Code in the Hands of Non-Computer Scientists: An Investigation of Coding Practices in Scientific Research at the University of Utah

Nathaniel Lanza
University of Utah

UUCS-23-004

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

5 December 2023

Abstract

Prior research has shown that non-CS scientists often rely on code to generate research results, so coding errors have the potential to cascade into inaccurate scientific findings. Papers have been retracted due to coding errors, demonstrating the critical need for increased understanding in this area. As coding becomes an increasingly essential tool across various scientific disciplines, further understanding of the differences in use between computer scientists and scientists in other disciplines (non-CS scientists) is necessary to understand what support and education non-CS scientists need. This paper aims to identify common practices in non-CS fields, assess potential barriers to effective coding, and lay the groundwork for improving code use in non-CS research at the University of Utah (the U). It investigates the programming knowledge and software verification capabilities of non-CS scientists through (1) analysis of code submitted by eight non-CS researchers and (2) interviews with three of these scientists. 50 examples of code antipatterns were identified across analyzed code, indicating issues with documentation, readability, and proper use of flow control structures. Summaries of analyzed files showed a wide range of coding styles, but indicated that some scientists frequently employ practices which increase the likelihood of errors in their results. Interviewees report having little applicable formal CS training and that the U provides no incentives to improve programming skills. Additionally, these interviewees lack testing techniques for their particular software applications. This paper recommends increased institutional incentives for improving programming skills, cross-disciplinary collaboration in CS and science education, and research into novel software testing methods for non-CS fields. Errors in non-CS software threaten the validity of research results, and changes at the institutional level in training, incentives, and collaboration are crucial to address this problematic gap in scientific rigor.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The integration of coding into various scientific disciplines has transformed the way researchers approach problems, analyze data, and present their findings. As a result, non-CS scientists are increasingly relying on code to facilitate and guide their research findings. Despite this growing reliance, there is limited understanding of the coding practices, challenges, and motivations among non-CS scientists, which could affect the quality and rigor of scientific results, especially since many non-CS scientists lack formal education in software development (Hatton, 1997; Hannay et al., 2009). This thesis aims to further our understanding of current coding practices in non-CS research at the University of Utah. Additionally, this work provides a baseline to inform two areas of future research: how CS educators can better prepare their non-CS students for the code they have to write in academia, and what software verification techniques work in scientific domains where traditional unit testing isn’t feasible.

CHAPTER 2

BACKGROUND

Scientists from various research domains rely on code to analyze their data and deduce publishable findings. As opposed to computer scientists, who study computing for its own sake, non-CS scientists use code as a tool to study another topic. This thesis conceptualizes non-CS scientists as “purpose-first programmers,” defined as conversational programmers who care about code’s purpose, but don’t attempt to understand the low-level details of precisely how it executes (Cunningham, 2020, p. 348). In other words, non-CS scientists use code out of necessity, because it provides the only efficient means to collect, analyze, and model the large datasets that they base their results on. However, this lack of emphasis on the details of coding itself may lead to poor quality code in these disciplines, and the consequences of bad code can be severe, including instances in which widely-cited papers were retracted (Hatton, 1997; Trisovic et al., 2021; Miller, 2006). Given the consequences of poor code in research, we seek to understand more about how scientists’ code writing skills affect their research and what might be done to mitigate the potential for incorrect research results.

Non-CS Scientists Learn to Code in Informal Contexts

A survey showed that most scientists learn to code from self-study and informal education, with only around ⅓ of respondents ranking formal education as important or very important (4 or 5 on a 5-point scale) (Hannay et al., 2009). As a result of the emphasis on self-study, these scientists seem unfamiliar or unconcerned with common concepts taught in formal software development courses. In the same survey, only 44.1% ranked software design as important or very important and an even lower 22.9% for software project management. Most concerningly, while 60.2% of scientists ranked software testing as important or very important, only 46.6% believe that they possess a good or expert understanding of how to test software. Additionally, only 42.4% believe that they possess a good or expert understanding of how to verify the correctness of software (Hannay et al., 2009). These findings underline a potential problem with the coding practices of non-CS scientists–if more than half don't have a good understanding of how to test and verify software, they may not be able to write code that produces a correct output in all situations.

Bad Code Has Consequences

In practice, it's possible for coding mistakes to slip into and compromise scientific results. A 1997 study into scientific coding found that scientific software is haphazard and prone to fault (Hatton, 1997). By running static analyzers on implemented code from a variety of non-CS fields, Hatton found a variety of detectable faults, such as interface inconsistencies and unconditional dependence on uninitialized variables (Hatton, 1997). These were weighted by Hatton on a zero to one scale according to the probability of causing a bug over the lifetime of the software, and weighted static faults per 1000 lines (a KLOC is 1000 lines) were quantified for each scientific field. To arrive at this number, Hatton added up the weights of every fault in each software package, then divided by KLOC in that package. The resulting fault rates differ wildly: some are less that one, indicating that the sum of weights of static faults in that package is less than the KLOC in that package, while others reach as high as 140. While substantial variability exists between different software packages, these results show that non-CS scientists often write software using poor practices–leaving the software, and therefore the research as a whole, vulnerable to serious errors. To examine whether the measured static fault rates corresponded with real errors, Hatton took a deep dive into one field, earth science, to investigate the accuracy of results produced by a variety of software packages in this field (1997). Each package has the same goal: to apply a series of mathematical functions to seismic data in order to remove noise. However, due to floating-point inaccuracies that were unknown to the non-CS scientists, the packages produced data with a spread of 100% at the final output, in addition to some of the packages having off-by-one array indexing errors, which shifted the data incorrectly (Hatton, 1997).

The situation has not improved much in the intervening decades since Hatton's study; a recent study attempted to re-run a sample of over 9000 R files used in scientific research and found that 74% crashed on initial execution, while only 56% crashed after automatic code cleaning tools were used to eliminate common errors (Trisovic et al., 2021). In other words, a substantial number of errors were found among non-CS code that could have been eliminated without human intervention.

These two studies point to automated code analysis tools as the answer to the woes of non-CS scientists. Unfortunately, not all faults capable of jeopardizing scientific results can be statically detected. For example, in 2006, a protein crystallographer was forced to retract five papers which had been published over the previous five years, the most influential of which had been cited 364 times (Miller, 2006). The bug in this case inverted the output protein structure, which may have been detectable in testing, but couldn't be found by a static fault analyzer or an automated code-cleaning tool.

These examples demonstrate that there exists a risk of inaccurate science being published and accepted due to coding errors. In order to prevent this from happening in the future, further understanding of how non-CS scientists write code is crucial–starting with an examination of what might have caused these errors to slip through in the first place.

Causes of Bad Code

The first possible explanation for bad code in scientific results is a lack of testing, as less than half of scientists feel comfortable with their ability to test, and only 60% believe it to be important (Hannay et al., 2009, p. 5). This lack of emphasis on testing relates to the concept of non-CS scientists as purpose-first programmers. Since scientists' purpose in coding is to test and model scientific theories, the code and theory are tightly coupled in their mind, so scientists often interpret incorrect output as an error in their theory rather than their code (Sanders & Kelly, 2008). A pertinent example of this comes from the protein crystallography case: scientists in that lab, as well as other biologists, noted that the protein structures indicated by the erroneous program were inconsistent with "a lot of things" (Miller, 2006, p. 1857). However, the scientists assumed that the errors resided in their theory, not their code, and it wasn't until another lab found a fully contradictory result that the software bug was revealed (Miller, 2006). Scientists either don't know how to test or don't see it as important, so many of them are not learning or deploying the skills necessary to adequately test their software. Alternatively, these findings raise the possibility that some fields of science don't have easy ways to generate correct results to test their software against.

