

MINIMIZE PAIN POINTS IN RESEARCH
SOFTWARE DEVELOPMENT

A COLLABORATIVE FRAMEWORK TOWARD MINIMIZING
PAIN POINTS IN RESEARCH SOFTWARE DEVELOPMENT

BY
ANGE (PHIL) DU, MSME

A THESIS
SUBMITTED TO THE COMPUTING AND SOFTWARE
AND THE SCHOOL OF GRADUATE STUDIES
OF MCMASTER UNIVERSITY
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTERS OF SCIENCE

© Copyright by Ange (Phil) Du, April 2025

All Rights Reserved

Masters of Science (2025)
(Computing and Software)

McMaster University
Hamilton, Ontario, Canada

TITLE: A Collaborative Framework Toward Minimizing Pain
Points in Research Software Development

AUTHOR: Ange (Phil) Du
MSME (Mechanical Engineering),
Embry-Riddle Aeronautical University,

SUPERVISOR: Dr. Spencer Smith

NUMBER OF PAGES: [xvii](#), [110](#)

Lay Abstract

Modern scientific and engineering research relies heavily on custom software for numerical analysis, yet the development of this class of software is often ad hoc for small projects. Domain experts and partnering developers frequently lack shared expertise. While existing solutions often assume large teams, this thesis proposes a practical framework enabling small domain expert-developer teams to build robust, sustainable research software.

Supported by a GitHub template, our approach centers on structured requirements elicitation, where developers guide domain experts through key questions about theory, use cases, computational scale, and testing. Answers directly shape design, verification, and documentation. Key recommendations include: (i) theory documentation to ease development, (ii) early Continuous Integration adoption, (iii) low-level design via code-embedded comments, and (iv) performance-aware modularization.

This framework elevates small research software projects from disposable tools to sustainable assets. The thesis reviews prior work, analyzes case studies, details our practices, and proposes experimental evaluations.

Abstract

Modern scientific and engineering research increasingly relies on software for data processing, analysis, and simulation. However, research software is often developed ad hoc, with limited regard for sustainability or reproducibility, as researchers (domain experts) untrained in software engineering practices and developers new to the domain theories must both tread unfamiliar waters. While existing solutions (e.g., documentation template, software life-cycle, CI/CD, formal methods) aim to bridge this gap, they often assume large, specialized teams, leaving small research groups underserved.

Building upon our experience from research software projects, this thesis proposes a practical framework to empower small teams of domain experts and developers—particularly those transitioning from scientific or engineering backgrounds—to collaboratively build robust, sustainable research software. The framework addresses common pain points, such as evolving requirements and researchers’ limited technical familiarity, while fostering practices that benefit both immediate project needs and long-term maintainability. Central to the approach is structured requirements elicitation, where developers guide domain experts through targeted questions about theories, typical uses cases, computational problem scale and possible tests. The answers directly inform modular design, verification, and documentation, all supported

by our GitHub template for a seamless development process.

Key recommended practices include: (i) theory should be documented with structures and notations that ease its transition into code, (ii) early introduction of continuous integration, (iii) low-level design documentation via code-embedded comments (e.g. docstrings), and (iv) performance-aware modularization suitable for the problem scale.

By prioritizing clarity, flexibility, and developer-domain expert collaboration, this work aims to elevate research software from disposable tools to sustainable, peer-review-ready assets. The thesis reviews past studies on research software, examines our own software projects, details our methodology, and proposes a preliminary experiment design that would serve as a means to the proposed process and techniques.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Dr. Spencer Smith. You always have great ideas and advices when I am unsure about the directions in my thesis and development work. The academic resources you provided me with made the whole process much smoother. This was not my original degree plan, but your efforts to make it all work out for me are greatly appreciated. And I am glad you encouraged me to submit my work to JOSS and the ISS conference. These are things I would never have done myself.

I also appreciate the feedback from my committee members, Dr. Ned Nedialkov and Dr. Richard Paige (and the much-needed funding from Dr. Paige as well!). Your questions and assessments were fair and helped me clarify my ideas and arguments.

I enjoyed working with Nikita Holyev on the SynthEddy project, our interactions and challenges faced together sparked many ideas that I would not have thought of otherwise. This gratitude also extends to his supervisor, Dr. Marilyn Lightstone and Dr. Stephen Tullis, for your great suggestions in the software revisions.

A big thank you to Cynthia Liu for reviewing my CAS 741 documents. It was probably not the easiest thing to go through, but your feedback was very helpful and provided much-needed alternative perspectives.

You might never be reading this, but cheers to my flight-sim buddies, friends at

GTRC and my viewers. You have been my stable source of joy and fulfillment over the years no matter where I am in the world. Let's go catch that 3 wire!

I would like to thank my family, my parents for always believing in me even when I was not sure about myself, and supporting me in my decisions. I only wish I could have spent more time with all of you.

Finally, I would like to thank those around me. I am aware that I am not the easiest person to deal with. Thank you so much for putting up with all my quirks and nonsense.

Contents

| | |
|--|-----------|
| Lay Abstract | iii |
| Abstract | iv |
| Acknowledgements | vi |
| Notation, Definitions, and Abbreviations | xv |
| Declaration of Academic Achievement | xvii |
| 1 Introduction | 1 |
| 1.1 Purpose and Motivation | 2 |
| 1.2 Intended Audience | 3 |
| 1.3 Scope | 5 |
| 1.4 Literature Review | 6 |
| 1.5 Methodology | 12 |
| 1.6 Roadmap | 15 |
| 2 Background | 17 |
| 2.1 6DOF Trajectory | 17 |

| | | |
|----------|--|------------|
| 2.2 | SynthEddy | 23 |
| 3 | Proposed Practices | 32 |
| 3.1 | Overall Considerations | 33 |
| 3.2 | Information Gathering | 37 |
| 3.3 | Consolidating Background | 56 |
| 3.4 | Software Requirements | 58 |
| 3.5 | Testing | 70 |
| 3.6 | Design and Implementation | 74 |
| 3.7 | User Guide and “Advertising” | 79 |
| 4 | Preliminary Experiment Design | 84 |
| 4.1 | Overview | 84 |
| 4.2 | Built-in Evaluation | 85 |
| 4.3 | Case Study and Focus Group | 90 |
| 4.4 | Comparison to Other Projects | 99 |
| 5 | Conclusion | 101 |
| 5.1 | Potential Adoption | 103 |
| 5.2 | Future Work | 104 |
| | Bibliography | 106 |

List of Figures

| | | |
|-----|---|----|
| 1.1 | 2009 survey result showing understanding vs. perceived importance of software engineering concepts by researchers [1] | 7 |
| 1.2 | A typical SRS table of content from [2] | 11 |
| 1.3 | Example of a quick issue creation button present on each section/subsection of the documents in our template, with generated issue linking back to this exact location. | 15 |
| 2.1 | Plot produced by the 6DOF trajectory simulator of projectile tip orientation, showing its self-stabilizing behavior after initial disturbance | 19 |
| 2.2 | Velocity magnitude cross-section plot of a full turbulent velocity field with zero mean velocity generated by SynthEddy. | 24 |
| 2.3 | Velocity magnitude plots showing the top and side view of a single eddy (left and center) and the wrap-around of an eddy (right) when it reaches the edge of the flow field to ensure conservation of mass. . . . | 25 |
| 2.4 | Poster of SynthEddy as presented as McMaster CAS Poster Competition | 26 |
| 2.5 | The instance model used in SynthEddy as documented in its Software Requirements Specification (SRS) | 28 |

| | | |
|-----|---|----|
| 2.6 | Velocity magnitude cross-section plot of a non-uniform mean velocity field generated by the updated SynthEddy. This field mimics a channel or pipe flow, with a parabolic velocity profile that has a zero mean velocity on the edges and a maximum velocity at the center. | 29 |
| 2.7 | A screenshot during early development of SynthEddy, showing it requesting an unreasonably large amount of memory when hit with the actual problem scale suitable for publication. | 30 |
| 3.1 | The original, full V-model which the current work builds upon, displayed in the paper [3] | 37 |
| 3.2 | Our V-model workflow, with numbering indicating suggested chronological order of activities. Bold frame items are the main focus of this chapter. Green (Information Gathering) is newly introduced in this thesis. Blue items are where we will primarily cut-down development overhead. | 38 |
| 3.3 | An excerpt of the meeting agenda page from the GitHub template. | 40 |
| 3.4 | Typical parallel and sequential relationships between equations in a model | 46 |
| 3.5 | A plot from the 6DOF Trajectory paper [4] showing the interpolation of normal force (lift) coefficients at different mach number to be used in the program, which is non-trivial if the user wants to use a different projectile type without appropriate data. | 51 |

| | | |
|------|--|----|
| 3.6 | An excerpt of a filled-in meeting note using the GitHub template. In this case the domain expert may want to create an issue regarding the “additional data” question, as the developer did not write down an exhaustive list. | 53 |
| 3.7 | Structure of the requirements sections within all development documents in the GitHub template | 58 |
| 3.8 | An example input value table in our GitHub template. Note how we include links on non-trivial default values (drag coefficient in this case) to further explanations | 59 |
| 3.9 | Default eddy profile provided by the domain expert to be used in SynthEddy. Without proper traceability, these are just opaque numbers. | 60 |
| 3.10 | An example of theoretical model in our GitHub template. | 64 |
| 3.11 | An example of derived instance model in our GitHub template. | 65 |
| 3.12 | The eventual chosen form of the theoretical model documented in the SynthEddy project. | 68 |
| 3.13 | The Design Decisions section in our GitHub template, with hints for the developers. | 73 |
| 3.14 | Instructions and further resources on how to set up GitHub Actions for continuous integration in our template. | 74 |
| 3.15 | An example module as seen in the SynthEddy Module Guide. | 75 |
| 3.16 | A detailed docstring example in SynthEddy. | 77 |
| 3.17 | The use hierarchy showing the original modular design of SynthEddy, with each “eddy” being an object use by the “flow field”. | 78 |
| 3.18 | An example function with detailed explanation in SynthEddy. | 80 |

| | | |
|------|---|----|
| 3.19 | An excerpt of the installation guide of SynthEddy. | 82 |
| 4.1 | Example Built-in Evaluation page at the end of testing documents in the GitHub template. | 86 |

List of Tables

| | | |
|-----|---|----|
| 2.1 | Simulation result summary in the form of a range card, showing the trajectory of a projectile with a 6DOF model. | 18 |
| 2.2 | Aerodynamic coefficients used by simple, 6DOF and Modified Point Mass (MPM) models in the 6DOF trajectory simulator | 21 |
| 2.3 | The list of user inputs in the 6DOF trajectory simulator | 22 |
| 3.1 | Traceability between meeting questions and document sections. . . . | 55 |

Notation, Definitions, and Abbreviations

Notation

\mathbf{v} A vector represented by a bold symbol

m A scalar

Abbreviations

6DOF Six-Degree-of-Freedom

BC Boundary Condition

CFD Computational Fluid Dynamics

CI Continuous Integration

DE Domain Expert

DD Data Definitions

| | |
|------------|-------------------------------------|
| DNS | Direct Numerical Simulation |
| IC | Initial Condition |
| IDE | Integrated Development Environment |
| IM | Instance Model |
| LES | Large-Eddy Simulation |
| MG | Module Guide |
| MIS | Module Interface Specification |
| MVP | Minimum Viable Product |
| NFR | Non-Functional Requirement |
| ODE | Ordinary Differential Equation |
| SRS | Software Requirements Specification |
| TM | Theoretical Model |

Declaration of Academic Achievement

The work of the software SynthEddy was started in CAS 741, but completed as a product that support more general use cases as part of the M.Sc. research. The process to develop such software is a major case study that contributes this thesis. The 6DOF trajectory simulation work, which is used as an example in this thesis, was completed before the current M.Sc. as part of a prior degree.

A [GitHub repository template](#) that consolidates the best practices of this thesis for direct usage is another artifact that is a part of this thesis.

Chapter 1

Introduction

In today's world, scientific and engineering research rarely stays on just pen and paper. Software is often a crucial part of the research process. It is often used to collect, process, analyze and visualize large amounts of data, or to simulate physical systems that are otherwise difficult study in real life [5].

Due to the nature of research, it is not uncommon for the required software to not exist in the first place; the software will need to be developed or adapted to suit the specific needs of the research. This either pushes the researchers themselves into the unfamiliar territory of software development, or requires collaborations between software developers and researchers [1] [6]. Going forward, we will also refer to the researchers as *domain experts*.

While some may view the software as just a temporary tool to get the research done and to get papers published, we disagree. Scholarly research requires that the research work can withstand the scrutiny of peer review and reproducibility; therefore, it is only logical to extend this expectation to the research software as well.

This thesis aims to provide a guideline for research software development, with a

focus on the developers’ point of view to efficiently collaborate with domain experts and produce high quality, reproducible software.

1.1 Purpose and Motivation

The main driving force behind this thesis is to prompt well-documented, well-executed software development in the field of scientific research. The hope is to move research software away from being a one-off tool for a single paper/project, to a sustainable product that can be reused, extended, and maintained by others in the future, which would benefit not just the original research team, but the research community as a whole.

As we will discuss in Literature review (1.4) and Background (2), there are a number of challenges or pain points that lie between us and our vision. These include technical challenges, such as researcher not familiar with tools like Git or concepts like testing and continuous integration, as well as broader issues not strictly related to programming itself, such as researchers’ time constraints and certain information not being available early when drawing up initial software requirements.

Our proposed practices, in the form of a development framework (a GitHub template as an example), aim to help developers who look to participate in a research project navigate around these challenges. Key strategies include gathering as much gathering certain information immediately relevant to the development early on, allowing for flexibility in requirements and design to accommodate the evolving nature of research, etc. Since there are currently existing practices that offer strong theory-to-code traceability (as discussed in Section 1.4.2), we strive to maintain the same level of traceability but with less overhead and a more approachable process. We hope

that by following this guideline, the developers can produce high quality software that can be easily understood and maintained by others, and that the researchers/domain experts can focus on their research without having to worry about the software side.

1.2 Intended Audience

The work in thesis is made for developers of research software. More specifically, those who did not come from a computer science or software engineering background, but are looking to acquire the relevant knowledge. This can be the researchers themselves on an ad-hoc basis. But additionally, we want to harness the talent of a specific group of people: those who started from an engineering or scientific background, but are looking to transition into software development, either through graduate studies, switching majors or even after years of working in the industry. We believe their ability to speak the language of both the domain experts and the software developers can be a great asset in producing high quality research software.

While the direct goal of this thesis is to increase research software project productivity, it also carries the potential bonus of aiding personal growth. Many aspects of the discussion here, such as requirements gathering, continuous integration and project management, are also applicable to software development in a broader context, regardless of whether our audience decide to stay purely in research software development or move to other software engineering roles.

Before diving deeper, we first need to define the roles being discussed in this thesis.

