. You don't want to force [a particular problem on the student] because sometimes people surprise you on a problem you think maybe is too hard—they figure it out. You never want to tell someone it's too hard to be working on it. Activity starts happening in this regard. Some of it is the science and some of it requires code development."

The production code development that counted as research was development for a recognizable problem. Importantly, students were expected to *use* the code to address a relevant problem. As discussed in the previous chapter, the coupling of development and use was a significant source of pride for the lab, and it was central to maintaining the 'brand' of the code.

General optimizations or the addition of functionality not oriented toward a research problem was not as highly valued. Interviewees made a distinction between the lab's work and other work in computational physics that, as they saw it, was not coupled with use: "[my colleague] doesn't do any physics with his code. He is all about developing it and then trying to put as much in it for users to decide. [...] his role is not to [develop code] and

try to understand physics... that's not what he does. We're trying to do both." The lab's production codes don't simply move forward *as epistemic objects* by making the simulation more accurate (i.e., by getting more physics into the model). "Accuracy" in this context is a relational concept that only has meaning in the context of a particular problem, and the code's use. (It's important to note that the work of maintenance, bug fixing, and the occasional major refactor of the codebase still occurred. These activities were carried out by senior members of the lab whose reputations were significantly attached to the code, not graduate students who actually carried out much of the research and *conceptual* development of the codes).

To illustrate, one of the problems that a student worked on during my fieldwork was the Numerical Cherenkov Instability (NCI). Cherenkov Radiation is a natural phenomenon that is analogous to a sonic boom, but with light. While the speed of light in a vacuum is a universal constant, light moves at different (slower) speeds through different media. For example, light propagates through water at $\frac{3}{4}$ of the speed that it propagates in a vacuum. The speed at which light propagates in a medium is related to the electrical permittivity of the medium—the degree to which it attenuates electromagnetic fields. Like the sonic boom that occurs when an object breaks the sound barrier, Cherenkov radiation occurs when a charged particle moves through a medium faster than light moves through the medium. For example, the high energy particles that escape nuclear reactors pass through the heavy water surrounding the reactor at a speed greater than the phase velocity of light in water; this causes the eerie glow we commonly associate with nuclear reactions. In essence the particle is moving faster than the electromagnetic disturbances it creates, so a kind of shock wave develops behind it, which we experience as a glow. This is 'real' Cherenkov radiation.

The so-called Numerical Cherenkov Instability (NCI) is a *undesirable* simulation behavior that is related to Cherenkov radiation by an analogy. The NCI is a type of numerical instability, an undesirable behavior of the simulated plasma caused by the numerical methods used in the code. Plasma simulationists deal with a range of these unwanted behaviors and they are sometimes given names relating them to the physical phenomenon they suggest (e.g., "numerical heating").

The NCI occurs because very small numerical errors in the simulation cause the permittivity of the simulated plasma to fluctuate in an "unphysical" manner. Usually these fluctuations are small and inconsequential, but when the simulated particles are moving very close to the speed of of light, the particle velocity may become greater than the phase velocity of light in the plasma. Even though the discrepancy may be small, it may be big enough to cause a behavior analogous to Cherenkov radiation. Further, in the right conditions the effect can grow rapidly and 'drown out' the physics of interest.

While the specific details of the student's efforts to address the NCI problem are well beyond the descriptive capacity of this dissertation, there are several features of this work that are important for understanding how plasma simulation development proceeds.

The first of these is the historical nature of the interest in NCI. The NCI was identified as a problem in the 1970s (Godfrey 1975), and it was a relatively narrow area of concern until recently. It became more urgent due to interest in another numerical technique, the Lorentz boosted frame (LBF) (Vay 2007). I will not describe the LBF here beyond saying that it is a technique for reducing the computational cost of certain kinds of simulations (of current interest) by adjusting the frame of reference of the simulation. In LBF, the simulation 'box' moves relative to the 'laboratory frame'. For an important class of problems, LBF significantly improves the speed of the calculation. Unfortunately the technique also exacerbates NCI. In short, recent interest in the LBF has renewed interest in the NCI problem, both of which are motivated by interest in simulating plasma-based accelerators.

Second, the NCI is representative of a general effort in plasma simulation research to analyze and mitigate unwanted numerical effects. As mentioned in chapter two, this work often proceeds by analyzing the 'unphysical' behaviors using the same approaches used to analyze 'real' plasma behaviors. (Hence, the numerical error is named after an actual plasma behavior). One of the student's tasks in addressing the NCI was to derive the numerical dispersion relation of the NCI. A dispersion relation is an analytical description of the relationship between wavelengths and wave frequencies in a particular medium; a *numerical* dispersion relation describes that relationship for the simulated plasma. Though the NCI is an 'unreal' behavior (it is a propagation of numerical error), it can be analytically described

as though it were a real wave. Equipped with this analysis, the student is better prepared to modify the production code to suppress the propagation of the error.

Finally, it is important to consider this works' legibility as physics (its relationship to physical theory) as a condition for it being acknowledged. PhD students in the lab often dismissed the final code work as 'mere implementation'—i.e., the *real* intellectual work was in deriving the equations for the model they were trying to implement. Nevertheless, integrating their new models into the existing production code often involved a significant effort, one which they discounted as insignificant. The matter of "implementation" (and what it actually consists of) is flexible—there is some distance that is being overcome between the software object and the scientific object. That distance might be interpreted in different ways; at times it is ignored in order to bolster the sense that certain kinds of code work are more 'scientific' than others.