Research on CS students learning to program proposes another potential cause for software errors in science. It suggests that these novice programmers–who have similar amounts of education to non-CS scientists who don't take many formal classes–often form incorrect mental models of how programming works, even with formal education. Novice programmers use the analogy of conversing with a human to begin to conceptualize how the computer interprets their code (Pea, 1986). This presents opportunities for misconceptions that cause bugs such as: parallelism bugs, where novices believe that multiple lines are executing simultaneously; intentionality bugs, where novices think the computer can interpret more of their intent than is actually written into code; metonymy bugs, where novices assume that keywords in a programming language mean exactly the same thing as in English; or algebraic bugs, where a novice assumes that a math expression means the same thing in code, though it may not (Pea, 1986; Qian & Lehman, 2017). Since novice programmers can form these incorrect models despite formal education, we hypothesize that scientists learning in informal settings will be as prone if not more prone to the same misconceptions.

To remedy inaccurate mental models, novices must be "made dissatisfied" with their prior conceptions (Qian & Lehman, 2017, pp. 1-14). One approach suggested by Qian & Lehman is to use good programs to exemplify how code should be written, but this cannot occur when self-taught scientists have no higher authority to determine which programming examples they see are good or bad (2017). Furthermore, these scientists' primary examples may come from other self-taught scientists, whose code may be of poor quality (Hatton, 1997). Another method is to teach good programming strategies and patterns explicitly to novices, but this rarely occurs outside of formal instruction (Qian & Lehman, 2017). Finally, having a program fail a test despite expectations that it will work forces novices to re-evaluate their conceptions, but the general lack of testing in non-CS scientific computing prevents this from occurring often–as Sanders & Kelly's interviews of 16 non-CS scientists revealed, "Scientists don't test the code in the computational engine for its own sake," (2008, p. 26). Either non-CS scientists' lack of formal education or their identities as purpose-first programmers seem to preclude many of them from learning programming standards, testing strategies, and other tools which could help them improve the accuracy of their results.

A final possible cause for observed inaccuracies in scientific results: purpose-first programmers, like non-CS scientists, generally don't concern themselves with the details of their programs; they only understand the minimum amount necessary for their purpose (Cunningham, 2020). This can lead to producing code that they believe to be correct, but which actually contains errors, such as the earth science code where the inaccuracies of floating-point calculations on computers led to erroneous output (Hatton, 1997). Another example of this type of error comes from a scientific software designed to measure neuroanatomical structures, which produced different results depending on the software version and operating system of the workstation (Gronenschild et al., 2012). There, the scientists likely failed to understand subtle differences between how different operating systems execute code, contributing to the observed inaccuracies.

Unfortunately, as we elaborate below, the obvious solution- more formal education for non-CS scientists- has its own problems in practice.

Formal CS Education Isn't Ideal

In many institutions, non-CS scientists are required to take programming courses. For example, at the U, physics majors are required to take a computation lab (Department of Physics and Astronomy, 2023). However, research shows that first and second year CS students have only 3.4% fewer errors than self-taught scientists on programming tasks, indicating that 1-2 years of formal education may not be better than informal learning for preventing errors in coding tasks (Rafalski et al., 2019). This likely results from the tendency of introductory programming courses to "teach & drill," where the instructor provides the solution for each problem faced in exercises. This circumvents steps 2-4 of the standard framework for problem solving in programming: search for analogous problems, search for solutions, and evaluate potential solutions. The instructor simply provides the solution for the given problem and leaves it to the student to implement. Therefore, it is possible for CS students to complete two years of undergraduate instruction without experiencing a full problem-solving cycle (Izu, 2021). When paired, these results indicate that entry-level CS courses may not give non-CS scientists the error-prevention skills necessary for their work, yet specially tailored courses or combinations of courses may be able to address this–if CS educators better understand the needs of non-CS scientists.

CHAPTER 3

RESEARCH QUESTIONS

Prior work makes it clear that non-CS scientists, as purpose-first programmers, constitute an entirely different audience from the CS students that CS educators focus on, indicating that traditional methods in CS education may not work for non-CS scientists. Additionally, these works show a lack of formal education among non-CS scientists, insufficiencies in low-level CS classes for non-CS scientists, inaccurate mental models among beginner programmers, uncertainty about testing strategies in non-CS fields, and erroneous results from scientific software, demonstrating that non-CS scientists may need better CS education to ensure the accuracy of their scientific results. To help bridge the conceptual gap between traditional CS education and the needs of non-CS scientists, this thesis seeks primarily to understand how and why the programming styles and software validation methods of non-CS scientists differ from CS norms, with a specific focus on the University of Utah. I divide this central research question into four specific queries:

- RQ1: How do non-CS scientists at the U who write code learn to do so?
- RQ2: Does code written by these scientists contain antipatterns which indicate conceptual misunderstandings about programming or make errors more likely?
- RQ3: Do these scientists have the skills needed to rigorously verify the correctness of their code?
- RQ4: What can the U do to improve the software development skills of its non-CS researchers?

CHAPTER 4

METHODS

First, I examined samples of code submitted by eight non-CS scientists at the University of Utah. Then, I interviewed three of these scientists about their education, use of code in their research, and their perceptions of general attitudes towards programming in their field. Phase 1 looks for specific issues in code written by scientists at the U, and phase 2 contextualizes these issues within the educational experiences, scientific fields, and institutional cultures of the code's authors. This human subjects research was exempted from review by the University of Utah IRB (IRB #00163640).

Phase 1

Recruitment

The first phase of the study involved a comprehensive direct analysis of code provided by eight non-CS scientists. For this study, I sought scientists who do research in a non-CS field at the U, write their own code, and do not have a degree in computer science. I recruited participants by emailing institutional contacts at the University of Utah, asking if they would like to participate, and asking them to refer other potential participants. I asked each participant to submit as many code files as they were comfortable with/had time to collect, with the requirement that they had written every code file and used it in research. This process was repeated until at least eight participants had filled out consent forms and submitted code. The code files collected were all used to perform research and were mostly or completely written by the participant who submitted them. All eight participants are University of Utah researchers who have mutual institutional contacts with this study's authors.

Code Analysis

Most scientists sent a limited number of files and I could review all of them. For scientists who submitted more code than could be reviewed, I skimmed 4-8 files and selected at least two that contained at least 500 lines cumulatively More than two files were summarized for certain participants at my discretion when said participant had unique or unusual code or if more context was needed to understand some of their other code. To ensure a thorough analysis, I read each file completely before moving on to the next one.

I produced two data spreadsheets from the code files. One is a non-exhaustive list of examples of antipatterns observed in submitted code. I chose not to attempt to exhaustively categorize every antipattern example due to time constraints and because a diverse set of anti-pattern examples better answers RQ2. Cataloging duplicate anti-patterns would not add understanding of how non-CS scientists at the U may misunderstand code or use bad practices. Instead, I searched for a broad variety of anti-patterns by reading through the code and documenting anti-pattern instances which were generally dissimilar to the other examples already documented. Gathering a diverse set of anti-pattern examples allowed me to better understand the various ways in which non-CS scientists differ in their coding practices, giving more depth to an issue which is not uniform among this study group.