1.2.1 Developers

The developers are the people who write the code, the associated documents and tests. They are responsible for the design and implementation of the software, and the intended audience for this work as mentioned above. They can be:

- In a one-person project, a researcher aspiring to develop high quality software, also known as an “end-user developer” [7].
- A developer attached to the research project. This can be a student or someone hired specifically to work on the software.
- A member of a research team who steps up to take on the software development responsibility.

1.2.2 Domain Experts

The domain experts are the people who focus on the research itself. They are the clients of the developers, and are the first users of the software. Other than the one-person team scenario, we have to make the following assumptions:

- They may have limited programming experience, if any. They likely do not know anything more sophisticated than writing simple MATLAB scripts.
- They are not familiar with software development practices, such as version control, issue tracking, testing, or documentation.
- Other than strictly writing codes, they have little experience with software development tools and infrastructure, such as GitHub, package and environment managers, or Linux systems.

- While they have a good understanding of the research problem, they may not have a clear idea of what the software should do or look like.
- They may not have a clear idea of the software development process, such as how long it should take, the milestones, or the resource requirements, potential limitations or even risks.

1.3 Scope

In this thesis we define software custom-built to advance certain research as “research software”, which may range from full-fledged computer programs to simple scripts. This does not include off-the-shelf software used in the research process, such as Ansys Fluent for fluid simulation, SOLIDWORKS for mechanical design, etc. Our scope does, however, include programs, add-ons or adaptations made to work in conjunction with such off-the-shelf software, such as a custom tool to analyze a standard output for a specific research requirement.

This thesis provides a guideline for the developers to efficiently collaborate with domain experts and produce software that both stratifies the immediate research needs, and is sustainable that future potential users can understand, adapt and maintain. The scope of this thesis is limited to the development of research software. It does not cover the research itself, the domain knowledge, or the research methodology.

While certain parts of this thesis may be applicable to very large scale problem that involve a sizable research team, the focus is on small to medium size projects (or a small slice of a larger project) where the developer is in direct contact with one or a few domain experts. To be more specific, the immediate result from the domain

experts utilizing the software built with this guideline is expected to be a single paper or a small set of papers (with possible follow-up research and development), rather than a large scale project that may span multiple years and involve multiple research groups.

Aspects and best practices for software development in general are also not the focus. They will only be discussed in the context of research software development and with further resources provided for those who wish to explore them further.

1.4 Literature Review

In this section, we will review some of the pain points identified in other studies that we aim to solve or avoid, and what is currently considered to be the best practices in research software development.

1.4.1 Pain Points

There had been a number of previous studies that have tried to identify the pain points in research software development, and the probable reasons behind those pain points. Some of the previous studies have focused on researchers themselves developing software, while others looked at (possible) larger projects where collaborations between domain experts and software developers are involved.

The issue that there is a widening gap of information and lack of communication between the scientific community and general software community has been noticed for quite some time [5]. This was the driving force behind a survey conducted back in 2009 with almost 2000 participants from the scientific community [1]. This study

and its replication in 2018 [8] focus on the first scenario where researchers themselves develop software.

Perhaps unsurprisingly, the lack of software engineering knowledge was identified as one of the main pain points [1] [8]. While most participants identified the importance of good software in research, and acquired some software development skills through mostly informal means like self-study or peers, their point of view of such “importance” likely differs from that of a professional software developer [1]. During research, these scientists were also the primary user of the software. The priority was to make the software work for their research. This was somewhat reflected by the fact that the participants did not feel that software maintenance and product/project management were that important; the main focus was on building the software itself [1]. One other interesting result found by the 2009 research [1] was that researchers generally concur with the value of testing, but unfortunately, also felt their lack of knowledge for doing so.

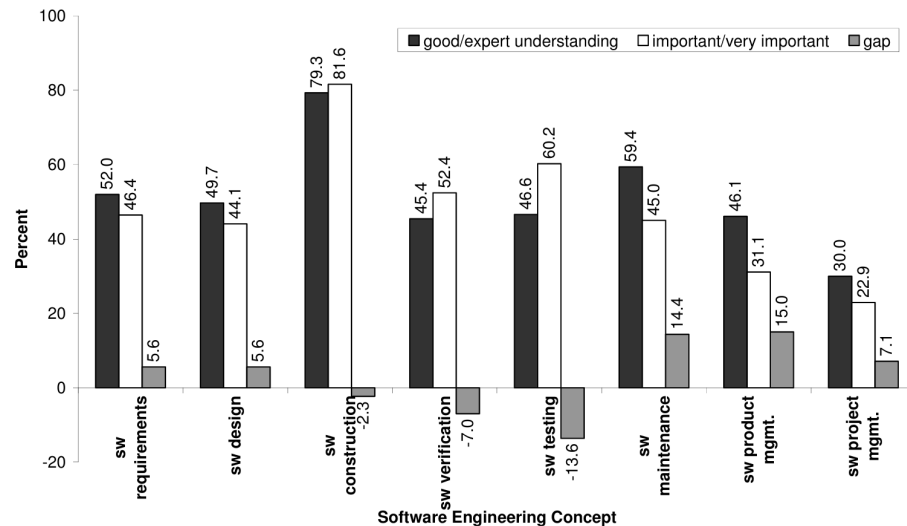


Figure 1.1: 2009 survey result showing understanding vs. perceived importance of software engineering concepts by researchers [1]

This is not helped by the fact that, research software, by definition, is used to “explore strange new worlds”. The requirements are often not clear and possibly keep changing as the research progresses, with the lack a ground truth or “oracle” for software testing [9].

The ever-changing research landscape can also be a contributing factor to “scope bloat” or “feature creep” when goals and requirements continuous to expand as research progresses, an issue identified by multiple papers [8][9][6].

Poor documentation was also identified as a major pain point [8]. However, besides simply pointing fingers to “fix your docs” and provide maintenance, we should recognize the likely underlying reason behind all this: the lack of proper reward, academically and psychologically. The survey participants felt that, other than themselves, there were few end users that would ultimately use their software and provide enough feedback [8]. And even in the case where their software was used, they did not feel that they are given the same level of credit or recognition as they would be for a paper [8][10]. Since they did not have enough dedicated time to work on the software anyway [8], it was perhaps not surprising that they would let requirements, design and maintainability slide. There have been calls for more recognition of software as part of the research output [11], but it certainly would take some time before it ever becomes a norm.

While a dedicated software engineer may address some of the issue above, introducing such member to the team is also not a silver bullet, and can even potentially cause more issues. For example, a developer may expect well-defined software requirements upfront, which is not always possible in a research setting, leading to a communication breakdown [9]. There are inherent differences between developing

software for research purpose and traditional software such as websites, accounting tools, games, etc. For the most part, developers would at least have some understanding of how a type of software should work on the surface level [9]. This is also eased by the fact that most clients of traditional software can give fairly clear requirements or descriptions of their workloads [9]. But the same cannot be said when dealing with research software where the field is highly specialized. One simply cannot expect a developer to understand whatever biochemical, fluid dynamics, or quantum mechanics problem that the domain expert bring to them, especially when the domain experts (researchers) themselves may still be figuring out new ideas at the same time [9]. Such a knowledge gap can happen in both ways. When working in a collaborative setting, even with a dedicated developer, the domain expert may lack some software know-how that is common in the software community, which the developer can no longer take for granted. As a result, communication between the developer and the domain expert can be slow and daunting, especially when they are not speaking the same language initially [12].

For the developers, these challenges may lead to mean special requirements just to make the software understandable for the domain experts, suitable documentations for the audience [10], and potentially additional responsibilities that is outside the traditional software development role, such as IT support [6].

An overarching idea attempting to address these pain points is the FAIR principals (Findable, Accessible, Interoperable and Reusable) [13], which our work would also contribute to. It was also recognized that while there are certainly many tools and workflow for software development, they may also come with high barrier to entry and limited guidance that can be ill-suited for research software [13].

1.4.2 Current Practices

The inspiration and jumping off point for this thesis is the document-driven approach proposed by Smith et al. [14]. It has since been further fleshed out into a full template workflow for designing scientific software [2]. This template has been used by a number of scientific computing projects, such as a Bridge Chloride Exposure Predictor [15] to investigate bridge corrosion in Ontario, and a turbulent flow field generator which we will discuss in more detail in the next chapter (Section 2.2). It has also been used in a software engineering capstone course at McMaster University, and enabled the analysis of team behaviors in software projects [3].

This template is geared towards specialized fields, which can require a substantial amount of domain knowledge. It has a strong emphasis on documented traceability from theory to design decisions to final implementation.

As a document driven framework, it requires the developers to produce several artifacts before any code is written. In its suggested order, these include:

- Problem Statement
- Software Requirements Specification (SRS)
- Verification and Validation Plan (V&V Plan)
- Design documents
 - Module Guide (MG)
 - Module Interface Specification (MIS)
- Verification and Validation Report (V&V Report)

| | | |
|----------|--|-----------|
| 1 | Reference Material | 1 |
| 1.1 | Table of Units | 1 |
| 1.2 | Table of Symbols | 2 |
| 1.3 | Abbreviations and Acronyms | 4 |
| 2 | Introduction | 4 |
| 2.1 | Purpose of Document | 5 |
| 2.2 | Scope of Requirements | 5 |
| 2.3 | Organization of Document | 5 |
| 3 | General System Description | 5 |
| 3.1 | User Characteristics | 6 |
| 3.2 | System Constraints | 6 |
| 4 | Specific System Description | 6 |
| 4.1 | Problem Description | 6 |
| 4.1.1 | Terminology and Definitions | 6 |
| 4.1.2 | Physical System Description | 7 |
| 4.1.3 | Goal Statements | 7 |
| 4.2 | Solution Characteristics Specification | 8 |
| 4.2.1 | Assumptions | 8 |
| 4.2.2 | Theoretical Models | 9 |
| 4.2.3 | General Definitions | 11 |
| 4.2.4 | Data Definitions | 13 |
| 4.2.5 | Instance Models | 15 |
| 4.2.6 | Data Constraints | 21 |
| 5 | Requirements | 23 |
| 5.1 | Functional Requirements | 23 |
| 5.2 | Nonfunctional Requirements | 24 |
| 6 | Likely Changes | 25 |

Figure 1.2: A typical SRS table of content from [2]

Among these, the SRS is arguably the one that would receive the most attention (shown in Figure 1.2), as it documents the physical system and the refinement process from theory to final instance model that will be implemented in the software. The SRS will be used to guide the development throughout the project.

This is inline with our expectation that the software contributing to a research study should be able to withstand the same level of scrutiny, for which any piece of software developed with this workflow would have the required information very

clearly laid out. Its somewhat rigid structure however does risk colliding with the aforementioned pain point that research can be evolving [8], so some information needed to build these documents may be hard to pin down early on. We will discuss our experience with this workflow and how we propose to address these issues in the next section.

Perhaps the envisioned holy grail of such strongly traceable approach is the Drasil Project [16] It aims to document theoretical knowledge from the most basic physical laws (e.g. Newton’s laws), and refine them all the way to the eventual, domain specific instance model that will be implemented in the software. This way, both the software code (in various languages) and the accompanying documentation can be generated from the encoded knowledge, as opposed to being written by hand. The hope is that with each subsequent project utilizing Drasil, the knowledge base grows, so that future projects can keep building on top of the existing knowledge, and potentially require less and less effort. Although this is a noble goal, Drasil has not yet captured enough scientific knowledge to be practically useful. Therefore, the work presented in this thesis will take a practical approach that can be used immediately.

1.5 Methodology

While a document driven approach is beneficial for research software development that requires deep domain knowledge [2], it is not without its own set of obstacles when used in practice. In our experience, we found that it can be challenging to produce good initial documentations (such as the SRS in [2]) when the research is still evolving. This is echoed by many previous studies that identified the ever-changing nature of research as a pain point [8][9][6]. In such cases, a highly structured document can be

as much of a bane as it is a boon. A rebuttal to the opponent of the document driven approach [2] called for “faking” a rational design process to maintain structure. But in reality, the execution of this strategy is up for interpretation, and could be a source of frustration for developers. We found ourselves often trying hard to fit the research into the document and spending more time completing what the framework required just for the sake of completion. Although the complete information is necessary for a generative framework like Drasil, for a practical software developer, this is time taken away from development. More importantly, for the domain experts (especially when they are also the developer), they will very much justifiably feel that this is not making progress on the research itself, as pointed out previously [8]. All these combined can perhaps explain why the document driven framework first proposed in 2007 [14] has not seen an even wider adoption.

To address these pain points, we propose a more pragmatic approach in this thesis, a “document-lite” workflow if you will. The goal is to provide structure in research software development, while minimizing overhead and allowing for flexibility when the research is still evolving. We aim to take as little time away as possible from: 1. the domain experts, so they can focus on their research, and 2. the developers, so they can focus on the software development. This involves, early on, asking only the questions most relevant to creating the initial software (the Minimum Viable Product, MVP). Instead of completing every aspect of the initial design documents or “faking it”, we accept “I don’t know” as an answer, and circle back if it can be answered when works progresses.

One caveat of our envisioned template is that it may capture less domain information compared to the path that would lead to things like Drasil. Since we only require

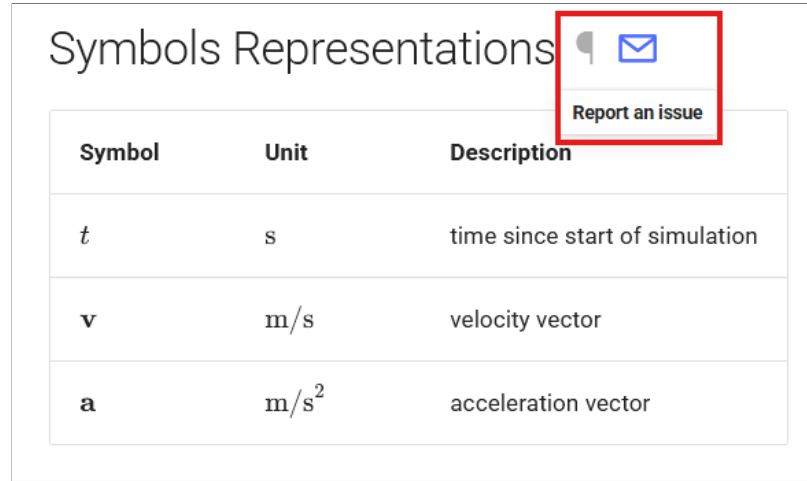
the most immediate domain knowledge to build the software, we lose the chance to document the refinement process from basic physical laws all the way to the final instance model. However, we believe that this is a fair trade-off for as this process is probably only feasible for someone with a deep understanding of the domain, which we do not necessarily require from our developers.

1.5.1 GitHub Template

One important artifact of this thesis is a document driven GitHub repository template that helps the developer to follow the practices we suggest, and facilitate the interaction between the developer and the domain expert. This template and accompanying workflows are inspired by the capstone project template by Dr. Spencer Smith [17], augmented with the lessons learned from when developing the projects described in the Background chapter (Section 2) and communication with the domain experts. While the focus of our template is on research software development, it is possible to adapt it to a more general use case.

Compared to the previous template, this new template cuts down some overhead (in terms of writing time) and focus more on the interaction between the developer and the domain expert.