4.6 Technical Condition: Reference Codes and Frameworks

Over the last several decades, the codes used by plasma physicists have become larger (in terms of lines of code) due the increasing complexity of both the numerical methods they include and the computing systems they are optimized for. As one senior faculty member in the lab remarked:

> "It used to be, [if] you were a grad student, you'd get a code with 500 lines. You could figure it out in a few days. Like old cars, under the hood, they didn't have a lot of parts thirty years ago."

Over the years, successive cohorts of researchers have contributed to the lab's production codes. In the process some have grown to well over a hundred thousand lines of code. Here we encounter explicitly technical constraint concerning the tension between the necessary technical complexity of the instruments and the capacities of new researchers to build and improve those instruments.

The complexity of cutting edge simulation codes can be a constraint on research ambi-

tions, particularly those of PhDs and postdocs who have only limited coding experience and little patience or interest in learning the intricacies of a established code base. Because so much of the group's research is driven by PhDs and postdocs who are not sufficiently familiar with the production codes to move them forward (i.e., towards a specific research question), the group has put a significant amount of thought into the problem of how to get new grad students 'up to speed.'

Reference codes and frameworks are responses to these concerns. Generally, these codes are 'building blocks' that play a variety of roles: they are used as scaled-down, model codes (models of models, if you will) that students can use to learn how PIC codes work; they are used to benchmark the performance of different kinds of numerical models on different kinds of hardware; and perhaps most importantly they are used as the basis of new production codes. While there are important distinctions between *reference* codes and *frameworks* (which I will discussed below), I group them here for several reasons. First, they are both oriented toward the technical aspects of simulation: the craft of code design, tasks such as benchmarking, and strategies for making code development more efficient. They are responses to explicitly technical concerns. They are also related historically. The lab's frameworks are derived from earlier efforts around reference codes. Accordingly, it is more suitable to think of the relationship between them as a progression—reference codes became frameworks. If the logic by which production codes move forward is, as described above, *phenomenotechnical* (a dialectic of material and conceptual constraints) then the logic of progression at work in reference codes and frameworks is centered more directly on the pragmatic concerns of providing new researchers with the tools they need to start development.

The lab's reference codes and frameworks are largely the project of a single individual, Trevor. Trevor is a senior member of the group and is widely regarded as an expert in PIC simulation techniques and numerical methods in scientific computing. Trevor's role in the lab is undeniably that of a "super technician" (Latour and Woolgar 1986)—his reputation is based more on technical rather than scientific expertise. Trevor's programming style is "old school", almost to a fault. He is skeptical of 'new' software techniques and practices like Object Oriented programming and version control. He retains certain programming

habits from the days when programs were written to punch chards—empty lines and long, descriptive variable names are not found in his code (a waste of punch card space). Over his professional career he has seen several dramatic shifts in high performance computing and the experience of writing and re-writing PIC codes optimized for successive architectures has lead to both a sizable archive of "trusted" code—which he scrupulously debugs and shares freely with member of the lab and collaborators—as well as the authority to make claims about general trends in high high performance computing. These details are important for describing the nature of the 'trust' involved in these codes.

4.6.1 Reference Codes

In software development, a "reference implementation" is a piece of software that satisfies a particular specification and is used to evaluate the effectiveness of other implementations of the specification. For example, standards organizations like the Moving Picture Experts Group (MPEG) produce reference implementations for video encoding standards that software companies can use to test their implementations of MPEG standards. Although Trevor's reference codes were not produced or used to satisfy an explicit software development methodology (for example, production codes are typically tested by comparing them to other similar production codes, not to reference codes), they were treated as 'model' codes in several important senses.

First, reference codes are simulations with minimal functionality. They were sometimes described as 'skeleton' codes, and they only include the most basic elements of the PIC algorithm: particle pusher, field solver, and some basic diagnostic checks. They exclude features that are necessary in production codes (i.e., antennas, cathodes, sophisticated boundary conditions and mesh geometries, diagnostics, etc.). A reference code is a few thousand lines whereas a production code is well over a hundred thousand lines. The relative simplicity of reference codes makes them suitable for researchers who need a tractable starter code on which to build. It also makes them an ideal resource for new simulation developers who want to learn simulation techniques by reading and analyzing the design of a working code.

Trevor's reference codes are better documented than much of the group's other codes—indeed, legibility is just as critical to the functionality of reference codes as executability. Students learned PIC code design principles by reimplementing the reference codes in different languages (e.g., python).

The reference codes were also developed to make them useful as benchmarking tools. As exemplars of high performance scientific programs, PIC codes are useful in settings beyond plasma physics. For examples, a computer scientist designing a new programming language for parallel processing might choose to use a PIC code to test their compiler against a 'real' problem. Hardware makers make similar use of PIC codes. Trevor's reference codes were designed with these kinds of uses in mind.

The reference codes differ from each other along four specific facets: dimensionality (1D, 2D, 3D), the physics they include (i.e., the subset of Maxwell's equations used), the programming language used, and the degree/kind of parallelism used. The latter accounts for the greatest variation between reference codes because different kinds of parallelism can be combined to create codes with multiple layers of parallel processing (up to three layers). Trevor has written individual codes for most of these combinations (dimensionality, physics, programming language, and parallelism) to keep each code as minimal as possible. (A single production code might include all of this variability, but it would be significantly more complex). These four facets are not arbitrary. They are meant to provide a degree of comparability for testing the performance of PIC simulations on different computer architectures. For example, using these codes, Trevor can easily benchmark the performance of 3D electromagnetic codes on computers with radically different hardware configurations. Given a dimensionality (e.g., 2D) and a subset of Maxwell's equations to include (e.g. electromagnetic), all of the codes (regardless of parallelism) should perform the same calculations given the same initial conditions. The important point to emphasize is that the variations in reference codes (dimensionality, physics, programming language, and parallelism) make it possible to determine how well a particular numerical approach performs on a particular computer architecture. This way, the suitability of a particular architecture for a particular type of simulation can be assessed.