The second data spreadsheet is a list of summaries describing each analyzed file, with summaries categorized according to common patterns and antipatterns observed in the code. I applied pattern and antipattern categories to files based on practices that can be easily observed to occur (or not occur, in the case of lack of comments) throughout a file. I limited these categories to language-independent stylistic characteristics that are easily identifiable while skimming through a file and do not require in-depth code comprehension. Assignment of a category to a file indicates that the file generally uses the corresponding pattern/antipattern, though the file may not use the pattern/antipattern in some specific cases. The categories of patterns/antipatterns in file summaries do not map exactly to the categories assigned to antipattern examples. In cases where an antipattern example category isn't present in the file summaries, the example category was never pervasive throughout an entire file. These generally represent instances where a subject failed to employ optimal code structure in a specific case. Some anti-pattern categories occur in both products because the anti-pattern can be identified in a specific example and is also prevalent throughout a file, such as use of magic constants. Categories that only appear in the file summaries are only useful when observed consistently throughout an entire file. For example, a single code block isn't significant on its own, but a file that consistently uses code blocks indicates a certain style preference by its author.

I chose these methods for the file summaries to give a wider view of the answer to RQ2. While individual examples of antipatterns provide good insight into which mistakes non-CS scientists at the U make while coding, the non-exhaustiveness of these examples limits any conclusions about the prevalence of coding errors and how they differ between subjects. By summarizing all analyzed files, a broader understanding of how code style and practices vary between scientists can be obtained.

Phase 2

Selection

In phase two, I interviewed three of the scientists from phase one for one hour about a variety of topics related to: their submitted code, their use of programming for research, their opinions on programming style, and their view of the general state of programming in their field. The three scientists were selected with the goal of representing a wide variety of coding styles. Since only four scientists consented to participate in phase two, options were limited. I invited scientists to participate in interviews after analyzing their code. The final three interviewees were an astrophysics professor, an atmospheric science professor, and a graduate student in astrophysics. All three have notable differences in coding style, both compared to each other and compared to norms in computer science, and these differences were reflected in their interviews.

Interview Format

I used a semi-structured interview format with a list of pre-prepared questions (see Appendix A) for each interview, but also asked follow-up questions on the spot and allowed interviewees to discuss whatever subjects came to mind while answering a question. I generated these questions based on topics of interest from the background research and the interviewees' code. Interviews were recorded by two devices in the room (or my computer audio and Zoom's audio recording, in the case of one interviewee who I interviewed over Zoom) and transcribed by Microsoft Stream's AI.

Interview Analysis

I thematically analyzed the interview transcripts using open coding. In the open coding process, I looked for topics that were substantively discussed in multiple interviews and related to my research questions. After generating topics, I went through each interview transcript in detail, line-by-line, and assigned excerpts from the transcripts to topics that they addressed. I was able to assign the vast majority of the duration of each interview to a relevant topic, and many excerpts fit into multiple topics. Finally, I reformulated the identified topics as questions that help to answer my overall research queries, answered these questions based on the associated interview excerpts, and grouped the answers into four overall categories which I cover in the results section.

CHAPTER 5

POSITIONALITY

I'm a fifth-year computer science undergrad at the U, and I also have four years of experience developing software for websites and web apps. I became interested in this topic due first to seeing research code written by a friend of mine (an undergrad in a non-CS field), and second to a prior research project of mine, which examined some of the subjects covered in this thesis's background. I've learned about good coding practices both formally in school and experientially from writing and maintaining code at my job. Both my education and experience have taught me how certain programming styles can reduce the likelihood of errors and make it easier to modify the code in the future–I've seen errors in code stem from confusing style, and I've had difficulty modifying my own code because I didn't write it in a flexible manner. After seeing my friend asked to write research code with no formal education and discovering the various errors detailed in the background, I wanted to investigate whether the U has issues with research code quality and bring attention to these issues if so.

CHAPTER 6

RESULTS

<u>**Phase 1**</u>

Issues

While reading submitted code files line-by-line, 50 individual examples of antipatterns were selected for study, added to a spreadsheet of antipattern examples, and later categorized by pattern/antipattern. I selected examples of antipatterns based on my personal knowledge of good programming practices, informed by my education in CS and employment as a lead software developer. I did not start from a list of code guidelines; instead, I used a process similar to code review in my workplace: I read through the code and look for parts that I know can be improved. Selected examples represent cases where changes to the code would improve the readability and comprehensibility of the code, reduce the probability of future errors, minimize redundant code, and/or improve performance.

I identified more issues with flow control than any other category, collecting 20 examples where additional or repeated code could have been avoided through the use of flow control structures. The clearest example comes from engprof/1.m, where what could be a nested loop structure is instead written out as a long sequence of if statements over 48 lines of code. This code displays a table in the program output, with a number of rows equal to the value of `kk` (a parameter which is assigned a value between 1-11 earlier in the program). Each row also has one more column than the previous row. The full code isn't shown for brevity, but this pattern continues up to `if kk>11`:

```
if kk>2
  disp(['EM-
3       ',num2str(round(angtheta(3,1))),'       ',num2str(round(angthe
ta(3,2))),'       ',num2str(round(angtheta(3,3)))])
end;
if kk>3
  disp(['EM-
4       ',num2str(round(angtheta(4,1))),'       ',num2str(round(angthe
ta(4,2))),'       ',num2str(round(angtheta(4,3))),'       ',num2str(r
ound(angtheta(4,4)))])
end;
```

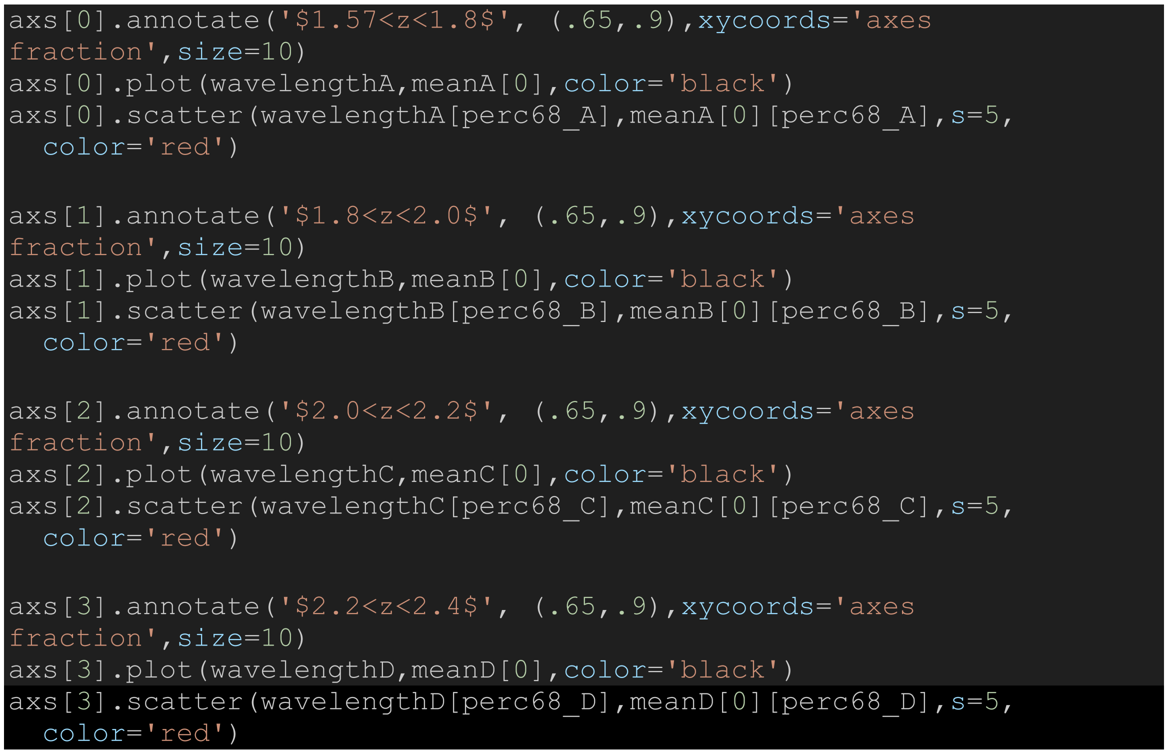
Figure 1: Two if statements (out of 11) which progress in a way easily expressed by a nested loop.

```
for i=1:kk
  disp('EM-',num2str(i),'       ')
  for j=1:i
    disp(['       ',num2str(round(angtheta(i,j)))])
  end;
end;
```