The entire documentation suite, including the development documents, user guides and any notes are written in Markdown instead of LaTeX, which was used previously. These are presented as a website using GitHub Pages and MkDocs [18, 19], with Continuous Integration (CI) workflow already setup using GitHub Actions to automatically build and deploy the website when the documents are updated. Unlike monolithic LaTeX PDF documents, the documents on the website are broken down



The screenshot shows a document section titled "Symbols Representations". Below the title is a table with three columns: "Symbol", "Unit", and "Description". The table contains three rows of data. To the right of the table, there is a "Report an issue" button with a mail icon, which is highlighted by a red rectangular box.

| Symbol | Unit | Description |
|--------------|------------------|--------------------------------|
| t | s | time since start of simulation |
| \mathbf{v} | m/s | velocity vector |
| \mathbf{a} | m/s ² | acceleration vector |

Figure 1.3: Example of a quick issue creation button present on each section/subsection of the documents in our template, with generated issue linking back to this exact location.

into smaller, more manageable pages, and can be reviewed independently before a whole document is finished. It provides better guidance to the domain experts evaluating the documents, as they can easily navigate to different sections. The writer (developer) can also include hyperlinks to any part of various documents when necessary, which is not possible in the previous template. As many domain experts may not be familiar with many common GitHub functionalities, our template allows any viewer to easily create issues with one-click that link directly back to the relevant part of the document, which can then be addressed by the developer (Figure 1.3).

1.6 Roadmap

In this chapter, we have discussed the purpose of this thesis, the problems we aim to address, and a brief overview of our intended approach.

In the next chapter, **Background** (2), we will examine the software projects that

were most influential in shaping our approach. This includes overview of the projects and the lessons learned.

The **Practices** chapter (3) provide details on our proposed research software development framework and the reasoning to support it, as well as some suggestions given to the developers in different stages of the project. It is written in a chronological order representing a likely software development process.

Although we did not have time to fully evaluate our framework with real participating software projects, we do follow up with an **Experiment Design** chapter (4). It gives the hypotheses to be tested, and the experimental setup to both evaluate the effectiveness of our framework and to gather feedback for future improvements.

Finally, the **Conclusion** chapter (5) that summarizes the key points of this thesis, and discusses our vision for potential future adoptions of our framework, given the current academic landscape.

Chapter 2

Background

Besides the suggestions presented by works in the literature review (Section 1.4), the practices proposed in this thesis are largely based on lessons learned from several software projects that the author has either participated in or closely followed. These include a Six-Degree-of-Freedom (6DOF) Ballistic Trajectory Calculator, and SynthEddy, a turbulent flow field generator.

In this chapter, we will briefly introduce these projects, and examine the lessons learned from our experiences with them that can be generalized to other research software projects in later chapters.

2.1 6DOF Trajectory

This was a previous work by the author of this thesis, for his Master’s Degree in Mechanical Engineering. It was a typical “one-person team” scenario where the developer was the domain expert. The software work enabled the author’s previous

research on Six-Degree-of-Freedom Trajectory Simulation at Embry-Riddle Aeronautical University (ERAU) [4]. It was a MATLAB program that acts as a high-precision ballistic computer to simulate spinning projectiles in a 3D space. Example results are shown in Table 2.1 and Figure 2.1. It has since been used by a few other researchers at ERAU working on related topics.

| Range (m) | Drop (cm) | Drop (MIL) | Drop (MOA) | Windage (cm) | Windage (MIL) | Windage (MOA) | Velocity (m/s) | Mach | Time |
|--------------|--------------|---------------|---------------|-----------------|------------------|------------------|-------------------|------|------|
| 0 | -2.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 892.00 | 2.62 | 0.00 |
| 50 | 0.74 | 0.15 | 0.51 | -0.04 | -0.01 | -0.03 | 840.73 | 2.47 | 0.06 |
| 100 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 0.00 | 791.03 | 2.32 | 0.12 |
| 150 | -4.66 | -0.31 | -1.07 | 0.13 | 0.01 | 0.03 | 742.80 | 2.18 | 0.18 |
| 200 | -13.76 | -0.69 | -2.37 | 0.36 | 0.02 | 0.06 | 695.99 | 2.04 | 0.25 |
| 250 | -27.94 | -1.12 | -3.84 | 0.71 | 0.03 | 0.10 | 650.63 | 1.91 | 0.33 |
| 300 | -47.91 | -1.60 | -5.49 | 1.19 | 0.04 | 0.14 | 606.76 | 1.78 | 0.41 |
| 350 | -74.55 | -2.13 | -7.32 | 1.83 | 0.05 | 0.18 | 564.46 | 1.66 | 0.49 |
| 400 | -108.90 | -2.72 | -9.36 | 2.66 | 0.07 | 0.23 | 523.82 | 1.54 | 0.59 |
| 450 | -152.21 | -3.38 | -11.63 | 3.69 | 0.08 | 0.28 | 484.68 | 1.42 | 0.68 |
| 500 | -205.98 | -4.12 | -14.16 | 4.96 | 0.10 | 0.34 | 446.84 | 1.31 | 0.79 |
| 550 | -272.07 | -4.95 | -17.01 | 6.51 | 0.12 | 0.41 | 410.56 | 1.21 | 0.91 |
| 600 | -352.75 | -5.88 | -20.21 | 8.38 | 0.14 | 0.48 | 376.36 | 1.11 | 1.04 |
| 650 | -450.80 | -6.93 | -23.84 | 10.64 | 0.16 | 0.56 | 344.99 | 1.01 | 1.17 |
| 700 | -569.49 | -8.13 | -27.97 | 13.35 | 0.19 | 0.66 | 318.64 | 0.94 | 1.33 |
| 750 | -712.36 | -9.50 | -32.65 | 16.61 | 0.22 | 0.76 | 297.81 | 0.88 | 1.49 |
| 800 | -882.90 | -11.04 | -37.94 | 20.48 | 0.26 | 0.88 | 281.32 | 0.83 | 1.66 |
| 850 | -1084.48 | -12.76 | -43.86 | 25.06 | 0.29 | 1.01 | 267.93 | 0.79 | 1.84 |
| 900 | -1320.30 | -14.67 | -50.43 | 30.40 | 0.34 | 1.16 | 256.67 | 0.75 | 2.03 |
| 950 | -1593.46 | -16.77 | -57.66 | 36.56 | 0.38 | 1.32 | 246.87 | 0.73 | 2.23 |
| 1000 | -1907.04 | -19.07 | -65.55 | 43.61 | 0.44 | 1.50 | 238.05 | 0.70 | 2.44 |

Table 2.1: Simulation result summary in the form of a range card, showing the trajectory of a projectile with a 6DOF model.

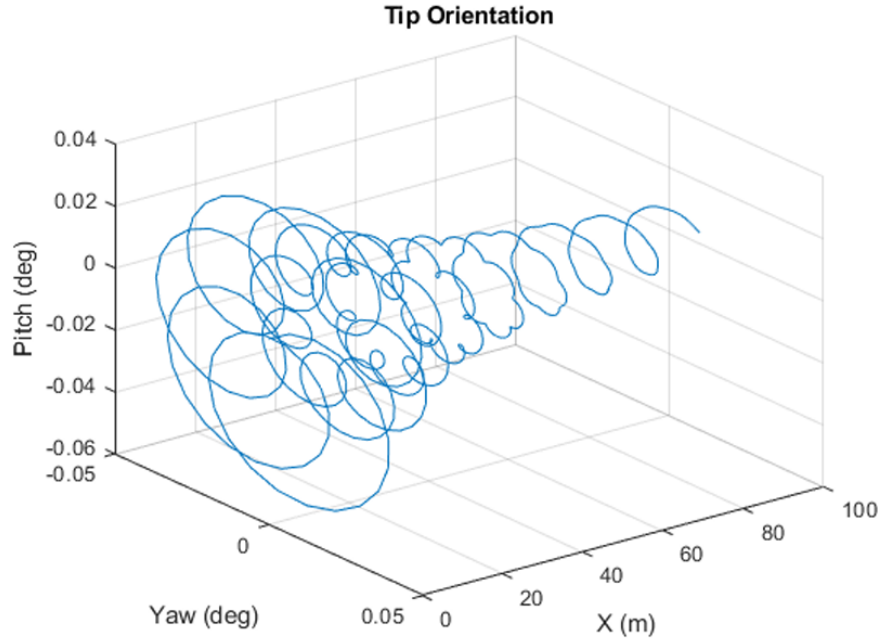


Figure 2.1: Plot produced by the 6DOF trajectory simulator of projectile tip orientation, showing its self-stabilizing behavior after initial disturbance

While the research itself was a success, the software, though functional, had left a lot to be desired. It was developed before the author had any formal knowledge of software development practices. Here we would use it mainly to contrast with the practices proposed in this thesis.

2.1.1 What Went Right

Despite many shortcomings, two positive takeaways still held from this project:

First is the validation approach, although the process is manual. The more complex 6DOF trajectory model (the governing differential equations of which is shown below) required a series of non-trivial aerodynamic coefficients for each of the force and moment components (Table 2.2) that were hard to obtain, and there was no

publically available implementation to test against. However, the author was able to standardize the inputs (Table 2.3) and pre-processing/post-processing steps across all possible models in the software. This allowed the software, while using such simpler models, to be validated against publically available ballistic calculators with the same underlying models.

$$\begin{aligned}\frac{d\mathbf{V}}{dt} &= \frac{\mathbf{F}_D + \mathbf{F}_L}{m} + \mathbf{g} + \mathbf{\Lambda} \\ \frac{d\mathbf{h}}{dt} &= \frac{\mathbf{M}_a + \mathbf{M}_{M_{p\alpha}} + \mathbf{M}_{I_p} + \mathbf{M}_{M_{q+\dot{\alpha}}}}{I_T}\end{aligned}$$

Terms in the equations:

- \mathbf{V} : Velocity vector of the projectile.
- \mathbf{F}_D : Drag force.
- \mathbf{F}_L : Lift force.
- $\mathbf{\Lambda}$: Coriolis Effect acceleration.
- \mathbf{h} : Angular momentum vector of the projectile.
- I_T : Total moment of inertia of the projectile.
- \mathbf{M}_a : Overturning moment.
- $\mathbf{M}_{M_{p\alpha}}$: Magnus moment.
- \mathbf{M}_{I_p} : Spin damping moment.
- $\mathbf{M}_{M_{q+\dot{\alpha}}}$: Pitch damping moment.

| Coefficient for | Variable Name | Used by | | |
|----------------------|---------------|---------|------|-----|
| | | Simple | 6DOF | MPM |
| Drag force* | CD0, CD2 | ✓ | ✓ | ✓ |
| Lift force* | CLa | | ✓ | ✓ |
| Overturning moment* | CMa0, CMa3 | | ✓ | ✓ |
| Magnus moment* | CMpa0, CMpa3 | | ✓ | |
| Spin damping moment | Clp | | ✓ | ✓ |
| Pitch damping moment | CMqPlusCMad | | ✓ | |

Table 2.2: Aerodynamic coefficients used by simple, 6DOF and Modified Point Mass (MPM) models in the 6DOF trajectory simulator

Second is the large number of well-documented inputs that the user can change to simulate a wide range of scenarios (Table 2.3). While these were much more than any commercial software may allow, and can be overwhelming to a new user at first, they did facilitate exploring many research questions in the problem domain without having to constantly modify the code.

| Category | Parameter | Variable | Unit | Note |
|------------------------------------|--|-------------|-------------------|---------------------------|
| Projectile (BLT) | Diameter | d | m | |
| | Length | L | m | |
| | Mass | m | kg | |
| | Axial moment of inertia | IP | kg-m ² | |
| | Transverse moment of inertia | It | kg-m ² | |
| | Muzzle velocity | MV | m/s | |
| | Barrel twist | twist | inch/rev | |
| | Various aerodynamic coefficients vs. Mach number data input as table | | | |
| Firing | Initial Altitude (ASL) | altitude | m | |
| Position | Initial Latitude | latitude | deg | (+) North |
| (POS) | Initial Azimuth | azimuth | deg | |
| Zeroing and Aiming (AIM) | Aiming line of sight | LOSAngle | deg | (+) uphill |
| | Scope height above barrel | ScopeHeight | m | |
| | Zero range of the scope | zeroRange | m | Provide one of the two |
| | Scope zeroing angle | scopeAngle | deg | |
| | Scope windage angle | windAngle | deg | windage zero |
| Atmospheric conditions (AIR) | Ambient air temperature | temp | °C | |
| | Relative humidity | humidity | % | |
| | Ambient air pressure | pressure | pa | |
| | Wind speed | windSpd | m/s | |
| | Wind direction | windDir | deg | |
| Computing | Simulation time span | tspan | s | |
| | Time step size | dt | s | |

Table 2.3: The list of user inputs in the 6DOF trajectory simulator

2.1.2 What Can Be Improved

The software was developed in a very ad-hoc manner, with no traceable requirements or design documents. While there were documentations explaining required user inputs, they were later found to be inadequate for other users to utilize the software independently. There was only one set of known-to-work inputs provided as an example, but not much detail regarding acceptable input ranges, which input should be modified for different scenarios, or what the user should do if certain inputs are hard to obtain (e.g. when they switch to a different projectile). This led to us suggesting more considerations for inputs when both gathering information (Section 3.2.5) and drawing up requirements (Section 3.4.1).

Another significant problem was that it was not version controlled with Git or similar tools. This had more than once led to awkward conversations between the author and other users seeking help, only for both parties to realize they were looking at different versions of the software. This had limited the potential users to only those who had direct contact with the author, as the software was not easily shareable.

2.2 SynthEddy

This was a more recent project that fell under the “developer & domain expert collaboration” setting, where the author served as the developer. The software [20] was a Python program to generate turbulent flow fields using the Synthetic Eddy Method [21] to be used as initial conditions (IC) and boundary conditions (BC) for computational fluid dynamics (CFD) simulations.

SynthEddy generates divergence free 3D velocity fields (Figure 2.2) consists of

many individual “eddies” (a circular flow, as shown in Figure 2.3), approximating real turbulent flows, without the need for a full physical simulation, which can be used to jump-start much more computationally expensive CFD simulations such as Large Eddy Simulation (LES) or Direct Numerical Simulation (DNS).