4.6.2 Frameworks

Like *reference*, the term *framework* has a fairly specific meaning in the context of software production. One dictionary of computer science defines a framework as:

> "A template for the development of software applications or components. Frameworks provide an outline for the software's structure in the form of objects that not only themselves provide basic functionality but also integrate with each other. Developers build on this to implement their projects' specific functionality by deriving their own objects via inheritance. Frameworks thus consist of at least a comprehensive set of such objects; other facilities, such as programming languages and a runtime environment, may also be provided." (Butterfield, Ngondi, and Kerr 2016)

As this definition suggests, contemporary approaches to framework design are closely associated with object-oriented programming, a design methodology popularized in the 1990s based on the principle of encapsulating related data and procedures in discrete "objects" that are used through well-defined interfaces (ibid.). An Object-oriented framework consists of a taxonomy of object-types, called class a hierarchy, and the interfaces programmers can use to interact with instances of those object-types.

The framework that Trevor developed constituted a conceptual evolution of the reference codes in the sense that it began with an effort to identify the features common to the different kinds of PIC codes. Trevor used these commonalities to construct a collection of general purpose modules—trusted components—that simulationists might use to build new codes. The key technical difference between reference codes and the framework is the degree to which software engineering principles such as encapsulation and 'information hiding' are embraced in the design of the latter (Parnas 1972). For example, The parameters for the functions of the reference codes include many 'low level' options that can make them tedious to use. The framework codes, however, encapsulate these low-level functions.

The shift from reference codes (simple, fully features PIC codes) to frameworks (trusted

components) also coincided with lab's move toward development practices associated with 'open science'—it corresponded to a more concerted effort to orient the lab toward the visions of cyberinfrastructure, as discussed in the previous chapter.

Reference codes and frameworks are oriented to narrow challenges associated with development and high performance computing (e.g., modularity and benchmarking). Their role as technical foundations is to make it easier for physicists to 'get up to speed' with a new project, rather than "reinventing the wheel." Nevertheless, as we have seen, they take distinct approaches in this regard. Reference codes are fully functional simulation codes that are more readily understood than large production codes; they play a pedagogical role for new simulationists studying PIC concepts, and they are the basis of new development work. Frameworks are more explicitly oriented toward new development in the sense that they provide a collection of general purpose components that simulationists can build *with*.

4.7 Classroom Codes

At first blush, classroom codes appear to be quite distinct from the other categories. Consideration of their role as instructional aids as well as the technical conditions of their development and use separates them from production codes and reference/framework codes. This distinctiveness begins to break down under scrutiny, however. As we will see, classroom codes are central to the perceptual training of simulationists; further, because they embrace experiential modes that cannot be realized in production codes (e.g., real time interactivity), classroom codes are somewhat aspirational. They represents a certain ideal of the kind of interaction that researchers *might* have with production codes, but cannot.

To the extent that the pedagogical function of classroom codes is rooted in their capacity to provide unambiguous experiences of plasma phenomena, classroom codes appear to be epistemologically uninteresting. "Epistemically interesting relations between concepts and objects do not take the form of ostension" (Rheinberger 2005a, 407). An ostensive character is apparent in the fact that classroom codes are often tailored around 'textbook' phenomena: magnetic field structures like an infinite wire and the Helmholz coil, as well as classic plasma

behaviors like landau damping and the two stream instability were among the concepts presented with classroom codes. On the other hand, as we saw at the beginning of the chapter, one of the benefits that the lab PI associated with classroom codes was the ability to push theoretical parameters to the point that the representational transparency of the simulation begins to break down. In this way, students can explore limits of the underlying model. Such an engagement, we might imagine, would afford a more intuitive grasp of plasma phenomena.

This idea of 'breakage' can be interrogated a bit further. What precisely is it that breaks in this scenario, and how is that breakage supposed to be perceived and made productive? It is *not* the physical theory that breaks (i.e., Maxwell's equations) because simulations are not isomorphic with the theories they are based on. (To be clear, the lab PI was not naive on this front). What breaks is clearly the *computational* model—the specific implementation of the physical theory—which is strictly speaking, a concern of simulationists and an approximate concern for physicists more generally. Furthermore, the path toward recognizing such a breakage and making it productive hinges on the ability to distinguish between the proper functioning of the computational model, a failure in the computational model, and any technical problems or bugs that that might exist in the software. Such distinctions are not trivial; in fact, they are among the central challenges of scientific computing. We can interpret the PI's endorsement of this mode of physics education as a covert endorsement of perceptual training of simulationists. The kind of intuition this experience produces is quite specific to the experience of making and using simulations.