Figure 2: An example of how Figure 1 could be re-written to eliminate many lines of code. Technically, this example produces a slightly different result because Matlab's disp() function always prints a newline. Since there isn't a version of disp() which doesn't, the real solution would have to use string concatenation and print below the inner loop, which I've omitted for brevity.

Other cases are similar: in all examples, it appears that the authors did not think to use a certain control structure. Generally these were instances where a loop would shorten the code, but I also found two examples where a switch/case statement would be more appropriate than an if-elseif structure. Most of the authors who neglected to use an appropriate control structure in a certain place used that control structure elsewhere, indicating that they're aware of the structure but don't always recognize where to use it. In one instance, an author did not use a switch statement anywhere, including one place I identified as appropriate for it, indicating that they may not know about switch statements.

12 instances were categorized as repetition, a subcategory of flow control. In all these cases, the author could reduce length and complexity by implementing a helper function or storing a computation in a variable rather than repeating exact lines of code or duplicating logic. To illustrate, here is a snippet from astrograd/1.py:



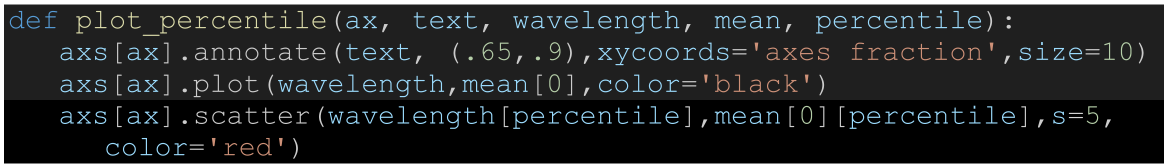
```
axs[0].annotate('$1.57<z<1.8$', (.65,.9),xycoords='axes
fraction',size=10)
axs[0].plot(wavelengthA,meanA[0],color='black')
axs[0].scatter(wavelengthA[perc68_A],meanA[0][perc68_A],s=5,
  color='red')

axs[1].annotate('$1.8<z<2.0$', (.65,.9),xycoords='axes
fraction',size=10)
axs[1].plot(wavelengthB,meanB[0],color='black')
axs[1].scatter(wavelengthB[perc68_B],meanB[0][perc68_B],s=5,
  color='red')

axs[2].annotate('$2.0<z<2.2$', (.65,.9),xycoords='axes
fraction',size=10)
axs[2].plot(wavelengthC,meanC[0],color='black')
axs[2].scatter(wavelengthC[perc68_C],meanC[0][perc68_C],s=5,
  color='red')

axs[3].annotate('$2.2<z<2.4$', (.65,.9),xycoords='axes
fraction',size=10)
axs[3].plot(wavelengthD,meanD[0],color='black')
axs[3].scatter(wavelengthD[perc68_D],meanD[0][perc68_D],s=5,
  color='red')
```

Figure 3: Four code blocks that perform the same operation with slightly different parameters.



```
def plot_percentile(ax, text, wavelength, mean, percentile):
   axs[ax].annotate(text, (.65,.9),xycoords='axes fraction',size=10)
   axs[ax].plot(wavelength,mean[0],color='black')
   axs[ax].scatter(wavelength[percentile],mean[0][percentile],s=5,
      color='red')
```

Figure 4: Example of how the code blocks in Figure 3 can be abstracted into a helper function.

12 examples were categorized as readability issues that could have been resolved by following a clear coding standard. These mostly involved excessive operations on one line, making the code's logic difficult to understand. From astroprof/3.pro:

```
apim=rot(fshift(apim,paperbgd[4+ifpm*3]/0.12096,paperbgd[5+ifpm*3]/0
.12096),$paperbgd[6+ifpm*3],/interp)*3.1865e5*paperbgd[ifpm]*0.12096
*0.12096
```

Figure 5: A very complex calculation in a single line, with repeated computations and constants.

```
ifpm3 = ifpm * 3
const1 = 0.12096
apim = rot(fshift(apim, $
                  paperbgd[4+ifpm3]/const1, $
                  paperbgd[5+ifpm3]/const1), $
           paperbgd[6+ifpm3], /interp)
apim *= 3.1865e-5 * paperbgd[ifpm] * const1^2
```

Figure 6: An example fix for Figure 5 which abstracts two reused values to variables and makes use of line breaks to enhance readability. I would use more descriptive variable names if I was a domain expert. I don't know whether a domain expert would agree that the first example is difficult to read or that this suggested change improves the readability.

The same issue in astrograd/1.py:

```
func = lambda x: np.sum((flux1[mask]-
(x[0]*wl[mask]**x[1])*flux2[mask])**2 / (sig1[mask]**2 +
((x[0]*wl[mask]**x[1])*sig2[mask])**2) )
```

Figure 7: A complex function defined in a single line, with one computation repeated. Elsewhere, this author does use def to define multi-line functions.

```
def func(x):
      var1 = x[0] * wl[mask]**x[1]
      numerator = (flux1[mask] - var1 * flux2[mask])**2
      denominator = sig1[mask]**2 + (var1 * sig2[mask])**2
      return np.sum(numerator / denominator)
```

Figure 8: An example of how to make Figure 7 more readable. A domain expert could use more descriptive variable names. Again, not all astrophysicists might agree that this change is warranted or necessary.

In other examples categorized as readability, authors use non-descriptive variable names, structure code in a manner which makes code tracing more difficult (such as storing unrelated variables in array positions or reusing a variable for different purposes), and raise errors without messages.

Commenting and documentation represented another broad category, with a total of 15 examples. Three were assigned the sub-category of no comments, where complex functions or calculations lacked any explanation of their purpose. In many cases, functions did not have a docstring documenting parameters and the return value. One was sub-categorized as a concerning comment for contradicting the line of code that followed it. Ten instances demonstrated inappropriate use of magic constants (a variable with a single fixed value) without descriptive names or explanatory comments. While these were likely comprehensible to the author, common practice in the CS community teaches that magic constants make refactoring and sharing code much more difficult. While some of the participants submitted unshared code, much of the code in this study has been shared between multiple authors, including some of the code that I identified magic constants in.