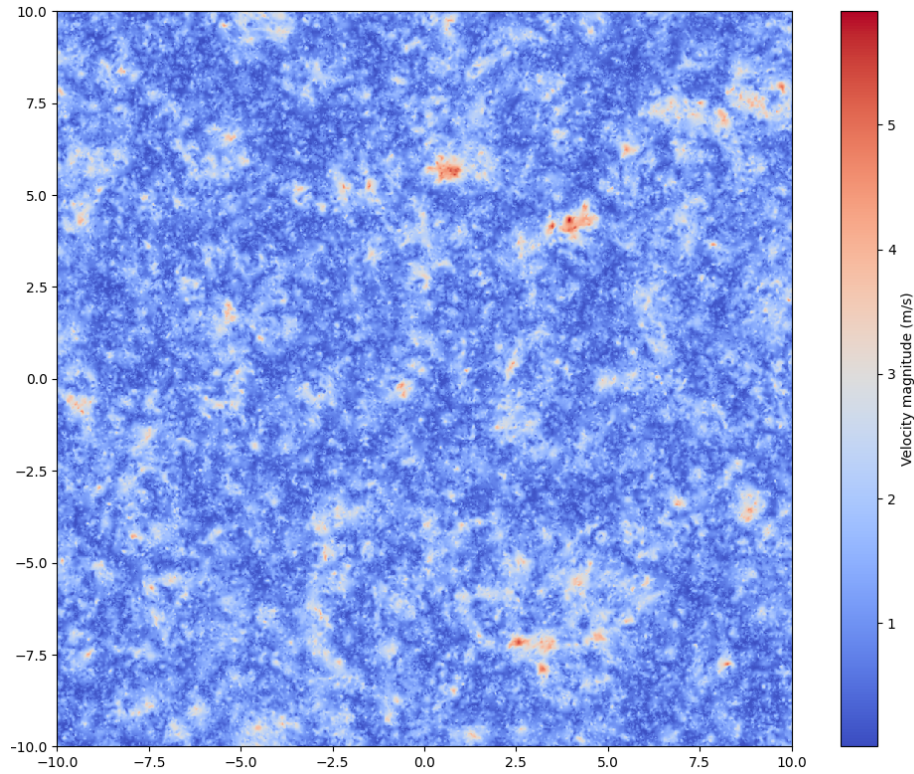


Figure 2.2: Velocity magnitude cross-section plot of a full turbulent velocity field with zero mean velocity generated by SynthEddy.

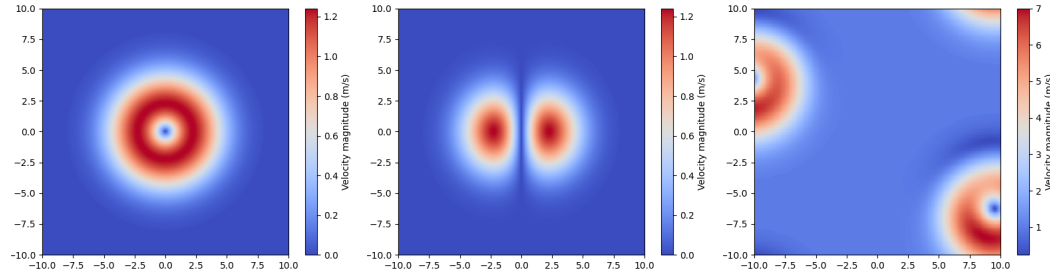


Figure 2.3: Velocity magnitude plots showing the top and side view of a single eddy (left and center) and the wrap-around of an eddy (right) when it reaches the edge of the flow field to ensure conservation of mass.

This project started during CAS 741 (Development of Scientific Computing Software), and was later further developed into a full-fledged product that supported more general use cases such as channel/pipe internal flow and boundary layer flow.

The domain expert and main user during its development phase was Nikita Holyev, a PhD student in Mechanical Engineering at McMaster University, whose research was on turbulent flow. This software has support at least one pending publication by Nikita Holyev.

SynthEddy itself was submitted to the Journal of Open Source Software (JOSS), pending review. It was also presented at the Seventh Computing and Software Poster and Demo Competition at McMaster University, where it won first place (Figure 2.4).

2.2.1 Highlights

Before diving into the development process and lessons learned, it is worth noting some unique aspects of SynthEddy that helps it stand out as a piece of research software.

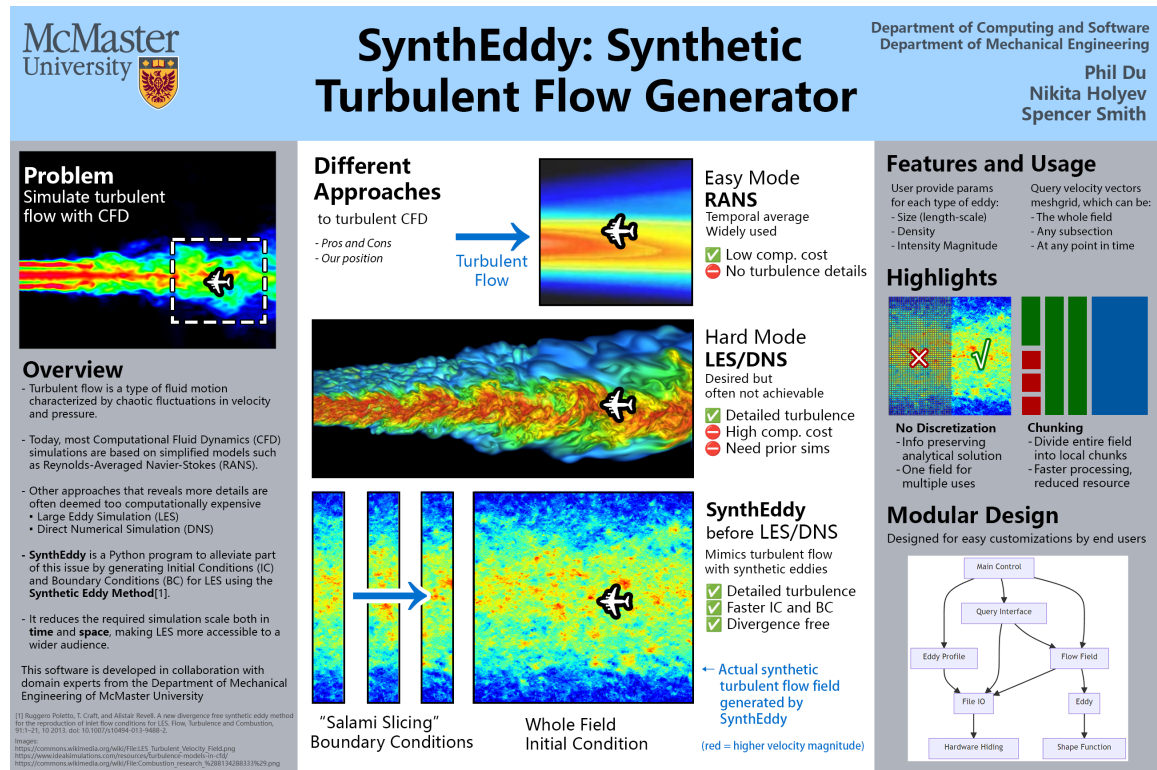


Figure 2.4: Poster of SynthEddy as presented as McMaster CAS Poster Competition

Information Preserving

A key design philosophy of SynthEddy was to preserve as much information as possible in the generated flow field. Since it is not a numerical simulation that solves the field like a CFD solver, we did not blindly follow what CFD solvers would do, i.e. discretizing the field at the beginning and updating it at each time step, which can lead to loss of information. Instead, the eddies are tracked as individual entities continuously moving in space and time, and the query process is independent of the field generation. This allows the user to query the same field multiple times with different parameters, such as having an overall coarse grid and a fine grid for a specific region of interest, or when performing grid sensitivity analysis. When queried

at any point in time, the program first finds the location of each eddy at such time analytically, without having to advance through prior time steps numerically.

This opens up the possibility for usage like “salami slicing” the field to obtain many thin strips of the field at any arbitrary rate demanded by the user, without concerning grid resolution or time step size. These can then be fed into a CFD solver as boundary conditions (see the bottom-center portion of Figure 2.4).

Chunking

The original theory of synthetic eddy method called for the sum of influence of all eddies for every point in the field, which would lead to a huge number of operations ($O(n^4)$ complexity for a 3D field) and unrealistic memory requirements. We recognized that the influence of each eddy is local, and thus designed the software to process the field in chunks. This drastically reduced resource requirements so that a 1000

2.2.2 Development Process

The software was developed using a document driven process with a focus on traceability from theory to requirements to final implementation. For example, Figure 2.5 shows a snapshot of its requirements document that includes an instance model of the software, which traceability is clearly presented. It represents a more formalized understanding of software development best practices by the author compared to previous works, with modularization, designed for change and continuous integration. Lessons learned from this project greatly contributed to this thesis.

| Number | IM1 |
|-------------|--|
| Label | Velocity at any position and time in the flow field |
| Input | $\bar{\mathbf{u}}$: Mean flow velocity, with zero y and z components (m/s) N : Number of eddies \mathbf{x}_0^k : Initial center position of each eddy (m) σ^k : Length-scale of each eddy (m) $\boldsymbol{\alpha}^k$: Intensity vector of each eddy (m/s) [DD4] \mathbf{x} : Any position in the flow field (m) t : Any time since start (s) |
| Output | \mathbf{u} , the velocity at any position and time |
| Equation | $\mathbf{u}(\mathbf{x}, t) = \bar{\mathbf{u}} + \mathbf{u}'(\mathbf{x}, t)$ where $\mathbf{u}'(\mathbf{x}, t) = \sum_{k=1}^N q_\sigma(d^k(\mathbf{x}, t)) \mathbf{r}^k(\mathbf{x}, t) \times \boldsymbol{\alpha}^k$ from [GD1] q_σ is any satisfying shape function as per [TM1] $\mathbf{r}^k(\mathbf{x}, t) = \frac{\mathbf{x} - \mathbf{x}^k(t)}{\sigma^k}$ from [DD1] $d^k(\mathbf{x}, t) = \mathbf{r}^k(\mathbf{x}, t) $ from [DD2] $\mathbf{x}^k(t) = \mathbf{x}_0^k + \bar{\mathbf{u}}t$ from [DD3] |
| Description | <p>The above model gives the velocity vector at any position within the flow field at any given time, by combining the influence from each eddy at such time as they move down the flow field with mean velocity.</p> <p>Such influence is calculated by the relative position of that point to the center of each eddy, according to its length-scale, intensity and shape function.</p> |
| Uses | GD1, TM1, DD1, DD2, DD3, DD4 |
| Ref. By | DD5 |

Figure 2.5: The instance model used in SynthEddy as documented in its Software Requirements Specification (SRS)

2.2.3 What Went Right

From a usability standpoint, the software was a success. With clear documentation and examples, the domain expert, who had no experience python environments or software development in general, was able to install and use the software both locally and on a computing cluster. He was able to run many different scenarios for his research need.

The software was developed with designed for change in mind. Initially it could only generate external flow with constant mean velocity across the entire field. However, when new requirements from the domain expert’s supervisor, Dr. Marilyn Lightstone and Dr. Stephen Tullis, called for any arbitrarily define flow (as shown in Figure 2.6), the majority of the change made was limited to just one module.

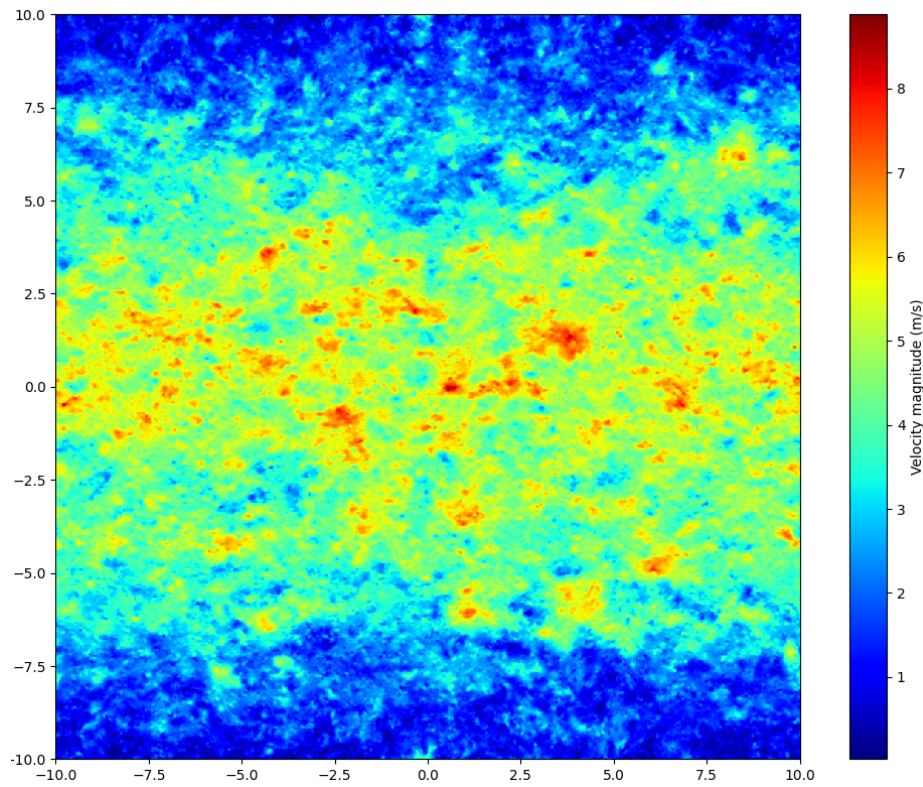
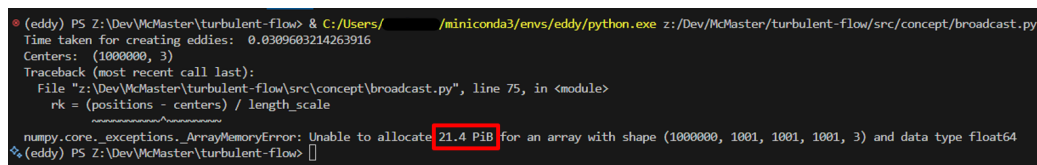


Figure 2.6: Velocity magnitude cross-section plot of a non-uniform mean velocity field generated by the updated SynthEddy. This field mimics a channel or pipe flow, with a parabolic velocity profile that has a zero mean velocity on the edges and a maximum velocity at the center.

2.2.4 What Can Be Improved

However, the initial rigid adherence to modularization and information hiding, when combined with a communication breakdown with the domain expert, led to one major redesign of the software early on. Since the initial presentation by the domain expert only showed a very limited number of eddies at play, the developer incorrectly presumed the scale of the problem and did not ask for further clarification on actual typical use cases. This resulted in the developer naively believe that he can capsulize all the eddies as objects of an eddy class, with self-contained states and methods. When the developer was finally aware of the actual output meshgrid resolution that is suitable for publication, the initial design turned out to be too slow and memory intensive to be practical (Figure 2.7). Part of the software had to be redesigned to put different eddy states as arrays in the parent module for broadcasting vectorized operations, which was much more efficient.



```
(eddy) PS Z:\Dev\McMaster\turbulent-flow> & C:/Users/ /miniconda3/envs/eddy/python.exe z:/Dev/McMaster/turbulent-flow/src/concept/broadcast.py
Time taken for creating eddies: 0.0309603214263916
Centers: (1000000, 3)
Traceback (most recent call last):
  File "z:\Dev\McMaster\turbulent-flow\src\concept\broadcast.py", line 75, in <module>
    rk = (positions - centers) / length_scale
          ~~~~~^~~~~~
numpy.core._exceptions._ArrayMemoryError: Unable to allocate 21.4 PiB for an array with shape (1000000, 1001, 1001, 1001, 3) and data type float64
(eddy) PS Z:\Dev\McMaster\turbulent-flow>
```

Figure 2.7: A screenshot during early development of SynthEddy, showing it requesting an unreasonably large amount of memory when hit with the actual problem scale suitable for publication.