A related argument concerning a more general kind of perceptual training is made in the education literature by authors who promote simulation as a pedagogical tool:

> "Many physicists find it quite mysterious and somewhat disturbing that carefully developed simulations are more educationally effective than real hardware. [...] A real-life demonstration or lab includes enormous amounts of peripheral information that the expert instructor filters out without even thinking about it. [...] A carefully designed computer simulation can maintain connections to real

> life but make the student's perception of what is happening match those of experts. This is done by enhancing certain features, hiding others, adjusting time scales, and so on, until the desired student perception is achieved." (Wieman and Perkins 2005)

There are several notable features in this argument. First, the process of pruning "peripheral information" tracks quite closely with representational practices in science as a whole. 'Filtering out' and 'enhancing' are not done just for the sake of students; they are central representational strategies in research as well (Vertesi 2014). This argument also calls attention to a notional "desired student perception" and the experiential differences between classroom codes and non-classroom codes. While the analysis of visualization techniques involved in plasma simulation was not a focus of my research, there are very apparent distinctions between the degrees of interactivity between them.

The temporality of the code's execution is the key distinction between classroom codes and non-classroom codes. Production codes are typically run as 'batch' processes that can take several minutes, hours, or longer to complete, even on high performance computers. This mode of operation deviates from the now-established norm of interactive computing, in which command and response are virtually instantaneous occurrences. As Chun (2011) has argued, interactive computing is strongly linked to the feelings empowerment and control associated with computing. Chun describes the sequence of call and instantaneous response as a source of "causal pleasure" (Chun, 2011, 42). Whereas production codes, the purview of experienced simulationists, often trade causal pleasure for more detailed simulations, classroom codes typically do not involve such a sacrifice. The "desired student response" is not one of delayed apprehension; it is of intuition derived from real-time manipulation. The goal is to get a feel of the computational model. The developer's sensitivity to these concerns is demonstrated, for example, in the the subtle 'frames per second' graph which appears in the top-left corner of Lars' program. Real-time interaction was, nevertheless an aspiration of some lab members. For example, one interviewee described enjoying simple 1D simulations because he could run many of them very quickly.

The pedagogical orientation of classroom codes introduces specific technical requirements that are not satisfied by production codes. Classroom codes not only require user-friendly graphical interfaces, they also need to be accessible and easy to run on general purpose personal computers. The context of execution is significantly different for classroom codes than for the other simulation codes. For example, most of the production and reference/framework codes are written in Fortran and distributed as source code, which needs to be compiled for the workstation or HPC system it will run on (i.e., the lab is not in the habit of packaging and distributing binaries built for Windows, MacOS, Linux etc.). As we saw in the case of Lars' simulation, at the beginning of the chapter, web browsers are an ideal context for running classroom codes, however modern web browsers only run programs written in Javascript, which is not a language physicists would typically work with.

4.8 Toy Codes

Like reference codes, toy codes are a somewhat looser grouping than production codes and classroom codes. When the physicists used the term, they were typically referring to relatively small pieces of code, quickly written over a short period of time for a very particular purpose. Although she doesn't use the term, one graduate student nicely conveys the scene of a hypothetical toy code's development:

> There are different dimensions of the software that you do when you're doing a research problem. For example, if you're trying to calculate something, or follow some simple phenomenon, and it is described theoretically by some equation that is very very complex, you start to think: 'ok I'm going to do some code that will try to do some approximation and allow me to see a little better how [...] this phenomenon develops.' But if you're doing something that has actually a simple analytical form, you can do in one afternoon a very small code just to see 'oh yeah the particle goes and turns and comes back' and everything. And that code—you're the only person in the world who's going to care about it. It's going to be like a hundred lines and you do it in—not an afternoon, because for

> sure you have plans and everything—but you do it, you look at it, maybe your professor looks at it. But no one really—it's not 'software'... It *is* something that you developed—it is software because it's a small code, but it's dimension is quite narrow".

It's worth exploring the student's initial description of toy codes as 'not software'. Software, as a 'ware,' is typically attributed with some broadly recognizable utility, which toy codes do not seem to posses. Toy codes have a narrow scope; as the creator of a toy code, "you're the only person in the world who's going to care about it." This narrow scope also suggests that toy codes are ephemeral and perhaps disposable, even to their creators. As commented on above, toy codes are not named, and this lack of naming reflects their abbreviated utility. A toy code is not a material continuity that is frequently returned to, that is itself "developed" over time. A practice of material engagement does not form around it, and certainly not a community of practice.

All of these details are important because they add up to the fact that toy codes are far from 'black boxes.' As the student points out, the point of toy code is not merely their output but a certain experience of "seeing a little better" some phenomenon that is otherwise quite opaque to experience (in its theoretical or analytical form). Furthermore, the experience is not intended *for* just anyone—it is entirely inconsequential to most everyone—it's for the creator of the code herself. Toy codes can, accordingly, be thought of as codes that are by and for an individual. They are not black boxes in the sense that the developer is not likely to treat the experience derived from the code as 'fact'. The experience attained, the author of the code moves on "because for sure [she] has plans and everything."

However, not all the interviewees characterized toy codes in these same terms. One postdoc described toy codes as having varying degrees of complexity, and in some case their outputs are used directly in publications. The postdoc referred to a code he had written with a group of physicists at a national lab over the course of the summer. The code was used to study inertial confinement fusion; in the resulting publication, the code is not named. His reason for referring to the project as a toy code was simply to differentiate the project from

a production code: "That code was developed in order to circumvent some approximations that they make in a much bigger, bulkier, but feature-rich code called [X]. It was a toy-code in the sense that it was a significantly simplified version of [X]." In this scenario, the development of a toy code constitutes an alternative to the difficult prospect of modifying an existing, large production code. The simplicity of toy codes is an important part of their appeal.