Four examples indicated misunderstanding of language features, a category more specific to the programming language than flow control. Two were categorized as misunderstandings of variable scope; the authors defined local variables as global variables outside of the scope of the only function where they are used, or they employed functions instead of global variables, such as this example from atmosprof/1.py:

```
def fontsize():
    fontsize = 6
    return fontsize
```

Figure 9: A function which returns a constant variable. A global constant could be used in place of this function; the function call is unnecessary as it performs no computation. Python's interpreter cannot do much optimization, so this program may suffer a performance penalty due to function call overhead if the font size is accessed many times.

Another example showed failure to initialize variables in C++, a known source of software bugs in languages with explicit memory management. Finally, a sub-category with four examples covered broader misunderstandings like not utilizing native data types in appropriate situations, misunderstanding order of operations (from astrograd/4.ipynb: `exclude = (((sky | coverage) | broken) | targeting ) | nodata`– here the order of operations enforced by the parenthesis is the same as without them), and failing to employ idiomatic operators (such as mecheprof/1.cc: `c = c + a * (y-y_ref);`—where the code could have employed the `+=` operator).

The remaining two examples were characterized as miscellaneous. One defined a series of unused variables, and the other responded to bad function parameters by terminating the program without an error message rather than throwing a meaningful exception.

This qualitative analysis uncovered 50 examples of antipatterns that provide insight into how non-CS scientists sometimes eschew programming standards endorsed within the CS community. The prevalence of issues related to flow control, repetition, readability, and commenting/documentation show that non-CS scientists at the U are sometimes either unaware of best practices or prioritize getting code to function over ensuring maintainability and optimization. In some cases, this likely suits their purposes for programming, but when applied to code that's reused or shared among multiple authors, it can slow updates and increase the probability of bugs.

I present these examples not to indict the source code, but to show specific instances where the code would benefit from certain coding standards common in the software development community. To be clear, the vast majority of code examined in this study is of good quality, as evidenced by the limited instances of misunderstanding language-specific features and the absence of any functional bugs apparent to a CS undergraduate. In answer to RQ2, it appears that many of the subjects in this study do have occasional issues which indicate conceptual misunderstandings or make errors more likely. However, as shown by the file summaries, these scientists write high-level code for a variety of different fields and purposes, and their coding styles and antipattern rates aren’t homogenous.

Files

After reading each file in its entirety, I summarized all 23 analyzed code files in a spreadsheet. While reviewing these summaries, I generated and assigned four pattern and three antipattern categories to the files. To generate categories, I looked for overall patterns and antipatterns that have three properties: they must be representative of a practice widely considered to be definitively positive or negative in the software development community; they must be representative of a practice that the University of Utah’s School of Computing teaches its undergraduate students to employ or avoid; and they must appear in multiple submitted code files. My determinations of the general software community’s views on particular practices are necessarily based on and limited by my experience and education. However, I limited the categories to practices that I have not seen any controversy over–I have only seen wide agreement on the desirability or undesirability of the categories generated for this thesis. This three-property category definition allows me to better answer RQ2 by examining whether non-CS scientists’ code follows well-established best practices in general. I assigned a category to a file when the file exhibits the corresponding pattern or antipattern in the majority of places where the author could choose to employ it.

In the antipattern section, “Confusing Formatting” indicates that a file frequently employs a formatting style which hinders readability, such as many expressions on one line, lack of whitespace lines or whitespace lines used in an unpatterned way, or inconsistencies in how different expressions of the same type are formatted. This category is based on principles taught in standard CS development classes at the U like Object-Oriented Programming, Software Development I, and Software Development II, as well as encouraged in the broader community, such as the common inclusion of a ruler in code editors to indicate the maximum recommended line length. “Repeated Code” describes files where a notable percentage of code lines in the file could be eliminated through abstraction or better flow control, such as employing loops and encapsulating repeated operations in helper functions. It’s well-known (and taught in Object-Oriented Programming at the U) that copy-pasting code to change a few things indicates that a helper function would reduce work for the programmer while better allowing code reuse. “Magic Constants” is a well-known antipattern in the CS field, where string and number values are used without documentation or a descriptive variable name. Object-Oriented Programming and Software Development classes at the U emphasize avoidance of magic constants so that other programmers understand the values’ purpose.. “Inconsistent Comments” indicates that the commenting of the file seems insufficient to allow a domain expert other than the author to easily understand the functionality of the code and purpose of various functions and variables within it. Object-Oriented Programming, Software Development I, and Software Development II at the U require commenting for full assignment credit, and many popular industry coding standards specify a minimum level of comments in code files, particularly for function definitions.

For patterns, “Commenting” indicates that the author frequently uses comments to describe important variables, functions and parameters, and code blocks, making the file almost certainly comprehensible to another domain expert. This is in opposition to and motivated by the same reasons as the “inconsistent comments” antipattern, however, the two are not mutually exclusive–files categorized as neither use some comments, but not a lot, so I cannot conclusively determine whether another domain scientist would be able to easily understand the code. “Consistent Code Style” indicates that an entire file appears to follow some sort of coding standard, whether explicit or only conceptualized internally by the author. Code standards are taught in Software Development I and II at the U, and many large software development teams use coding standards, some of which have been published on the internet. “Code Blocks” indicates that most of the file uses whitespace lines to divide the code into blocks which each perform a single task. Software Development I and II teach this as a way to make code more readable, and all of the code that I’ve seen in the workplace employs code blocks. Table 1 presents a list of analyzed files and the anti-pattern/pattern categories assigned to them:

	Anti-Patterns				Patterns		
File Author/Name	Confusing Formatting	Repeated Code	Magic Constants	Inconsistent Comments	Commenting	Consistent Code Style	Code Blocks
astroprof/1.pro			X	X		X	
astroprof/2.pro		X	X	X			
astroprof/3.pro		X	X	X			
chemeprof/1.m		X		X			X
chemeprof/2.m			X	X			X
chemeprof/3.m					X		X
chemeprof/4.m					X		
chemeprof/5.m				X			
chemeprof/6.m					X	X	X
engprof/1.m		X				X	
engprof/2.m					X	X	
engprof/3.m	X		X			X	
medprof/1.py		X	X		X		X
medprof/2.py		X	X		X		X
astroprof2/1.py			X		X	X	X
astroprof2/2.py			X	X		X	X
astrograd/1.py	X	X	X	X			X
astrograd/2.py	X	X	X	X			
astrograd/3.py	X	X	X	X			
astrograd/4.ipynb		X	X				X
atmosprof/1.py			X		X	X	X
atmosprof/2.py					X	X	X
mecheprof/1.cc	X	X	X		X		X
mecheprof/2.cc		X			X	X	X

Table 1: Anti-Pattern and Pattern categories assigned to files, sorted by file author.

This table exemplifies the diversity of coding patterns in the submitted files. Some authors exemplify multiple antipatterns in the majority of their files, whereas others have almost no regular antipatterns in any of their files. This table shows that much of the submitted code uses antipatterns that hinder maintainability. The majority of submitted files contain some repeated code or magic constants, which both complicate refactoring efforts. On the patterns side, it appears that many non-CS scientists do pick up common practices like commenting and code blocks. While I did identify many examples of problematic anti-patterns earlier, the file summaries show that some participants–such as atmosprof, astroprof2, and chemeprof–generally avoid antipatterns and use patterns familiar to the software engineering community. . Thus, the determination (in answer to RQ2) that some participants write more error-prone code varies by participant. Phase one only addressed RQ2, so in phase 2, I interviewed scientists with a focus on the other three research questions.