The developer was also able to identify the local influence of the eddies and thus let the software process whole flow field space in chunks, saving memory and number of operations. However, it should be identified that this was possible partly due to the developer’s Mechanical Engineering background, which allowed him to better understand the underlying physics of the problem domain, which may not be generalizable

to other research software projects. As such, the actual best practice proposed in this thesis is to gather the correct information from the domain expert to begin with, and not fall into similar traps.

Another interesting aspect that may be changed if the project were to be redone is the choice of programming language. Since the influence of each eddy was local, the software would actually be very easy to parallelize with a language like C++. However, this discovery was made too late into the project that was already written in Python, which was chosen for its better support for vectorized operations using the NumPy library. Parallelization attempts in Python did not see significant speedup. Given the increased complexity, this branch was put on hold. This highlights the importance of understanding not just the theoretical model itself, but also how it would be invoked in a practice as early as possible in the project.-

Chapter 3

Proposed Practices

This chapter provides a wholistic view of our proposed workflow that we believe would be a best practice for developing small to medium scale research software, derived from the lessons learned from the software projects mentioned in the Background chapter (Chapter 2). It represents a typical development workflow from the perspective of a developer. Sections in this chapter are ordered chronologically on how each step would likely take place in a research software project. The life cycle model subsection provides a visualized overview (Figure 3.2) of the workflow and the focus of this chapter:

We start with the first technical meeting with the domain expert for **Information Gathering** (Section 3.2), providing specific guidelines on how to extract as much immediately relevant information as possible from the domain expert early on, and justifying the significance of various pieces of information.

This is followed by forming the **Software Requirements** (Section 3.4), which is adapted to the changing nature of research software, especially when a lot of the details need to be ironed out as the research progresses. A **Testing** (Section 3.5)

plan is produced next. We discuss how to circumvent the challenge of a lack of test cases, and better serve domain experts who may not be familiar with software testing. **Design and Implementation** (Section 3.6) is then the final piece of the puzzle. The above three sections map to the Requirements, Testing and Design documents, respectively. Readers will see that even in the context of a document driven workflow, these documents can still be produced in the most pragmatic way to minimize overhead without compromising traceability.

Aside from strictly talking about developing the software itself, we also include a **User Guide and “Advertising”** section (3.7) that focuses on the documentations facing all potential end users (not just the domain expert), and how they may also help to promote not just the software itself but also the domain expert’s research.

3.1 Overall Considerations

We will focus on the practices that are directly applicable to research software, while also providing some resources for best practices in software development in general. That being said, the writings in this thesis do not go into every single detail of the development process; it is limited to what we consider to be additions or modifications to a typical software development process. The responsibility of presenting the entire workflow end-to-end is left to the [GitHub template](#), which is included as an artifact of this thesis.

Before going into the following sections, we should reiterate the assumption that we are looking at a scenario with a developer working with a domain expert to develop a piece of software that represents and/or aids the domain expert’s research. More specifically, we assume that the domain expert has traits laid out in the introduction

chapter (Section 1.2.2). We will be discussing interactions, such as meetings between the developer and the domain expert.

The only technical expectation of the domain expert is that they will register for a GitHub account, so that they can post or respond to issues similar to on a forum. They are not expected to use other functionalities of GitHub, such as pull requests or branching, or any other software development tools. Note that they may learn these skills during the course of the project or over time, but these by no means should be a requirement for the project to proceed.

This may sound confusing and not applicable to scenarios where the developer themselves is the domain expert (How do I have a meeting with myself?). However, even in such cases, it is perhaps not the worst idea to have two imaginary roles for thinking, which offers different perspectives on the same problem. This can potentially lead to a more relatable way of thinking when it comes to serving any future user/researcher/developer who does not possess the full knowledge of both the research and the software.

3.1.1 Life Cycle Model

Choosing a life cycle model is a critical decision in software development. However, when working with a research domain expert who does not yet appreciate the importance of proper software development practices, this can potentially lead to some pushbacks, since they may view any rigid model as a hindrance to the research process. While prior case studies by Carver et al.[22] on scientific and engineering software indicate that a more “free-form” approach was better accepted, the unfavorable view of a structured plan was mostly due to too many unanswerable questions early on when

forming requirements. We believe that this can be alleviated by acknowledging the unknown and changing nature of research, only asking the most pragmatic questions related to the development, and allowing some blanks early on as these can themselves be research questions. The requirements and design should then accommodate exploration, which we will discuss in the following sections.

Adopting any model requires time spent on writing and reviewing documentation, coming up with test cases and other activities not directly related to producing research codes. While it would be hard to convince anyone otherwise before the project has started, we can still minimize such friction by forming a workflow more suited to research software development. This will reduce using the domain expert's time as much as possible.

Despite the potential pushback, one must be aware that following any structured model, however simple, can still be more beneficial in the long-term. It is much harder to bring an already chaotic project back to order when the need arises, a lesson learned from the 6DOF Trajectory project (Section [2.1.2](#)).

In the context of the current work, considerations for choosing a life cycle model include:

- It must be a document driven process that provide clear traceability between requirements, design, and implementation. This allows the domain expert who is not familiar with code to still understand the software. Any outside reviewer or future user will also benefit from this to replicate or build upon the work.
- To provide credibility to the research, the software must be tested and validated, ideally by test cases coming from the domain expert, not just the developer themselves.

- It has to accommodate the exploratory nature of research, such that the needs may change over time.
- It should not take any unnecessary time away from the domain expert, which may be seen as a hindrance to their research process.

The overall workflow we propose is based on the V-model [23], which was also used in the original template as shown by a recent paper around it (Figure 3.1)[3]. The main reason for choosing a V-model instead of a waterfall model is driven by the involvement of the domain expert. The V-model pushes the developer to gather as much information from the domain expert as possible upfront, for both requirements and testing. This allows the domain expert to have a decreasing involvement as development work moves closer to implementation, since they may not be that well-versed or interested in the development work. Another added benefit of working on system tests early is reinforcing the requirements by taking a complementary view.

Figure 3.2 shows our proposed V-model workflow, which also highlights the focus of this chapter and the two major documents where we would like to reduce overhead. The goal here is to get a Minimum Viable Product (MVP) out to the domain expert as soon as possible, without compromising adherence to requirements and testing. At the same time, this lays the foundation for proper software and documentation structure for version tracking and future development. Any future developments, may not need to go through the entire V again, but they should remain document-driven, with tests added/updated as necessary.

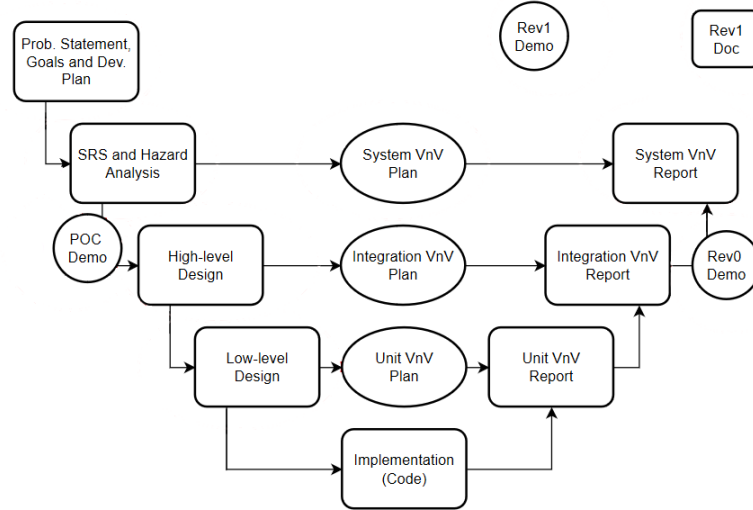


Figure 3.1: The original, full V-model which the current work builds upon, displayed in the paper [3]

3.2 Information Gathering

In our proposed workflow, writing the requirements document is the responsibility of the developer. However, before they can start writing, they need to gather the necessary information and understand the problem domain, as shown in Figure 3.2.

A lesson learned from working with domain expert in the SynthEddy project is that too much information is almost the same as too little information. It is understandable that the domain expert is very enthusiastic about their research field, and would like to share as much as possible. However, this can quickly lead to the developer feeling they are “drinking from a firehose” and not knowing what to do with the information. As we have established with the Intended Audience (Section 1.2), while it is possible that the developer may come from a similar background as the domain expert, which can allow them to achieve a research-level understanding

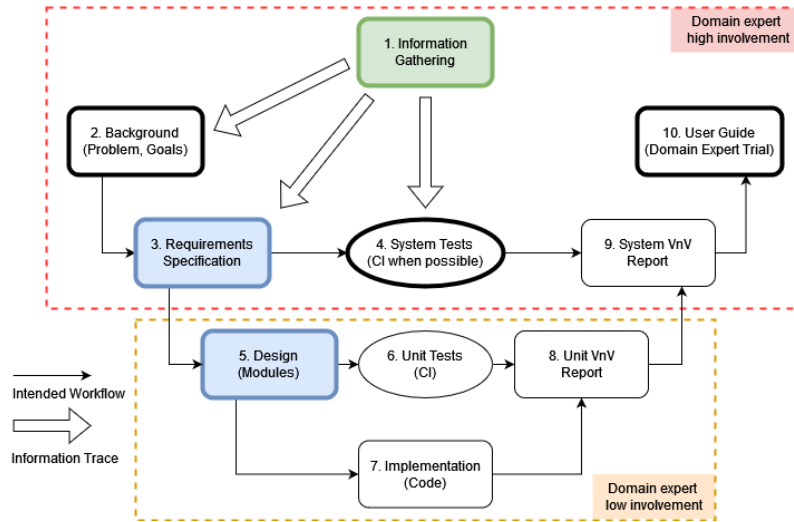


Figure 3.2: Our V-model workflow, with numbering indicating suggested chronological order of activities. Bold frame items are the main focus of this chapter. Green (Information Gathering) is newly introduced in this thesis. Blue items are where we will primarily cut-down development overhead.

of the problem, this is not always the case and certainly not their primary goal. The developer’s goal is to understand the problem enough to be able to represent it in software, and to be able to communicate with the domain expert in a way that is understandable to both parties. To be more specific, if there is an equation used in the software, the developer should be able to explain what each term in the equation represents in the context of the problem, but not necessarily why/how the theory behind the equation works, or how the equation is derived.

We propose a single (technical) meeting with the domain expert to extract as much required information as possible, while filtering out any unnecessary details. This meeting is directed by the developer using a set of proposed questions. These questions are divided into the following categories: **Problem Domain** (Section 3.2.1) helps the developer to understand the big picture of research field; **Software Goals** (Section 3.2.2) focuses on what the software should achieve; **Theoretical Model**


(Section 3.2.3) gathers the resources needed to implement the theory; **Scale of the Problem** (Section 3.2.4) sets performance expectations for solving a typical problem; **Data and Inputs/Outputs** (Section 3.2.5) is about how users can realistically interact with the software; And **Testing** (Section 3.2.6) discusses possible ways to verify and validate the software to build confidence in its correctness. The answers to these questions will help elicit software requirements and testing plans. The full details of the traceability between questions and documentation is shown in Table 3.1 and discussed in section 3.2.7.


Our envisioned meeting format is either in-person or virtual with screen sharing. The developer should take notes under each question, in their own words based on their understanding of the domain expert’s response. Questions and tips are provided in the <https://omltcat.github.io/research-software-template/template/first-meeting/> meeting agenda within our GitHub template (Figure 3.3 shows an excerpt of the agenda). The developer can simply edit this page to add their meeting notes. The domain expert can give comments or corrections on-the-fly, but they will also have a chance to review the notes at their own pace after the meeting and provide any additional information or corrections via the quick GitHub issues creation mechanism integrated into our template (Figure 1.3).

For simple problems, this meeting can be done in a single (one-hour) session. But for more complex problems, it may be necessary to schedule a longer meeting or break it down into multiple sessions, each focusing on different categories of questions.

The subsections below contain the list of questions that we propose to ask the domain expert in this meeting, along with the rationale behind each question. Note, not all questions may be applicable to all projects; it is acceptable to skip some


Meeting Start

- People introducing themselves and their roles.
- Developer:
 - Add the domain expert to your GitHub repository as a collaborator.
 - Let them know they can edit any document by clicking the `Edit this page` button on the top right.
 - Let them know they can open issues regarding any section of a page by clicking the  icon next to the section title.
(They should change the automated issue title to something more descriptive.)
- Domain Expert:
 - Give their short presentation.

 **Tip**

The `Report an issue` button exists on the right side of each question.

Problem Domain

 **Info**

The first questions are to gather some resources for getting a basic understanding of the problem domain. These can be informal/non-academic sources like online articles and videos just to help the developer get a sense of the problem domain.
We should not spend too much time on this as more specific questions will be asked later.

If I were to understand more about the problem domain, what keyword should I search for?

Is there any literature review, general overview paper or book chapter that you would recommend?

Figure 3.3: An excerpt of the meeting agenda page from the GitHub template.

questions if they are not relevant.

If possible, the questions and answers of the first category (Section 3.2.1) should be exchanged several days before the meeting (over email or other means) to give the developer a head start in understanding the problem domain. This will potentially improve the efficiency and focus of the information exchange during the meeting.

3.2.1 Problem Domain

At the beginning of the meeting, after any necessary introduction of personnel, the domain expert should give a brief presentation as an overview of their research field and problems they are working on. After this presentation and any immediate Q&A, the developer should start directing the meeting.

The first questions are to gather some resources for getting a basic understanding of the problem domain. We should not spend too much time on this as more specific questions will be asked later (Section 3.2.3).

- Q1.1: To understand more about the problem domain, what keyword should I search for?
- Q1.2: Is there any literature review, general overview paper or book chapter that you would recommend?

The keyword search does not necessarily mean scholarly sources, but can also include any web articles or videos that can quickly give a general idea of the problem domain. They can be informal as they are only to facilitate the developer’s understanding and likely not to be cited in the final software documentation. These can be stored as resources for the developer to refer back to as needed, not necessarily to be read in full before starting the project. It is also an opportunity for the domain expert to “unleash their firehose” before we start narrowing down the focus in the following questions.

3.2.2 Software Goals

Based on the presentation given by the domain expert and exchanges about the problem domain, the developer may already have a general idea of what problem the domain expert wants to solve with the software at this point. If so, the developer can simply voice their understanding and ask for confirmation. Otherwise, they should ask for a brief description of the problem.

- Q2.1: So my understanding is that you want to calculate/generate/analyze/etc X. Is that correct?
- Q2.2: Can you clarify what are we trying to achieve with the software?
- Q2.3: Is there any research questions or hypotheses that we are trying to answer with the software (if known at this point)?
- Q2.4: Other than yourself, who else might be the target audience of the software?
- Q2.5: At the very minimum, what should the software be able to do to be useful to its users?
- Q2.6: Are there any special requirements in terms of usage? Such as...
 - A graphical user interface (GUI) is preferred over command line (CLI).
 - The development goal is a library that is imported into other projects.
 - Must be able to interface with certain software/workflow.
 - Plan to run on computing clusters.