The development and use of toy codes was described as an important first experience in a student's introduction to plasma simulation: the same postdoc fondly recalled one of his first code development experiences, which involved building a small program. "It was actually very interesting. It was almost something similar to what I do now, which is very cool, but it was just a single charged particle going through an electromagnetic field, and I was asked to kind of code it up, and it's really simple–it was just like 200 lines, but as a sophomore it was complicated enough that I felt rewarded [because] I was getting payed for it and my advisor thought it was good work and [...] it may be used toward something that was actually publishable."

Toy codes aren't an easy category to describe in terms of their dynamics of evolution since their very status as 'software' is somewhat marginal. In the case of a toy code that is a "significantly simplified version" of a research code, we can see how toy codes participate in the dynamics of evaluation, testing, and rectification of production codes. The simplicity and scale of toy codes—that they are the kind of thing you might make in an afternoon without ruining your evening—is a relational quality. The kinds of complexity and functionality that might be included within the category of a "toy" change with the programming environments and tools that simulations have at their disposal.

4.9 Discussion

These categories might be organized and described along a number of different axes—by the size of the associated codes, their ages, or the kinds of communities associated with them. The key concern in this discussion is how they relate to the terms *epistemic thing* and

technical object.

Production codes are often treated as technical objects in experimental work where the code is used to test certain experimental parameters (or a space of parameters). As discussed in the previous chapter, the instrumental use of codes can lead to disputes between users and developers because experimenters want codes that 'just work' while developers understand their codes in a more provisional way (i.e., as epistemic things). In the preceding discussion, we encountered specific examples of the unfolding of production codes as epistemic things in the example of the history of the charge-conserving current deposit scheme and the NCI. In the latter case, the 'problem' of NCI (i.e., NCI as an object of study) evolved in time as areas of research like plasma-based acceleration became more important and as new numerical methods renewed attention to problems once thought to be fairly settled (or uninteresting). PULSE—and other production codes—are sites within which these larger shifts are carried out. In the process, they become "very, very big and quite old," as one interviewee put it.

The example of the charge-conserving technique and its 'local solver' is interesting because it shows how, in the 'unfolding' of production codes, the significance of numerical techniques can shift from a narrow concern in a particular research area (diodes) to a more general purpose strategy (a way to do plasma simulations on computer clusters with parallel processing). In this example we can see how the epistemic things in plasma simulation can change their character and become more widely relevant as technical conditions. Production codes appear to correspond to Merz's account of event generators quite closely—they are both epistemic things and technical conditions. However, objects that are more clearly 'technical' can emerge and detach themselves from this process of unfolding (in a sense they can move between production codes).

Reference codes, frameworks, and classroom codes are more strongly associated with technical conditions. Though they remain problematic and subject to revision (as the demonstration of Lars' program suggests), the broad intention behind these categories is to define a separate context of development work, one safely removed from the meanderings of the research process and the weighty concerns of a production code's "brand recognition." Within such spaces, the stability associated with technical conditions is potentiality achievable. Ref-

erence codes appear to be *particularly* technical objects. As stripped-down, simplified PIC codes, their utility is narrowly defined and their operational complexity is bracketed. They are models of models.

The categories constitute a system of commonplaces—positions of development activity that are narrowly oriented toward particular intentions and relationships to different communities (i.e., students, fellow plasma simulationists, and non-physicists). Understood as a tactic for managing relationships with other communities, the categorization is related to conceptions from science studies that describe how different communities cooperate around shared objects. Instead of boundary objects—"objects which are both plastic enough to adapt to local needs and constraints [...], yet robust enough to maintain a common identity across sites" (Star and Griesemer 1989)—we encounter a system of object *types* associated with different activities and community relationships. These are objects *with* boundaries. Importantly, the simulationists are able to selectively draw aspects of objects in one category and apply them to those in another (i.e., to stabilize those aspects as technical conditions).

These categories reflect different kinds of memory practices, as described in chapter two. Memory practices concern the production of a useful past, and each of the code types discussed here constitute different kind of past. A production code follows the trajectory of a research area—the input deck used to configure the code takes on the status of a kind of lab record. On the other hand, frameworks suggest a quite different kind of memory, one guided by common standards and conventions in the *development* of new codes (i.e., the components that codes commonly have). Likewise reference codes—particularly in their capacity as programs used to instruct simulationists in design principles—are oriented toward the production of a record that facilitates software development and other technical design practices. Where production codes suggest continuity in the *use* of particular objects, reference codes and frameworks suggest continuity in the forms of expertise required to *make* those objects.

4.10 Conclusion

This chapter demonstrates that we can learn a lot about the interactions between scientific objects and technical conditions by recognizing the diversity of simulation practices in local settings. While production codes—the charismatic megafauna of computational science—receive much of the attention from scholars in science studies and philosophy of science, the mechanisms of the lab's simulation ecology only become apparent through consideration of more modest species: classroom codes, reference codes, frameworks, and toy codes. Most obviously, we learn that the lab's simulation codes appear in many surprising places—in student homework and in the technical design processes of non-physicists, for example.

The question at the center of the chapter was whether the lab's simulation codes are simultaneously epistemic (objects characterized by absence and ongoing material definition) and technical (stable conditions of scientific activity), or whether they *move* between these poles, particularly from the epistemic toward the technical. These positions were associated with Knorr-Cetina (2008) and Rheinberger (1997) respectively. While production codes are aptly characterized as both epistemic things and technical objects—they are "multiplex and unfolding" (Merz 1999)—the other categories of code represent (at the very least) an explicit effort to constitute stable technical conditions for plasma simulation. This effort is significantly, though not exclusively, motivated by pressure from funding agencies. These categories are therefore closely related to the 'vision' of software development discussed in the preceding chapter.