<u>**Phase 2**</u>

I selected three scientists to interview: a professor of physics and astronomy (Astroprof), a PhD student studying astronomy (Astrograd; within a different research group than Astroprof), and a professor of atmospheric science (Atmosprof). I identified and coded twelve topics from the interview transcripts, which were organized into four broader subjects: education, where I investigate how non-CS scientists learn to code and how helpful CS education is for non-CS scientists; non-CS code quality, where I ask the interview subjects' opinions on how code should be written and whether they think peers in their fields have enough programming knowledge to do their work accurately; code in research, where I look at the necessity of programming, how code is tested outside CS fields, and whether these fields have accurate oracles (cases where the output of a program for a given input is known) for testing use; and social norms around coding, where I evaluate the sharing of code in non-CS fields and whether programming quality is valued by the field in general.

Education

All three interview subjects reported very similar experiences when learning to program: all three took a single CS class (which wasn't required for their degree) in a language that they no longer use, then were expected to write programs in grad school and self-taught the additional things they needed to learn. Atmosprof described programming in grad school as “baptism by fire”; Astroprof was advised not to take CS classes because “you learn as you go.” Atmosprof and Astrograd described how current grad students at the University of Utah generally teach themselves and learn from other grad students with little help from professors, who are either too busy or don’t know enough about programming to teach it.

All three participants generally agreed that formal CS education would help to improve the quality of code in their field, but outlined a variety of caveats to this view. Atmosprof pointed out the time tradeoff between teaching students good coding practices and having them graduate on time, implying that adding CS classes to their graduate program would require extra semesters for their students. Astroprof and Astrograd both agreed that much of what CS programs teach isn’t useful to most astrophysicists, implying that choosing to take CS classes can be difficult when much of the content doesn’t apply to one’s field of study. Furthermore, Astrograd pointed out that some useful CS classes for astrophysicists, such as multi-processing, require a lot of pre-requisites, making them inaccessible to most astrophysics students.

Overall, these interviews showed that these three scientists have little formal education. Formal education provided some benefits to them, but they believe it could be tailored better to the specific purposes of their fields. In answer to RQ1, the participants interviewed here are mostly self-taught.

Non-CS Code Quality

Next, I investigated these three scientists’ opinions on how code should be written. Astroprof and Atmosprof agree that readability is the second most important goal in code after functionality; Astrograd elected efficiency in time and space. All three use comments to describe the functionality of the code, but vary drastically in the amount– Atmosprof likes to comment every few lines of code, whereas Astroprof prefers not to comment code that won’t be shared with others (“it's much less cluttered to me”) and Astrograd falls in between. All three agree on the importance of comments when sharing code, and Atmosprof additionally noted how comments help their future self understand code. Otherwise, these scientists displayed a diverse range of opinions about proper coding style with little agreement between them. Astroprof prioritizes readability over speed, unlike Astrograd, which could reflect a difference in the computational requirements of the code they write. Atmosprof mentioned some things not touched on in the other interviews, such as how they prefer to break code into smaller snippets to aid correctness and a general goal of minimizing redundant code while programming. Astroprof likes to leave old code in as a comment after re-writing it to improve readability.

Despite a lack of broad consensus on an ideal coding standard, all three subjects generally agreed that scientists in their field don’t utilize best practices when programming. All three mentioned instances of bugs found in code written by other researchers in their field. Astrograd believes that any computer scientist, “even at an undergraduate level, would be like, ‘Yikes!’ if they looked through “most scientists’” code. Astroprof mentioned that many researchers haven’t been trained in best practices, and this often becomes an issue when they move to industry jobs, where these practices are enforced. Atmosprof noted that many scientists in their field don’t know the supposedly-standardized netCDF file format used to store atmospheric data, so each time

Atmosprof imports data into code, they must figure out what format the data uses and re-write their code to accommodate it.

All three expressed reservations about the quality of the code they produce. Both professors admitted to having used code which produced strange or slightly erroneous results, justifying their decisions with the fact that the errors in question were too small to affect the final results. Both were under time constraints, which played into their decision to use the code, and its results, before they were fully satisfied with it. Atmosprof even admitted that, "there probably are coding mistakes affecting everything we do at some level," but questioned whether the time invested in hunting down these mistakes would actually benefit the experimental snowfall forecasting that they do. They see their studies as low-consequence, and they question whether intensive software verification would be worthwhile to their research. However, they do caveat that they see software verification as much more important to scientific disciplines where results have more consequences. Astrograd noted that their wider collaboration uses a variety of good practices familiar to computer scientists, such as version control, object-oriented code, defined coding standards, and code review, but Astrograd themself doesn't exactly understand how object-oriented code works and isn't comfortable writing it.

This further expands on my previous answer to RQ2, showing that these scientists believe that issues with software quality may be more widespread than is shown by the limited sample of code that I reviewed. While having only three perspectives limits this finding, it does seem that non-CS software quality at the U may need improvement.

Code in Research

All three scientists unanimously agree that coding is crucial for their research. Astroprof stated that all stages of their research involve coding, and Astrograd estimated that 70-90% of their research process involves writing code. They mainly use programs to format and analyze data, model theories, and visualize results.

All three subjects generally test code by giving it a limited number of sample inputs and manually inspecting the results for anything that seems off, though both professors acknowledged that a correct test result doesn’t ensure a correct program. These two also stated that they do more verification when they have to release the code to other researchers. All three use print statements to perform this intuition test on specific pieces of code within the overall program.

Both professors mentioned being able to test some programs by calculating the result by hand and checking that the output matches. Astrograd mentioned that their research group tests simulations with sample data that has known results, and Atmosprof alluded to very specific scenarios with analytical outcomes that can be used to verify weather models (which Atmosprof does not write). Both professors seemed to have difficulties with testing due to a lack of oracles to test with. Astroprof mentioned that, in many situations, they don’t know if an odd-looking result comes from a bug or not because they don’t know what the output should be. Atmosprof enumerated a variety of scenarios that they can’t test, such as plotting precise weather data over a map and verifying the correctness of weather model predictions when real data from weather stations is sparser than the model output.

In answer to RQ3, it appears that a lack of oracles provides a real problem for intensive testing of scientific software, forcing these subjects to rely on intuition for much of their testing purposes. It also appears that testing rigor is often determined by the availability of oracles and whether the code will be shared, rather than the consequences of an incorrect output.

Social Norms Around Coding

All three scientists generally don't share the code that they write, mostly because they aren't confident enough in its quality to be comfortable sharing it. Astrograd stated that they don't know how to write object-oriented code, the standard in their field, so they aren't comfortable releasing the non-OO code they write. Atmosprof mentioned that "management" wants their lab to focus on science, not the "operational environment," so they don't write any releasable code that would be used in weather models. Astroprof expressed concerns with the time suck maintaining code and dealing with bug reports if they released their programs; these activities wouldn't be rewarded by the U, so they don't want to invest the time.