Due to the uncertain nature of scientific research, it is possible that the domain expert may not have a clear idea of what the software should exactly do at this point. This is fine, and we have Q2.3 as optional. We will greatly narrow down the focus in the following questions about the theoretical models (3.2.3) and user inputs/outputs (3.2.5).

The initial goal of the elicitation is to acquire the information needed for a Minimum Viable Product (MVP). This will help the developer scope their first version of the software.

As for the research questions/hypotheses (Q2.3), they are not the same as the software goals themselves, but rather what the domain expert hopes to do with the software that is directly relevant to their research. This can potentially be an easier question to answer than the straight-up software goals. However, the domain expert may use the software in different ways than an external user, warranting additional considerations if we were to generalize the software for broader use. We will discuss this further in the requirements section (3.4).

The answer to Q2.6 can influence the choice of programming language, libraries, and other software design decisions. The developer should also help the domain expert to understand their expectations. For example, if the domain expert prefers a GUI, the developer should explain the potential overhead in development time and maintenance, and the potential limitations in functionality, such as that you cannot run the GUI software on a computing cluster if that is also a requirement.

3.2.3 Theoretical Models

Based on our experience with the previous projects, the domain expert has most likely already extracted the most useful information of the theoretical models from the literature to use in their own calculations. They may have developed their own simple program or scripts using such models. The developer should specifically ask for this information to avoid going over the same process in a domain where they may not have a deep understanding.

- Q3.1: Regarding the model you are working on, is there a paper or book or your own writing, that I should read to gain a basic understanding of WHAT it is and HOW it works, without going into WHY it works? (Those answers could already be included in the previous suggested readings, i.e. Q1.2.)
- Q3.2: Which chapters or sections should I focus on?
- Q3.3: Which equation or set of equations is the core of the model?
 - If there are multiple equations, what are their relationships/order of calculation?
 - Are there any terms/symbols in the equations that are not defined in the literature, because they are considered common knowledge in the field?
- Q3.4: Are there any variations (different forms) of these equations that I should be aware of?
- Q3.5: For your own research or our software implementation, are there any modification to the equations from the original literature?

- Q3.6: Are there any simpler, better-understood models, perhaps less accurate models for the same problem? We can potentially use them for testing purposes.

In the case with multiple equations, while their relationships may be obvious just from observing the equations, it is still good practice to ask the domain expert to confirm. Typical relationships include parallel (several equations can be calculated independently and feed into one governing equation) and sequential (one equation depends on the result of another) as shown in Figure 3.4.

A parallel relationship examples would be, in the 6DOF Trajectory project, the governing velocity and rotation differential equations:

$$\begin{aligned}\frac{d\mathbf{V}}{dt} &= \frac{\mathbf{F}_D + \mathbf{F}_L + \mathbf{F}_{N_{p\alpha}} + \mathbf{F}_{N_{q+\dot{\alpha}}}}{m} + \mathbf{g} + \mathbf{\Lambda} \\ \frac{d\mathbf{h}}{dt} &= \frac{\mathbf{M}_a + \mathbf{M}_{M_{p\alpha}} + \mathbf{M}_{I_p} + \mathbf{M}_{M_{q+\dot{\alpha}}}}{I_T}\end{aligned}$$

In the above equations, while every \mathbf{F} and \mathbf{M} term is a complicated function on its own, none of them depends on each other within a single time step, and can be calculated in parallel.

Note that this is purely a discussion of the theoretical models structure, and not yet about software implementation (e.g. how to parallelize the calculations).

3.2.4 Scale of the Problem

These questions are to gather information on how the models fit into a usable software, and how the software is expected to be used by the domain expert and other potential users.

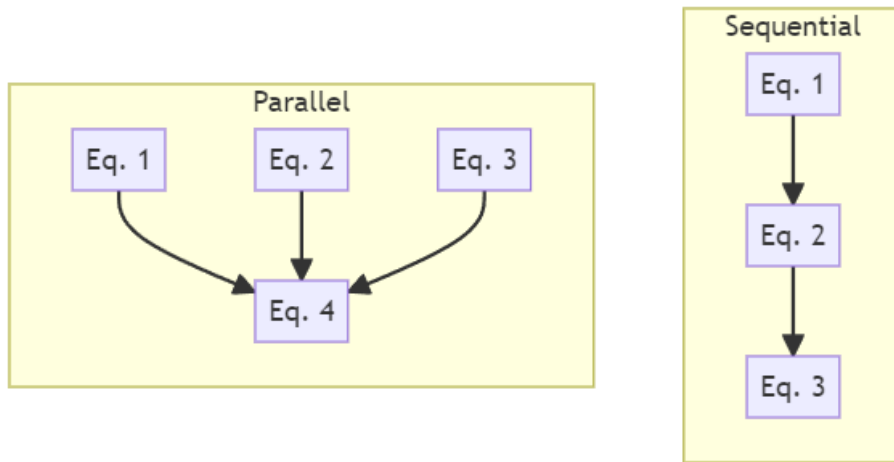


Figure 3.4: Typical parallel and sequential relationships between equations in a model

- Q4.1: When using the model for actual problem-solving, how is the model invoked? Once per input/data point or many times in series/parallel to approach a solution?
- Q4.2: In a typical use case, what is the approximate scale of the problem. Depending on the problem/model, this can be in terms of:
 - Number of data points
 - Matrix/vector/meshgrid size
 - Simulation length and time step
 - Data rate expected to flow through the software (for a real-time application)
 - Number of iterations/steps expected before the result can be considered acceptable
- Q4.3: Given the problem scale, is there any performance expectations or metric

that we should be aware of? (The domain expert may not have a clear idea of this, and we may only be able to get a better picture after the proof of concept phase. However, we should record the initial response.)

It is possible that the theoretical model does not represent the problem from end to end. For example, if the model is a differential equation, it only describes how the system should evolve at any given point in time, like in the case of the trajectory simulation project. Or the model only represents a localized point in space, and the solution would be the sum of influence of many such points, like in the case of the SynthEddy project. In such cases, the model is invoked many times to approach/generate a final solution.

Given Q4.1 and Q4.2, we may be able to perform some preliminary calculations to get a rough performance implication at the expected problem scale, and if the performance expectation from Q4.3 is feasible at all. This is a hard lesson learned from the SynthEddy project, where the typical use case in terms of meshgrid size was not clearly communicated between the developer and the domain expert, leading to the developer to believe that it was feasible for the software to use minimum computing resource. If the developer was aware of the generally accepted grid resolution for publication, which was 1000^3 to 2000^3 at the time, a simple calculation would reveal that if each node consisted of three floating point numbers (3D vector), the minimum amount of data to be processed would be non-trivial, even after best efforts in memory optimization:

$$1000^3 \times 3 \times 8 \text{ bytes} = 24 \text{ GB}$$

$$2000^3 \times 3 \times 8 \text{ bytes} = 192 \text{ GB}$$

The scale of the problem and any performance expectations may therefore influence the software design, which we will discuss in more detail in the section [3.6.2](#). Thus, it is important to gather this information early on so that the developer and domain expert stays on the same page. This can avoid major redesign later on due to performance issues, like what happened in the SynthEddy case regarding the aforementioned memory management challenge (more details are in Section [2.2.4](#)).

3.2.5 Data and Inputs/Outputs

These questions tell the developer what inputs and outputs the software should expect, and how the domain expert or other users would interact with the software. It also helps the developer to understand where the theoretical model sits in the grand scheme of the implementation with respect to the input/output.

- Q5.1: Does the model require any additional data to function? This can be fixed or tabulated coefficients, tweaking parameters, etc.
 - If so, is the user responsible to provide these for their specific use case, or should we include them in the software?
 - If we were to include them in the software, where do we get these data from? Are they from any literature, your own experiments, etc.?
- Q5.2: Ideally, what should the user provide as inputs?
- Q5.3: Realistically, what can we currently expect the user to be able to provide as inputs? If they have difficulty providing some inputs, what should we do (e.g. default values, derive from other inputs, etc.)?

- How does solution quality degrade if some inputs are not provided (if this is known)?
 - If the method to treat missing input is non-trivial, we should ask for examples, literatures, or even treat them a separate theoretical model.
- Q5.4: What does the user expect as output from the software? Does the user want any additional information like accuracy, intermediate results or metrics, etc.?
 - Q5.5: Are the input and output directly to and from the model, or are there any pre-processing or post-processing steps that the software needs to do? If these are non-trivial steps, we should again ask for examples, literatures, or even treat them a separate theoretical model.

Real world values can be messy and may not always be available in the format that the model requires. Here we should direct the domain expert to consider when they are using the software, what can they realistically provide as input. For example, in the 6DOF trajectory project, users may face difficulties in providing all the required inputs when switching to a different projectile type (Section 2.1.2). While the developer strives to make the software degrade gracefully when some inputs are missing, the domain expert should be aware of the need for potential “educated guesses”. The domain expert should be the one to provide guidance on how to make such guesses, as their choice can inadvertently affect the results of their research and potentially other people in their field. This information should be recorded as assumptions in the requirements document. It should be noted that some of the research questions themselves can be about the inputs, which the domain expert would rightfully not

have a clear answer to at this point. We should simply record the possibilities now and build the software to accommodate such exploration later on (Section 3.4.3).

The lack of data required to power the model can be a major roadblock in the applicability of the software in more general use cases. In the 6DOF trajectory project, the author was only able to find and clean up the set of required aerodynamic coefficients for one type of projectile from the published literatures (Figure 3.5 shows one of such coefficient as an example). While technically the user can bring in their own coefficients, in reality this can be a very challenging task that require either CFD simulations (for which a clearly documented process could not be found at the time of that project) or real-life experiments (which is not possible for most users or if the projectile is still hypothetical). These limitations should be made clear to the domain expert and documented. However, if resources permit, more serious consideration should be given to how to overcome these limitations.

3.2.6 Testing

As a part of the V-model, we should gather information on testing early on. As you will see in the following sections, the availability of test cases can even influence the software design, as we seek proof of correctness of the software implementation.

- Q6.1: If you were to use the software in your research, what would you give as typical inputs (or input ranges)?
 - It would be even better if expected output can also be provided, but this will most likely not be the case, except for some very simple/degenerate special cases which we will specifically ask for in Q6.5.

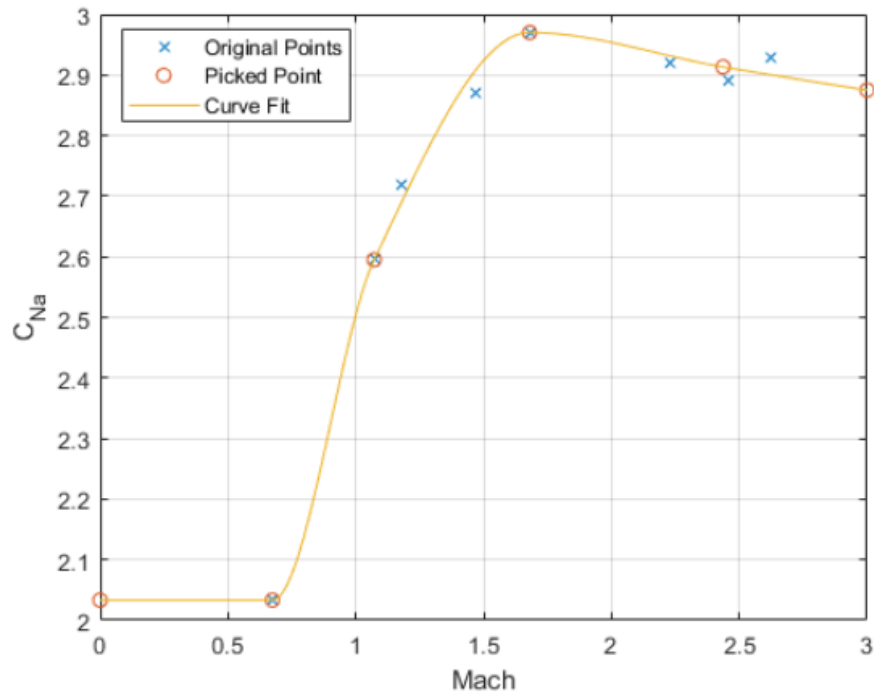


Figure 3.5: A plot from the 6DOF Trajectory paper [4] showing the interpolation of normal force (lift) coefficients at different mach number to be used in the program, which is non-trivial if the user wants to use a different projectile type without appropriate data.

- Q6.2: Are you aware of any (pseudo) oracles that we can use for testing? These can be...
 - Known to work implementations. If not on our main model of interest, perhaps on a simpler model that you mentioned earlier.
 - Analytical solutions, if they exist, even on a simpler model.
 - Experimental data that we can adapt into input/output pairs.
- Q6.3: What are some general trends or patterns we can expect from the model given different inputs?

- Q6.4: If other models are provided, how do you expect the results to differ in certain special cases?
- Q6.5: Are there any simple/degenerate/special cases that we can use to test the software?

In Q6.2, a (pseudo) oracle is a source of truth that we can use to validate the software against. This can be a known-to-work implementation (another piece of software already developed), analytical solutions, or experimental data. For example, in the 6DOF Trajectory project, the calculators publically provided by [JBM Ballistics](#) were used as oracles to the simple point mass model.

Naturally, an oracle would ease our validation process immensely, but as discussed in the literature review, this is more often than not unavailable in the case of research software [9]. So we instead ask for other information that may help us build test cases to increase our confidence in the software. One set of such information is trends and patterns (Q6.3) that allows us to use strategies like Metamorphic Testing (MT), which allows automated system testing in the absence of an oracle [24].

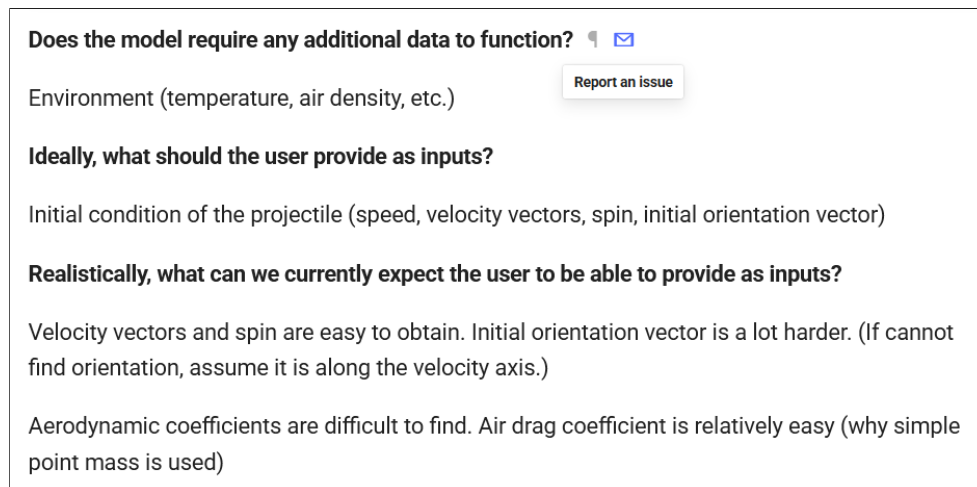
Information gathered here will be refined into test cases in the testing plan, which will be discussed in the Testing section (3.5). The developer should note the sources of any non-trivial test cases (whether they come from the domain expert’s own work or other literatures) to maintain traceability in the testing documentation.