Reference codes, frameworks, classroom codes, and toy codes express an intention to carve out spaces of development activity—commonplaces—that are separate from the ponderous concerns (e.g., credibility) of production codes. In particular, reference codes (small codes used for benchmarking) represent the best evidence that stability in the conditions of plasma simulation can be achieved. On the other hand, toy codes, which barely rise to the level of 'software' by some accounts, correspond to spaces of personal meaning making—here we encounter the poetic center of simulation as a practice. What we witness in these categories, then, is the "poetics and pragmatics" (Shankar 2007) of plasma simulation—the way in

which simulationists constitute their work in a manner that is "wholly personal and yet intrinsically professional." This is a theme that I explore further in the final chapter.

4.11 Bibliography

Akera, Atsushi. 2007. "Constructing a Representation for an Ecology of Knowledge Methodological Advances in the Integration of Knowledge and Its Various Contexts." *Social Studies of Science* 37 (3): 413–41. doi:10.1177/0306312706070742.

Buneman, O., T. Neubert, and K. I. Nishikawa. 1992. "Solar Wind-Magnetosphere Interaction as Simulated by a 3-D EM Particle Code." *IEEE Transactions on Plasma Science* 20 (6): 810–16. doi:10.1109/27.199533.

Butterfield, A, Gerard Ekembe Ngondi, and Anne Kerr. 2016. *A Dictionary of Computer Science*.

Carlsten, Bruce E. 2002. "Small-Signal Analysis and Particle-in-Cell Simulations of Planar Dielectric Cherenkov Masers for Use as High-Frequency, Moderate-Power Broadband Amplifiers." *Physics of Plasmas* 9 (5): 1790–1800. doi:10.1063/1.1467931.

Chun, W.H.K. 2011. *Programmed Visions: Software and Memory*. MIT Press.

Gisler, G., M. E. Jones, and C. M. Snell. 1984. "ISIS: A New Code for PIC Plasma Simulation." *Bull. Am. Phys. Soc* 29: 1208.

Gisler, Galen, Richard VE Lovelace, and Michael L. Norman. 1989. "Electrodynamic Formation of Astrophysical Jets-Simulations." *The Astrophysical Journal* 342: 135–45.

Godfrey, Brendan B. 1975. "Canonical Momenta and Numerical Instabilities in Particle Codes." *Journal of Computational Physics* 19 (1): 58–76.

Hacking, Ian. 1983. *Representing and Intervening: Introductory Topics in the Philosophy of Natural Science*. Cambridge University Press.

Knorr Cetina, Karin. 2001. "Objectual Practice." In *The Practice Turn in Contemporary Theory*, edited by Theodore R. Schatzki, Karin Knorr Cetina, and Eike von Savigny. Routledge.

Knorr-Cetina, Karin. 1997. "Sociality with Objects Social Relations in Postsocial Knowledge

Societies." *Theory, Culture & Society* 14 (4): 1–30. doi:10.1177/026327697014004001.

———. 2008. "Objectual Practice." In *Knowledge as Social Order: Rethinking the Sociology of Barry Barnes*. Vol. 83.

Knuuttila, Tarja. 2011. "Modelling and Representing: An Artefactual Approach to Model-Based Representation." *Studies in History and Philosophy of Science Part A*, Model-based representation in scientific practice, 42 (2): 262–71. doi:10.1016/j.shpsa.2010.11.034.

Knuuttila, Tarja, Martina Merz, and Erika Mattila. 2006. "Computer Models and Simulations in Scientific Practice." *Science Studies* 19 (1): 3–11.

Latour, Bruno, and Steve Woolgar. 1986. *Laboratory Life: The Construction of Scientific Facts*. Princeton University Press.

Marder, Barry. 1987. "A Method for Incorporating Gauss' Law into Electromagnetic PIC Codes." *Journal of Computational Physics* 68 (1): 48–55.

Merz, Martina. 1999. "Multiplex and Unfolding: Computer Simulation in Particle Physics." *Science in Context* 12 (02): 293–316. doi:10.1017/S0269889700003434.

Mori, Warren B. 2002. "Recent Advances and Some Results in Plasma-Based Accelerator Modeling." *AIP Conference Proceedings* 647 (1): 11–28. doi:10.1063/1.1524854.

Parnas, D. L. 1972. "On the Criteria to Be Used in Decomposing Systems into Modules." *Commun. ACM* 15 (12): 1053–8. doi:10.1145/361598.361623.

Rheinberger, Hans-Jörg. 1997. *Toward a History of Epistemic Things: Synthesizing Proteins in the Test Tube*. Stanford, Calif.: Stanford University Press.

———. 2005a. "A Reply to David Bloor: 'Toward a Sociology of Epistemic Things'." *Perspectives on Science* 13 (3): 406–10.

———. 2005b. "Gaston Bachelard and the Notion of 'Phenomenotechnique'." *Perspectives on Science* 13 (3): 313–28.

Shankar, Kalpana. 2007. "Order from Chaos: The Poetics and Pragmatics of Scientific

Recordkeeping." *Journal of the American Society for Information Science and Technology* 58 (10): 1457–66. doi:10.1002/asi.20625.

Spencer, Matt. 2015. "Brittleness and Bureaucracy: Software as a Material for Science." *Perspectives on Science* 23 (4): 466–84. doi:10.1162/POSC_a_00184.

Star, Susan Leigh, and James R. Griesemer. 1989. "Institutional Ecology, 'Translations' and Boundary Objects: Amateurs and Professionals in Berkeley's Museum of Vertebrate Zoology, 1907-39." *Social Studies of Science* 19 (3): 387–420. doi:10.1177/030631289019003001.