In fact, all three subjects discussed how putting effort into improving their programming abilities wouldn't bring them any tangible benefit. Both professors expressed passion for programming: Astroprof called it, "the best part of the job at a certain level," and Atmosprof elaborated on how more advanced coding techniques advance their passion for better snowfall forecasting, but neither can justify spending much time learning how to program better since there's no institutional reward for doing so. Both work on code just enough to get it working, then leave it so that they have enough time for other professorial duties. Atmosprof referred to this as, "limping along plugging holes." Similarly, Astrograd doesn't learn more about coding unless they need the new knowledge for a specific task: "basically all" of the things they've learned have been for a specific purpose. Atmosprof clearly knows what they don't know, and mentioned wanting to know more about best practices like version control and object-oriented programming, so they regret that they don't have the time to do so. Astroprof stated that having a dedicated software engineer on staff would increase productivity enough to be worthwhile but, "no one would ever fund it." Atmosprof doesn't believe they'd be able to find someone to help with their coding work; stating, "nobody would ever want to do it." All three scientists showed marked interest in improving their programming skills, but none could justify taking much time out of their already-crammed schedule to do so.

In answer to RQ4, these scientists make it clear that, as far as they are aware, the University of Utah does not give them dedicated time and resources to improve their coding skills at a graduate or professional level. While it's possible that the U does have these resources and these three participants simply aren't aware of them, that would still indicate a failure on the part of the U to publicize these resources, especially since both professors have been at the U for a number of years. This dearth of resources causes particular concern in combination with my finding for RQ2 that the general quality of software in these fields may be questionable.

CHAPTER 7

DISCUSSION

While limited in its sample size, this study provides answers to my four research questions, which generally agree with the conclusions drawn by prior work. Like most of the respondents surveyed by Hannay et al. (2009), the subjects I interviewed in phase two had little formal education in programming, and they didn't feel that the classes they took were tailored to their fields. Likely as a result of this, I identified a number of anti-pattern examples in programs submitted by scientists, though file summaries showed that code quality varies dramatically between different subjects. Like the scientists interviewed by Sanders & Kelly (2008) and the majority of respondents to Trisovic et al.'s survey (2021), the scientists I interviewed do not have robust testing strategies which work for all of their needs. Finally, the University of Utah does not appear to be taking necessary steps to remedy these shortcomings in scientific software development.

None of my interviewees took more than one formal course in software development. Qian & Lehman (2017) suggest that novice programmers must be "made dissatisfied" with their work to learn how to write better code, which only occurs in a context where feedback is given. Since these interviewees spent little time in an environment where this feedback was available before their "trial by fire" going into research, we see a few examples of conceptual bugs similar to those detailed by Pea (1986) and Qian & Lehman (2017), such as the 20 antipattern examples where scientists failed to recognize how a flow control structure or helper function could simplify their code. Structured feedback could help these scientists notice situations where they should use certain flow-control patterns, but lack of such feedback prevents them from realizing these occurrences. While I did not identify any compromising bugs in phase one, many of the files showed pervasive use of antipatterns.

Moreover, both my interviews and the prior work (Sanders & Kelly, 2008; Hannay et al. 2009) show that scientists often lack appropriate methods to test their software, forcing them to rely on intuition to decide whether to investigate their code's correctness further. In answer to RQ3, it appears that the U should worry about the potential for erroneous publications, given that researchers here don't always rigorously test the software used to produce their results. Future cross-disciplinary research involving both software verification experts in CS and experts in non-CS fields should investigate what software verification methods can be applied in situations where an oracle isn't readily available, and the U should invest resources into teaching these methods to its non-CS scientists.

Regarding RQ4, it appears that a lack of software development classes tailored to scientists contributes to a general perception in non-CS fields that formal education doesn't provide much benefit (Hanney et al. 2009). All three interviewees had various issues with their options for formal education, ranging from the time commitment, to the relevance of subjects taught in low-level CS courses, to the inaccessibility of more relevant high-level CS courses. This fits with my conceptualization of non-CS scientists as purpose-first programmers, as defined by Cunningham (2020): these scientists have no interest in or need for many of the topics taught to CS students. Additionally, prior work shows that introductory CS classes may not provide substantial benefits in terms of error reduction over scientists' methods for self-teaching (Rafalski et al., 2019). Since current CS offerings don't provide a time-efficient way of furthering the purposes of non-CS scientists, they won't choose to take them.

There are a myriad of potential solutions which future research could investigate. Resources to encourage self-teaching, time allocated for learning more about programming, and even software developers available to help with programming tasks, could all benefit non-CS software. It's also important to investigate to what degree some of these fixes are necessary. For example, some non-CS scientists might not be interested in certain stylistic changes or agree that they make the code more readable. More importantly, improving code's maintainability isn't important in situations where the code won't be reused- CS faculty and experienced software engineers certainly write poorly-commented (and otherwise hard-to-maintain) code for one-off uses. Similarly, non-CS scientists don't need to test code when they don't rely on its results for scientific findings. Future research in this area must first consult with non-CS scientists to establish their precise needs before designing plans to help them.

While this paper does not specifically investigate how Large Language Models (LLMs) such as ChatGPT and Github Copilot are beginning to influence non-CS scientific software, these tools will inevitably play an increasingly important role in all software development; I have no doubt that many non-CS scientists already use an LLM to aid with their programming. At the time of writing, these tools still have a propensity to hallucinate and produce code that contains slight errors. In my experience, these errors are often small enough to be difficult to catch without detailed code tracing or comprehensive testing. I believe that purpose-first programmers—who wish to produce functional code while engaging with its low-level details as little as possible—are uniquely at risk of allowing LLM hallucinations into final products, because engaging with the details of a program is often necessary to catch these hallucinations. Limited testing in non-CS fields only exacerbates the potential for LLM errors to make it into research software, and while LLMs can write unit tests, they can't help in situations where an oracle isn't readily available. Despite these risks, LLMs can aid in code comprehension and often generate code of very high quality, making them an attractive tool for non-CS scientists and—if used correctly—a potential solution to some antipatterns in non-CS software. Given the potential risks and benefits, future research should also investigate the use of LLMs in non-CS programming, specifically how non-CS scientists currently use LLMs, how LLMs can be best used to aid these scientists, and how purpose-first programmers can prevent and catch hallucinations while utilizing LLMs.

Our interviewees made it clear that, as far as they are aware, the U does not provide them with the time or resources to improve their programming skills. Additionally, the CS classes they had taken weren't very applicable to their area of study. None of these classes, it should be noted, were at the U, but I don't think this absolves the U of the responsibility to ensure that its non-CS scientists have sufficient programming skills. While all three expressed interest in learning to code better, none have been given any incentive or opportunity to do so outside of their free time. While the U cannot control the undergraduate programs of the institutions that my interview subjects hail from, it should ensure that its undergrads take courses which are tailored to the fields they will go into by designing courses with input from both non-CS scientists and CS educators. Moreover, it should offer resources, time, and compensation for researchers to improve their coding knowledge in a domain-specific and time-efficient manner. A full answer to RQ4 goes beyond the scope of this study, but it likely requires cross-disciplinary collaboration between CS educators and experts in non-CS fields to design effective and relevant curricula.

CHAPTER 8

CONCLUSION

Software errors pose a concerning risk to the veracity of scientific results. While the quality of software written by non-CS scientists varies drastically, many examples of scientific code use practices which correlate with a higher likelihood of erroneous output. I found that attitudes and practices surrounding non-CS software development at the University of Utah sometimes matched patterns observed in background research, patterns which have led to published errors and high-profile retractions.