This is also a good opportunity to introduce the concept of continuous integration (CI) to the domain expert. Without going into the technical details that they may not be familiar or interested in (GitHub Actions, pull requests, etc.), we should explain that if there are test cases that should always be true regardless how the software is changed in the future, we can use them to set up an infrastructure that automatically

run these tests whenever the software is updated.

3.2.7 Refining Gathered Information

When following the GitHub template, the end result of this meeting should be a viewable webpage with the developer’s understanding of the answers to each applicable question. The domain expert can give any immediate feedback either during the meeting, or in the following days via GitHub issues, which can be linked to each question thanks to the template. A ready-for-review meeting note excerpt from a mock meeting using the 6DOF Trajectory project information is shown in Figure 3.6. This mock meeting is later discussed in the Experiment Design section (4.3.1).



The image shows a screenshot of a GitHub meeting note template. The title is "Does the model require any additional data to function?" with a "Report an issue" button. The content is as follows:

Environment (temperature, air density, etc.)

Ideally, what should the user provide as inputs?

Initial condition of the projectile (speed, velocity vectors, spin, initial orientation vector)

Realistically, what can we currently expect the user to be able to provide as inputs?

Velocity vectors and spin are easy to obtain. Initial orientation vector is a lot harder. (If cannot find orientation, assume it is along the velocity axis.)

Aerodynamic coefficients are difficult to find. Air drag coefficient is relatively easy (why simple point mass is used)

Figure 3.6: An excerpt of a filled-in meeting note using the GitHub template. In this case the domain expert may want to create an issue regarding the “additional data” question, as the developer did not write down an exhaustive list.

After the feedback is addressed, the developer will start gradually refining the informal meeting note into more formal documents, which will be used in the software development process. These documents include:

- A background document defining problems and goals (3.3).
- A requirements document as the overall guide to development process (3.4).
- A testing plan to validate the software (3.5).

After this point, the developer should direct the attention of the domain expert to these formal documents for further communications instead of lingering on the informal meeting notes. Table 3.1 show the approximate relationships between the question asked in the meeting and the documents/sections that they will be refined into. Note that this is not a definitive mapping, as the responses given by the domain expert may not fit neatly into these categories, and the developer may need to make some judgment calls on how to best organize the information.

From the meeting notes and throughout the development process, the developer should also maintain a list of useful resources (literature) and a quick reference with common symbols, abbreviations, acronyms, mathematical notations and key concepts used in the project. This page can be linked from any other page throughout the documentation website, and may keep growing as the developer gains deeper understanding of the problem and writes more documentation. Throughout the project, words and symbols on this page should be the standard way of communicating, documenting and code naming. This address the problem of the domain expert and the developer not speaking the same language, and terminologies in communication becoming disconnected from code, as mentioned in Domain-Driven Design [12].

As it is common for multiple sources to have different symbols and notations for the same or similar concepts, it is important that standards in a project are established early on to maintain consistency in all documentation, code and communication.

| | Background | | | | Requirements | | | | | | | Testing |
|---------------|-----------------------------|----|----|----|--------------|----|----|-------------------------|----|----|-----|---------|
| | Lt | PS | GS | SH | SS | As | SD | TM | IM | FR | NFR | ST |
| Q1.1 | X | | | | | | | | | | | |
| Q1.2 | X | | | | | | | | | | | |
| Q2.1 | | X | | | X | | | | | | | |
| Q2.2 | | | X | | X | | | | | | | |
| Q2.3 | | X | X | | | | | | | | | |
| Q2.4 | | | | X | | | | | | | | |
| Q2.5 | | | | X | | | | | | X | | |
| Q2.6 | | | | X | | | | | | | X | |
| Q3.1 | X | | | | | | | X | | | | |
| Q3.2 | | | | | | | | X | | | | |
| Q3.3 | | | | | | | | X | | | | |
| Q3.4 | | | | | | | | X | X | | | |
| Q3.5 | | | | | | | | | X | | | |
| Q3.6 | | | | | | | | X | X | | | X |
| Q4.1 | | | | | | | | | X | | | |
| Q4.2 | | | | | | X | | | | | X | |
| Q4.3 | | | | | | | | | | | X | |
| Q5.1 | | | | | | X | X | | | | | |
| Q5.2 | | | | | | | X | | | | | |
| Q5.3 | | | | | | X | X | | | | | |
| Q5.4 | | | | | | | X | | | X | | |
| Q5.5 | | | | | | | X | | X | | | |
| Q6.* | | | | | | | | | | | | X |
| Abbreviations | | | | | | | | | | | | |
| Lt | Literatures | | | | | | PS | Problem Statement | | | | |
| GS | Goal Statement | | | | | | SH | Stakeholders | | | | |
| SS | Software Scope | | | | | | As | Assumptions | | | | |
| SD | System Description | | | | | | TM | Theoretical Models | | | | |
| IM | Instance Models | | | | | | FR | Functional Requirements | | | | |
| NFR | Non-functional Requirements | | | | | | ST | System Tests | | | | |

Table 3.1: Traceability between meeting questions and document sections.

3.3 Consolidating Background

After the technical meeting with the domain expert, ideally, the developer should understand what they need to develop. However, this may not always be the case. While most may point to the high specialization of the research field and worry about the developer’s true understanding of the problem, we argue that an even greater challenge is the open-ended nature of research, which can make it hard for the domain expert themselves to articulate exactly what they were looking for in the software. This was the case in the early phase of the SynthEddy project, where the domain expert gave a good presentation on the theoretical side, but the developer then had to gain a deeper understanding of the problem domain to dig out the practical use case of the software. Fortunately in this instance, the end product was quite usable by the domain expert. But naturally, we want to minimize the risk of not being on the same page.

This challenge lead to the first formal writings that the developer should create the Problem Statement and Project Goals. These should be born from the developer’s own understanding of both the meeting notes and any further reading that they may have done. Even if the domain expert’s initial description is vague, the developer should make some best-effort guesses and let the domain expert review them. This will likely spark further discussions with the domain expert to narrow-down the understanding. One mistake that the developer of SynthEddy made was waiting too long to discuss these with the domain expert, leading to misaligned expectations of what the software was actually for, and by extension the performance problem as discussed in Section [3.2.4](#).

3.3.1 Feasibility Considerations

While a full-on feasibility study that include aspects like economic, legal or cultural feasibility is out of scope of the methodology described in this thesis, some considerations of technical feasibility should be made early on. Since we have a research problem on our hand, it is unlikely that someone has already “invented the wheel” for us. However, development will be more efficient if we do not have to make everything from scratch. From the domain expert’s description of the problem, further literature readings and looking at other open-source projects in similar fields, the developer may be able to note some potential candidates of languages, libraries and tools (the “tech stack”) that can mark an easier path to the final software.

While usually we think that the tools should adapt to the problem, and the above discussion should happen much later in the development process (after requirement and in the design or implementation phase), our experience shows that for research software, such adaptation can work both ways. Often the problem of theoretical model can take different forms, some easier to work with given existing tools than others. For instance, writing all ODEs in the standard form of $y' = f(y, t)$ will make later interfacing with a numerical ODE solver much easier. The tech stack considerations here will provide some insights when we try to refine these models in the requirements document and come up with the design, which we will discuss in details in the section [3.4.4](#), with an example from the SynthEddy project regarding choice of notations.

3.4 Software Requirements

In this section we discuss some key considerations for the requirement document and the rationale behind them. The full structure of this [document](#) in our GitHub template is shown in Figure 3.7.

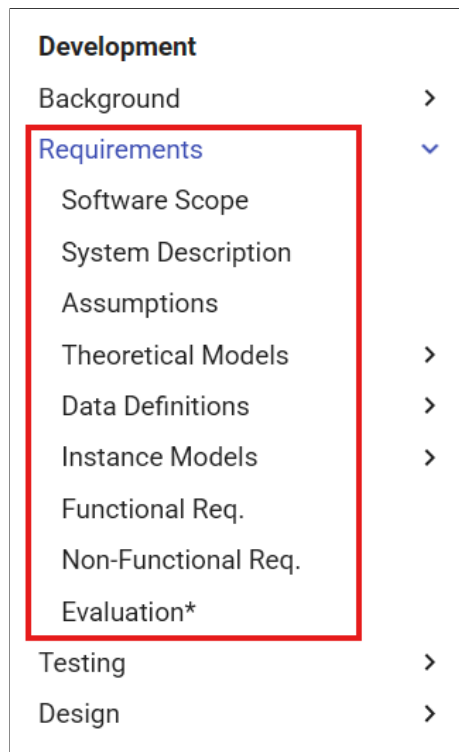


Figure 3.7: Structure of the requirements sections within all development documents in the GitHub template

3.4.1 Inputs Values and Data

While it is common for software to only require a subset of the input values from the user and populate the rest with default values, this takes on a different level of significance in research software. When the input is related to certain physical/experimental values, it is possible that the user may have some difficulty in providing them, as we

have discussed in questions with the domain expert. When writing the requirement document, not only should these default values be noted, but the sources of how these values are decided should also be traced.

For some trivial values this can be a developer’s decision, such as the default flow velocity being zero in SynthEddy. But for values that can potentially affect the results of the research, the domain expert should be making the decision. If more lengthy justifications of these values are needed, a “Default Value Decisions” or “Data Sources” subsection should be added to the requirements document. Figure 3.8 shows an example of how we can document the input values in our GitHub template.

| Name | Symbol | Unit | Default Value | Acceptable Range or Typical Value | Notes |
|------------------|-------------------|-------------------|---------------------------|--------------------------------------|---|
| Initial Speed | v_0 | m/s | 10 | $0 < v_0$ | |
| Initial Angle | θ | rad | $\frac{\pi}{4}$ | $0 \leq \theta < \frac{\pi}{2}$ | |
| Drag Coefficient | C_d | - | see below | 0.8 | Interpolated from table |
| Gravity | g | m/s ² | 9.81 | $0 \leq g$ | |
| Air Density | ρ | kg/m ³ | 1.225 | $0 \leq \rho$ | |
| Cross Wind | v_{wind} | m/s | 0 | $-\infty < v_{\text{wind}} < \infty$ | Perpendicular component to flight path |
| Time Step | Δt | s | 0.01 | $0 < \Delta t$ | Will affect accuracy and computation time |
| Some Input | x | unit | 0 | $0 \leq x \leq 1$ | |
| ... | ... | ... | ... | ... | ... |

Figure 3.8: An example input value table in our GitHub template. Note how we include links on non-trivial default values (drag coefficient in this case) to further explanations

On a similar note, in most modern research, it is rare to have a model functioning on its own without any external data. This can be in the form of tuning parameters, coefficients (such as the aerodynamic coefficients in the trajectory project), or even other models/functions that can plug into the main model (such as the eddy shape function in SynthEddy). They can be datasets from other sources, or even the domain expert's own experiments or calculations (such as the default eddy profile in SynthEddy as shown in Figure 3.9). Unlike the inputs, these are usually not changed from run to run, but can have a significant impact on the results.

```

"variants": [
  {
    "density": 618.1,
    "intensity": 0.3977,
    "length_scale": 0.04000
  },
  {
    "density": 348.2,
    "intensity": 0.5137,
    "length_scale": 0.06108
  },
  {
    "density": 99.96,
    "intensity": 0.6328,
    "length_scale": 0.09325
  },
  {
    "density": 26.14,
    "intensity": 0.7521,
    "length_scale": 0.1424
  },
  {
    "density": 7.157,
    "intensity": 0.8681,
    "length_scale": 0.2174
  },
  {
    "density": 2.035,
    "intensity": 0.9737,
    "length_scale": 0.3320
  },
  {
    "density": 0.5662,
    "intensity": 1.059,
    "length_scale": 0.5069
  },
  {
    "density": 0.1481,
    "intensity": 1.117,
    "length_scale": 0.7739
  },
  {
    "density": 0.02769,
    "intensity": 1.145,
    "length_scale": 1.182
  },
  {
    "density": 0.002625,
    "intensity": 1.153,
    "length_scale": 1.804
  }
]

```

Figure 3.9: Default eddy profile provided by the domain expert to be used in SynthEddy. Without proper traceability, these are just opaque numbers.

If these are included in the software in case the user does not bring their own, then in a way we are publishing/republishing the data. To be fully transparent, they

should be added to Data Definitions (Section 3.4.4) in the requirements document, with the source of the data cited.

3.4.2 Typical Use Cases

Just like a non-linear system is often linearized at a certain point before analysis, a piece of research software may be developed to best serve a certain use case, as we have tried to understand from the domain expert in the previously discussed meeting questions (3.2.6, Q6.1). In the most ideal case, we are able to turn these use cases into upper and lower bounds of each input value, within which the software should be guaranteed to function correctly. This would vastly simplify the writing of test cases later on. But it is possible that even the domain expert may not have a clear idea of these bounds. Therefore, we instead ask for one or more typical use cases, near which the software should stay trustworthy.

3.4.3 The Dual-Purpose of Research Software

We have mentioned that research software can have a dual-purpose: to aid the domain expert in their immediate research, and to be used by other potential users in the future after the relevant theory is fleshed out and has been published. For the latter group, the software is more of a pre-programmed calculator: given a set of inputs, it will return some outputs. This is more inline with traditional software. However, for the domain experts, they may use the software in a more exploratory way, and ask research questions like “If I change X, what will happen to the result?”.

For example, in the 6DOF trajectory study, one key research question that emerged

was the effect of different physics simulation time step sizes on precision and computing time. This was a parameter of the differential equation solver, instead of a direct input to the theoretical model. For a commercial ballistic calculator, this would be a certain optimal value likely not exposed to the user, but it was exactly such research question that would find this optimal value that was unknown at the beginning of the project.

Thus, we propose a more “liberal” approach to defining the requirements for software inputs. Instead of strictly adhering to the inputs of theoretical models, we should leverage Q2.1-2.3 to include more parameters that tweak the software behaviors that the domain expert may want to explore in their research, of which they would potentially come back to change the default values once they have a better understanding of the problem.

This would be two steps ahead of simply limiting the software with scope definitions and assumptions based on what is currently known, and one step ahead of designing for change, which still requires the domain expert to come back and ask for changes and the developer to modify the code.

Another realization of this philosophy is the eddy shape function (which defines the tangential velocity distribution with respect to the eddy center) in the SynthEddy project. Instead of forcing the domain expert to give us “the best one” (which they may not be able to answer yet) to use in our calculation, we include this choice as an input to the software, and even allowed user defined shape functions in a designated file.