Sundberg, Mikaela. 2009. "The Everyday World of Simulation Modeling the Development of Parameterizations in Meteorology." *Science, Technology & Human Values* 34 (2): 162–81.

Vay, J.-L. 2007. "Noninvariance of Space- and Time-Scale Ranges Under a Lorentz Transformation and the Implications for the Study of Relativistic Interactions." *Physical Review Letters* 98 (13): 130405. doi:10.1103/PhysRevLett.98.130405.

Vertesi, Janet. 2014. "Drawing as: Distinctions and Disambiguation in Digital Images of Mars." In *Representation in Scientific Practice Revisited*, edited by Catelijne Coopmans, Janet Vertesi, Michael E. Lynch, and Steve Woolgar. MIT Press.

Villasenor, John, and Oscar Buneman. 1992. "Rigorous Charge Conservation for Local Electromagnetic Field Solvers." *Computer Physics Communications* 69 (2-3): 306–16.

Wieman, Carl, and Katherine Perkins. 2005. "Transforming Physics Education." *Physics Today* 58 (11): 36–41.

Wimsatt, William C. 2007. *Re-Engineering Philosophy for Limited Beings: Piecewise Approximations to Reality*. Harvard University Press.

CHAPTER 5

Conclusion

5.1 Poetics and Pragmatics

Shankar (2007) characterizes the "poetics and pragmatics" of laboratory scientists' personal recordkeeping practices this way: "Documents [. . .] must be useful and 'accountable' to the records creator, but also must be accessible to the larger community of practice. [. . .] [T]he record evolves through the accretion of layers of meaning and personal-knowledge management"; the result is "a document that is paradoxically wholly personal and yet intrinsically professional" (1463).

In this dissertation I have sketched some of the features of the "poetics and pragmatics" of plasma simulation. I have described how lab members navigated the twin demands of individual meaning and adherence to increasingly prominent standards for scientific software development. In chapter two's discussion of memory practices, I characterized the coherence of the lab's simulation development as an ongoing activity of elaborating and explicating the unexpected. The ways of making the unexpected events of plasma codes meaningful were shown to be highly localized and inherently historical. However, in chapter three, the discussion of the lab's encounter with cyberinfrastructure and open science illustrated how changing norms of scientific software challenged this local poiesis. Changing norms around scientific software pressured the simulationists to constitute their codes as sharable technical conditions (e.g., cyberinfrastructure). The negotiation of poetics and pragmatics in plasma simulation manifests itself in the lab's differentiation of codes (open and closed; production code and framework) with different relationships to the broader community, as discussed in the previous chapter. These are not boundary objects, they are objects with boundaries.

The properties of these objects can be selectively transferred from one context to another as simulationists see fit.

5.1.1 Implications for Open Science & Software Curation

This dissertation complicates the vision of open science by arguing that code sharing and greater 'openness' are not universally beneficial to scientific work. As shown in chapter three, plasma simulationists were ambivalent to openness. On the one hand, they welcomed policy shifts associated with open science that have brought greater attention to software as a form of scholarly production. Funding agencies increasingly recognize software development as a *fundable* activity (which was not the case only a few decades ago), and simulationists eagerly took advantage of these funding opportunities to develop new codes—codes oriented toward both research and non-research goals—and to "modernize" existing codes. Some codes were released with open source licenses and more focus was given to the development of codes that the broader community might benefit from, such as frameworks ("trusted components") and classroom codes. On the other hand, the expectation of funding agencies that scientific software should be used by an extended community of users constituted a significant shift in the simulationists' relationship to their codes.

In particular, the case of PULSE demonstrates that it can be difficult to 'open' mature codes that originate in 'closed' communities without significantly affecting the social arrangements that ground trust in simulation results. PULSE's developers chose not to publicly release the code because of concerns that casual users are not sufficiently motivated to invest the time (and sometimes the development effort) required to "get good results." The expertise needed to effectively use PULSE is stubbornly localized. Here we see a conflict between the common instrumental understanding of software (software as a technical condition) and the provisionality that characterizes simulationists' relationship to software (software as epistemic object). Lab members recognized that it was often necessary to modify a code in order to get good results—these modifications were not seen as indications of a fault in the code; they were regarded as an inherent feature of computational physics. So

long as the community of PULSE users was limited to a small group of trusted colleagues, the code could evolve, accrue credibility (i.e., "brand recognition") and *also* remain provisional in this way. However, in the regime of open science—where the expectation is that software will be taken up as a shared technical condition—provisionality and maturity are contradictory qualities. A mature software tool, in this context, is one that has become routinized and stabilized. (The reference codes, discussed in chapter four, exemplify this combination of qualities). By keeping PULSE closed, developers worked to retain the social arrangements within which the code has emerged as a credible simulation code.

In keeping with the observation that "solutions to the problem of knowledge are solutions to the problem of social order" (Shapin and Schaffer 1985, 332), the historically 'closed' social order of plasma simulation is a very different kind of "solution" than that proposed by open science—nevertheless, the former has clearly yielded results, as the long history of plasma simulation demonstrates. My argument, then, is not that open science policies are misguided. Rather, I mean that a move toward greater openness sometimes entails a difficult transformation in the social order that grounds scientific work. Opening a code is not an universally beneficial action.