Scientists primarily fail to use best practices in their programs because they are unaware of them. Nevertheless, formal CS education isn’t tailored well to the needs of scientists, so they don’t have strong reasons to take CS classes. Additionally, as far as they are aware, the University of Utah does not provide support to any of the scientists interviewed in this study for improving their coding skills. As a result, these researchers don’t have any good opportunities to pick up techniques that would help ensure the accuracy of their scientific results.

Accurate oracles for testing also pose a major problem for many non-CS scientists. Simple techniques, like unit testing, are insufficient in situations involving massive amounts of data, visualization, or novel theoretical approaches. Non-CS scientists write code for applications that prove very difficult to test, and may need more advanced or novel approaches to software verification to be sure of their code’s correctness. Additionally, fields and institutions (such as the U) need to establish standards to help determine which code needs rigorous validation so that non-CS scientists don’t have to waste time testing code that they don’t rely on for scientific results.

I believe that the solution starts at an institutional level. Right now, the career researchers I interviewed at the U are not given time, resources, or encouragement (or have not been made aware of such resources) to improve their programming skills. By offering incentives to learn better code, the U could immediately improve coding knowledge and hence software quality. Alternatively, offering funding for dedicated software developers with CS education and experience within non-CS labs merits investigation as a way to improve the quality of code without taking time away from busy researchers. Regardless, the apparent lack of appreciation for the difficulty and importance of software within non-CS fields isn’t a sustainable way to ensure valid scientific results.

Additional cross-disciplinary research between CS and non-CS fields is also necessary to improve educational offerings and solve testing challenges faced by non-CS researchers. To accurately reflect both the needs of non-CS fields and the latest developments in CS educational research, CS educators need to collaborate with experienced scientists in non-CS fields to design relevant and useful curricula using the latest techniques for teaching programming. Moreover, non-CS scientists need experts in software verification to help study and develop validation and testing methods that work in situations where oracles are rare or nonexistent, which could then be incorporated into programming curricula for non-CS students. Input from non-CS scientists is crucial at this stage, as styles and methods from the CS world may not directly apply in non-CS fields. Finally, the pros and cons of LLMs for scientific software development merit investigation both as potential solutions and concerning sources of error.

While plenty of non-CS scientific software is of high quality, and the recommendations of this thesis have likely been implemented in certain labs, the fact remains that many scientific results in non-CS fields remain vulnerable to coding errors. The writing is on the wall, and it shouldn’t take a major retraction or other publicity disaster to spur change. I hope that, in the future, the University of Utah will value the programming done by its non-CS scientists and take seriously the importance of developing high quality and well-tested research software.

APPENDIX A

INTERVIEW QUESTIONS

Education

- How did you learn to program?
    - Classes?
    - What were the most useful things you learned?
    - What wasn't useful?
    - Did this formal education adequately prepare you?
- Did anyone mentor you in programming?
    - Were there any programs you used as examples while learning?
- How often do you have to teach yourself how to do something while coding?
    - What's your process for learning while coding?
- What do you think current undergraduates/graduates who will end up in a similar position to yours should be taught about programming?
- Based on your observations, do you think there is too much, too little, or enough programming education for researchers in your field?
    - Why do you work to improve your programming?

Process

- Do you use code throughout your research, or only during specific phases?
- What are your primary purposes for coding?
- What program do you use to edit code?

- How do you verify that your code works correctly?
- Do you ever write code to test other code?
- How do you go about resolving errors in your code?
    - Have you ever had an error that only became apparent well after the code was used?
- How often do you re-use your code?

Collaboration

- Does anyone else on your team write code?
- Does anyone else look at/edit/use your code?
    - Do they review it/give feedback?
    - Does your code leave your lab?
- Do you rely on code written by others in your lab?
    - Do you rely on code written by people outside your lab?
        - What are your criteria for trusting a program written outside your lab?
    - What are your criteria for trusting someone to write code in your lab?
- Does your lab try to reproduce results from other labs?
    - Do you use their code or write your own?
    - Have you ever had issues using another lab's code?

Coding Literacy

- Are you familiar with the term "Object-Oriented Programming"?

    - Why don’t you use it?
- Do you use version control ie git? If so, why?
- What, if any, are the most important things (other than functionality) for a programmer to keep in mind when coding?
- Do you believe there’s any particular “style” (ie rules other than the syntax of the language) of code that people should write with?
    - Are there any things allowed by the syntax of a language that you think people shouldn’t do?
    - What are the features of this style?
    - Why do you prefer this style?
- When and why do you choose to move code into its own function?
- How do you use comments in your code?
    - Are you just trying to describe functionality, or also your reasoning?
- Code specifics:
    - Differed by participant; related to things I observed in their code

REFERENCES

Cunningham, K. (2020, August 7). Purpose-first programming. *Proceedings of the 2020 ACM Conference on International Computing Education Research*. http://dx.doi.org/10.1145/3372782.3407102

Department of Physics and Astronomy. (n.d.). *Majors, Emphases & Minors*. University of Utah. Retrieved March 26, 2023, from https://www.physics.utah.edu/undergraduate-program/majors-emphases-minors/

Gronenschild, E. H. B. M., Habets, P., Jacobs, H. I. L., Mengelers, R., Rozendaal, N., van Os, J., & Marcelis, M. (2012). The effects of freesurfer version, workstation type, and macintosh operating system version on anatomical volume and cortical thickness measurements. *PLoS ONE, 7*(6), e38234. https://doi.org/10.1371/journal.pone.0038234

Hannay, J. E., MacLeod, C., Singer, J., Langtangen, H. P., Pfahl, D., & Wilson, G. (2009, May). How do scientists develop and use scientific software? *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*. http://dx.doi.org/10.1109/secse.2009.5069155

Hatton, L. (1997). The T experiments: Errors in scientific software. *IEEE Computational Science and Engineering*, *4*(2), 27–38. https://doi.org/10.1109/99.609829

Izu, C. (2021, March 3). Exploring the inchworm problem’s ability to measure basic CS skills. *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. http://dx.doi.org/10.1145/3408877.3432367

Miller, G. (2006). A scientist's nightmare: Software problem leads to five retractions. *Science*, *314*(5807), 1856–1857. https://doi.org/10.1126/science.314.5807.1856

Pea, R. D. (1986). Language-Independent conceptual "bugs" in novice programming. *Journal of Educational Computing Research*, *2*(1), 25–36. https://doi.org/10.2190/689t-1r2a-x4w4-29j2

Qian, Y., & Lehman, J. (2017). Students' misconceptions and other difficulties in introductory programming. *ACM Transactions on Computing Education*, *18*(1), 1–24. https://doi.org/10.1145/3077618

Rafalski, T., Uesbeck, P. M., Panks-Meloney, C., Daleiden, P., Allee, W., Mcnamara, A., & Stefik, A. (2019, July 30). A randomized controlled trial on the wild wild west of scientific computing with student learners. *Proceedings of the 2019 ACM Conference on International Computing Education Research*. http://dx.doi.org/10.1145/3291279.3339421

Sanders, R., & Kelly, D. (2008). Dealing with risk in scientific software development. *IEEE Software*, *25*(4), 21–28. https://doi.org/10.1109/ms.2008.84

Trisovic, A., Lau, M. K., Pasquier, T., & Crosas, M. (2021, March 23). *A large-scale study on research code quality and execution*. ArXiv.Org. https://arxiv.org/abs/2103.12793