This account of plasma simulation has implications for information studies, particularly for understanding the unique challenges associated with the stewardship and curation of scientific software (Lynch 2014). As described in the introduction, curation is 'care work' (Karasti, Baker, and Halkola 2006); it involves the collection, description, preservation, and ongoing maintenance of digital objects with the aim of supporting their value. In keeping with the visions of software development offered by cyberinfrastructure and open science, this "value" is commonly understood as a capacity for reuse by an extended community (Mayernik et al. 2013). Current approaches to software curation have something of a blind spot when it comes to forms of value that *do not* fit this vision. Again, the case of PULSE demonstrates that plasma simulationists do not always identify the value of their codes with accessibility and reuse—in fact, they sometimes see accessibility and reuse as *threats* to that value. How should programs like PULSE be curated, then? This discussion has suggested that a possible answer is to attend to the negotiation of poetics and pragmatics within a

given research setting. Curators should be mindful of alternative forms of epistemic value and adjust the work of stewardship accordingly. Promoting reuse is not the only way to care for code.

5.2 Conspicuousness Revisited

A central theme of this dissertation has been the 'movement' of science and the manner in which a field like plasma simulation constitutes itself as an ongoing activity. The experience of the unexpected (i.e., in the behavior of the simulated plasma) is a central form of continuity in plasma simulation, as it is in experimentation (Rheinberger 1997). The unexpected has the quality of a *recurrence*: it is always being (re)defined in relation to what it was (not) before. Simulations' past is continuously and permanently created as a horizon of expectations subsequently found to be misguided. It is "a process that permanently transforms the truth of today into the error of yesterday" (Rheinberger 2005, 319).

While this movement is not teleological, there is a certain directional structure that follows from the tendency of scientific objects to become new technical conditions. This touches on the debate between Rheinberger and Knorr-Cetina over the degree of separation between the objects of science and the conditions of science. Knorr-Cetina argued that contemporary scientific instruments are also epistemic things. However, as argued in the previous chapter, while simulation codes remain problematic and conspicuous, we can still identify efforts to stabilize and share common technical conditions in the case of the lab's frameworks, reference codes, and classroom codes. On the other hand, it's notable that these kinds of code are not directly enrolled in research (e.g., they are not production codes). Rather, they have the quality of infrastructural components.

The central question that launched this project was whether there are *different kinds* of conspicuousness occurring and interacting within scientific software development. The question followed from Knorr-Cetina's (2008) argument that scientific instruments (e.g., plasma simulation codes) are also epistemic objects (e.g., plasma)—they have the same "unfolding" ontology; they are both non-self-identical. If that is so, then the two trajectories

of research software development noted in the introduction—the transformations of the code and the transformations in our understanding of the scientific object—might be understood as independent but interacting "chains of wanting" (*ibid*).

The observations of chapters two support and clarify this portrayal of scientific software development. I noted that the experience of the unexpected (a form of conspicuousness) occurs within a representational space. In fact, representational spaces are conditions of possibility for the unexpected in the sense that an expectation entails an iteration and repetition (Rheinberger 1997, 105). (The sense of representation here is more one of re-presentation). This holds for both codes and plasmas. Further, the representational systems within which the conspicuousness of codes and plasmas occur (i.e., in which codes and plasmas come into being *as* codes and plasma) are historically specific. The field of software engineering, for example, helped to constitute "software" as a historically specific form of writing and representational practice (Frabetti 2015). Similarly, when simulationists relate simulated plasma to "real" plasma, they are making a comparison to representations derived analytically (i.e. theoretically) or experimentally—both should be understood as inherently historical technical conditions of the 'scientific real'. It follows that conspicuousness is rooted in the experience of working with materially constituted and historically specific representational systems. The *kinds* of conspicuousness emerge from the practice of representation itself.

5.3 Bibliography

Frabetti, Federica. 2015. *Software Theory: A Cultural and Philosophical Study*. Rowman & Littlefield International.

Karasti, Helena, Karen S. Baker, and Eija Halkola. 2006. "Enriching the Notion of Data Curation in E-Science: Data Managing and Information Infrastructuring in the Long Term Ecological Research (LTER) Network." *Computer Supported Cooperative Work (CSCW)* 15 (4): 321–58. doi:10.1007/s10606-006-9023-2.

Knorr-Cetina, Karin. 2008. "Objectual Practice." In *Knowledge as Social Order: Rethinking the Sociology of Barry Barnes*. Vol. 83.

Lynch, Clifford. 2014. "The Next Generation of Challenges in the Curation of Scholarly Data." In *Research Data Management: Practical Strategies for Information Professionals*, edited by Joyce M. Ray, 395–408. West Lafayette, Indiana: Purdue University Press.

Mayernik, Matthew, Tim DiLauro, Ruth Duerr, Elliot Metsger, Anne Thessen, and G. Choudhury. 2013. "Data Conservancy Provenance, Context, and Lineage Services: Key Components for Data Preservation and Curation." *Data Science Journal* 12 (0). doi:10.2481/dsj.12-039.

Rheinberger, Hans-Jörg. 1997. *Toward a History of Epistemic Things: Synthesizing Proteins in the Test Tube*. Stanford, Calif.: Stanford University Press.

———. 2005. "Gaston Bachelard and the Notion of 'Phenomenotechnique'." *Perspectives on Science* 13 (3): 313–28.

Shankar, Kalpana. 2007. "Order from Chaos: The Poetics and Pragmatics of Scientific Recordkeeping." *Journal of the American Society for Information Science and Technology* 58 (10): 1457–66. doi:10.1002/asi.20625.

Shapin, Steven, and Simon Schaffer. 1985. *Leviathan and the Air-Pump*. Princeton University Press Princeton, NJ.