3.4.4 Refining the Models

The goal of this refining step is to arrive at one or more “Instance Models” (IM) that represent the specific calculations that the software should implement. An instance models should be in the form of a mathematical function that take in some inputs to come up with an output. This refinement starts from “Theoretical Models” (TM) as seen in literatures, which may take a more “liberal” form such as a set of equations, inequalities, or even a set of rules.

In the workflow originally proposed by Smith [2] and in Drasil [16], such refinement should begin with the most basic physical laws and gradually build up to the domain specific models for maximum traceability and reusability. We however, recognize that this is likely not feasible in our typical development scenario, as it would require the author of this requirements document, namely the developer, to have a deep, research-level understanding of the problem domain to describe how the model are derived step by step; even in the case of domain experts being the developers themselves, this would be a time-consuming process and likely viewed negatively.

Instead, we propose a much shorter path of refinement: from the theoretical models as seen in the most relevant literature to the instance models that should eventually be implemented in the software. The traceability in our case for how the original theoretical models is derived would be “out-sourced” to the literatures and cited instead.

Figure 3.10 shows an example of a theoretical model in our GitHub template, which is an umbrella equation that describe the forces acting on a projectile. Figure 3.11 shows an example of an instance model, which is a more specific case of the theoretical model, namely a simple point mass model. Links are used to provide traceability between different pages.

TM1: Velocity Differential Equation

Equation

$$\frac{d\mathbf{v}}{dt} = \frac{\sum \mathbf{F}}{m} - \mathbf{g}$$

Description

This differential equation describes the change in velocity at each time step (Δt) given the forces acting on the object and the gravitational acceleration.

- \mathbf{F} is the sum of all forces acting on the object.
- \mathbf{g} is the gravitational acceleration vector, pointing downwards.
- m is the mass of the object.

Notes

To use this differential equation, we need to start from an initial velocity \mathbf{v}_0 and update the velocity at each time step.

Sources

Carlucci 2007¹

Uses

Other models/definitions that are used in this model.

[DD1: Sum of Forces](#)

Referenced by

[IM1: SPM](#)

Figure 3.10: An example of theoretical model in our GitHub template.

IM1: Simple Point Mass 📄 📄

Inputs

- ρ : Air density
- S : Cross-sectional area of the object
- C_D : Drag coefficient at the current Mach number
- \mathbf{v} : Current elocity vector
- v : Magnitude of the velocity vector
- m : Mass of the object
- \mathbf{g} : Gravitational acceleration vector

Equation

$$\frac{d\mathbf{v}}{dt} = \frac{-\frac{1}{2}\rho S C_D \mathbf{v} v}{m} - \mathbf{g}$$

Description

The differential equation describes the change in velocity at each time step (Δt) of a point mass object experiencing only drag force and gravitational acceleration.

Notes

Uses

[TM1](#), [TM2](#), [DD1](#)

Figure 3.11: An example of derived instance model in our GitHub template.

It is possible that the theoretical models and the instance models are the same, with minor modifications or rearrangements, etc. But more likely, the instance models represent a more specific case of the theoretical models, refined by assumptions or added information as discussed with the domain expert in the meeting. For example,

in the trajectory project, if we were to put it into our template, the theoretical models would contain all the forces and moments exerted onto a projectile. Then we take the assumption that the Magnus force is too small thus negligible, and the instance model would be the differential equations of motion without the Magnus force term.

Arguably, the model definitions are the most important part of the requirements document, as the developer will keep referring back to them throughout the development process. Therefore, the following subsections (Section 3.4.4 to 3.4.4) will provide some key advices on how to complete this part with our template to better serve development work down the line.

Assumptions vs. Scope Definitions

There is a potential confusion between assumptions and scope definitions. In our template, assumptions are part of the refinement process that helps to shape the theoretical models into instance models. Some assumptions may also be marked as likely changes as future version of the software may expand to accommodate other/more general cases.

Scope definitions, on the other hand, are the boundaries of the software, before even considering models. They represent decisions that we made for usually more practical reasons. For example, in the SynthEddy project, we decided to only consider 3D space. Could models for 2D space be drawn? Yes, but such use case would likely not be very useful for the domain expert or any other potential users, so there was no reason to divert resources to implement them.

Data Definitions

Most likely, the models in our requirements document will include a number of non-trivial variables that are not taken directly as inputs, but rather require their own calculations. We would record these as Data Definitions (DD), so that there is a clear path from inputs to outputs with no ambiguous terms in between.

The difference between a data definition and a model is that the former is usually simpler calculation, which is more or less considered “common knowledge” in the field, while the latter are the more complex calculations that are the main focus of the research.

Model Forms and Notations

In the meeting with the domain expert, we asked for different forms of the theoretical models that we should be aware of. To be more specific, we are referring to different mathematical representations of the same model (or with slight variations). In the literature, these sometimes come in as several derivation steps to a final form preferred by the author. The intermediate forms are just as valid.

When drafting the requirements document, we want to choose a form that is most suitable for software implementation, which may not be the go-to form by the domain expert. We have experienced this in both the trajectory and turbulent projects, which were both three-dimensional problems. Generally speaking, if the programming language or libraries that a project intends to leverage have well-optimized vector/matrix operations (such as MATLAB or Python with NumPy), then representing the model in vector form would likely lead to a more efficient implementation,

compared to looping through each dimension. The domain experts however, may prefer a different form in their own work that may help them control variables, ease hand calculations/derivations/reasoning, etc. In the case of SynthEddy, the original governing equation in the paper by Poletto et al. for fluid velocity fluctuation was first introduced in vector form (Equation 4 in [21]). It was then derived into Levi-Civita notation (Equation 5 in [21]), enabling researchers to decompose the equations into each component further in the paper. The developer eventually chose to use the vector form before any derivation as a starting point in the software documentation (Figure 3.12) and implementation, to better leverage the aforementioned vector/matrix optimizations.

| | |
|-------------|---|
| Number | TM2 |
| Label | Velocity Fluctuation Field |
| Equation | $\mathbf{u}'(\mathbf{x}) = \frac{1}{\sqrt{N}} \sum_{k=1}^N q_{\sigma}(d^k(\mathbf{x})) \mathbf{r}^k(\mathbf{x}) \times \boldsymbol{\alpha}^k$ |
| Description | <p>The equation gives the fluctuation of velocity (the deviation from the mean flow velocity) at point \mathbf{x} in the flow field. It is the normalized sum of the velocity fluctuation by each eddy, with shape function q_{σ} [TM1].</p> <ul style="list-style-type: none"> • N is the number of eddies. • \mathbf{r}^k is the normalized distance to the center of the kth eddy [DD1]. • $\boldsymbol{\alpha}^k$ is the intensity vector of the kth eddy (m/s) [DD4]. |
| Notes | According to TM1, at any single position, the fluctuation from most eddies will likely be 0, due to outside the boundaries of most eddies in the field. |
| Source | Poletto et al. (2013) |
| Uses | TM1, DD1, DD4 |
| Ref. By | GD1 |

Figure 3.12: The eventual chosen form of the theoretical model documented in the SynthEddy project.

This model variation may also go another direction, namely being too abstract. In the trajectory simulation project, the original author introduced terms like “specific

moment” which lacks any direct physical meaning, but helps to reduce complexity of the equations for hand calculation. As the developer may not have the same level of domain expertise, such forms may pose a challenge to their understanding compared to a form with more direct physical meaning.

Since our responsibility as software developers is only to implement the model and not to do the research, we are not bound by the same constraints and preferences as the domain expert. Instead, we should choose the form that is either most efficient to run or easiest to understand and implement/maintain in the code.

While different forms and notations are equivalent in theory, the implication can go beyond what is easy to read, understand, implement in code or performance impact. The developer should give their reasons for choosing a certain form in the requirements document for the domain expert to review and respond to. For example, in the case of SynthEddy as mentioned above, while the vector form in the software requirements and implementation is essentially equivalent to the Levi-Civita notation in the paper, there was a future challenge on the horizon: Because the domain expert used the paper format in his own work, it was easier to keep everything pointing upward (positive- z), and then apply a transformation matrix. This is however not needed in the code as a computer would have no problem calculating vectors in any arbitrary direction. But because of this preference, the domain expert progressed his work with the transformation matrix built-in. This could create a future challenge when we want to integrate his new work into the existing codebase, as it is like a merge conflict, but between theory and code.

3.4.5 Requirements Validation

Because our requirements document consists of individual pages instead of a single monolithic document, they can be reviewed piecemeal by the domain expert, submitting issues on each page and section as they see fit. If a more formal validation is needed, this bite-sized structure also fits well into the Task Based Inspection process that breaks down the review process into manageable tasks. This was done by another developer-domain expert paired project [15] [25].

3.5 Testing

While all software requires some form of testing, however informal, to ensure that it works as intended, for research software testing takes another level of significance. Since we argue that research software should withstand the same level of scrutiny as the research publication itself, proper testing routines provides not only confidence in the software but also credibility to the research that uses it. Previous studies also showed that this is also one aspect that is particularly challenging for the researchers themselves due to lack of relevant knowledge [1], and thus requires more attention from the developer’s side.

In this section, the focus of our discussion is on planning the “System Tests” (ST) and setting up testing infrastructure, although ideas Continuous Integration (CI) and related tools are also applicable (and highly recommended) for unit tests. Irrespective of how exactly the software is implemented, the system tests view the software as a whole and check if the requirements, as outlined above, are met. We can defer the making of unit tests, which test individual components of the software, to

the implementation phase as we write the code, since the requirements do not specify the inner workings of the software.

3.5.1 Influence on Software Design

In a V-model, testing is considered before the actual design and implementation. Based on Q6.* (Section 3.2.6) asked during our meeting with the domain expert, we may have only so many resources at our disposal to create test cases. Thus, our later design may be influenced by what CAN be tested, rather than blindly ask for what SHOULD be tested. This necessitates a plan so that testing activities is optimized to match available resources.

For example, in the trajectory project, for simpler point mass models, there were plenty of oracles in the form of ballistic calculators available publically. But for the more complex 6DOF model in question, none was available. Situations like this may call for a modular design where different models can be plugged in over a common interface. If tests over the simpler models pass, then we can at least draw the conclusion that everything surrounding the model interface is working correctly (input pre-processing, output post-processing, etc.). The only unknown left is the model itself, which may also be the research question that the domain expert is trying to answer anyway, and thus is justifiable that we do not know the answer beforehand. The “untestable” part in this case should be designed as small as possible, containing little information beyond what is outlined in the models section of the requirements document, which would allow us to isolate it and potentially validate it via alternative, manual techniques, such as via code walkthroughs, code inspections, etc. [2].

This practice of keep the models concentrated to themselves, away from the rest

of the code is similar to the idea of a dedicated “Domain Layer” in Domain-Driven Design [12]. Evans strongly discourages “diffusing” domain-related code throughout the software, although his reasoning went beyond just testing.

As mentioned in section 3.2.6, the domain expert may also suggest some simpler (degenerate) cases that can be used for testing (Q6.5). For example, in SynthEddy testing, several test cases used a single large eddy in the center of the entire field. Cases like this may not be part of the general use cases, but to allow such tests, the design needs to either support them natively or allows for easy modification of relevant data from the test scripts.

In our GitHub template, we include a “Connection to Test Cases” section in the design document, with some hints on how to design the software to facilitate testing (Figure 3.13)

3.5.2 Testing Tools and Continuous Integration

As mentioned above, while the domain expert may in general support the idea of rigorous testing, they could lack the technical knowledge of the exact practices. To future-proof the eventuality that the developer may leave the team and the domain expert take over, or the software is forked by other potential users, the developer should set up a robust testing infrastructure as part of the initial development process. In other words, continuous integration (CI) become a necessity for research software.

Luckily, there are plenty of tools available these days. Our suggested go-to pairing is GitHub Actions and a testing framework for the programming language of choice. Some resources include:

- GitHub Actions: [Quickstart](#)

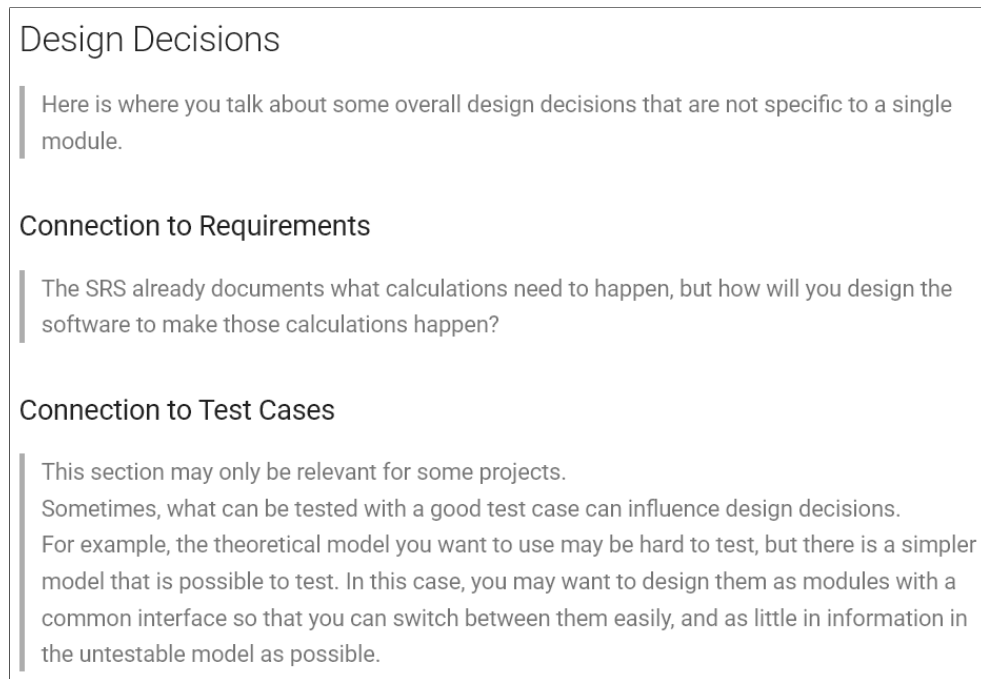


Figure 3.13: The Design Decisions section in our GitHub template, with hints for the developers.

- For Python: [pytest](#) and [pytest-cov](#) (for code coverage)
- For MATLAB: [Official GitHub Actions](#)
- For C++: [GoogleTest](#) or [doctest](#) for smaller projects

The actual test cases run with GitHub Actions may be a subset of all the cases, since for some projects, a full-scale system test may take too long or use more resources than what is feasible with GitHub Actions ([free tier limit](#)). The slower cases can be included to run on the developer’s local machine, or on a cluster, if that is part of the plan. Among the SynthEddy test cases, we marked some as “slow” so they are excluded from the [CI workflow](#).

The GitHub template of this thesis shows an example of how GitHub Actions can be set up for continuous integration, as it automatically builds the documentation

website when a commit is made to the “docs” directory. The related instructions inclined in our template is shown in Figure 3.14. For actual project code, we suggest having GitHub Action run on every pull requests to the main/master branch.

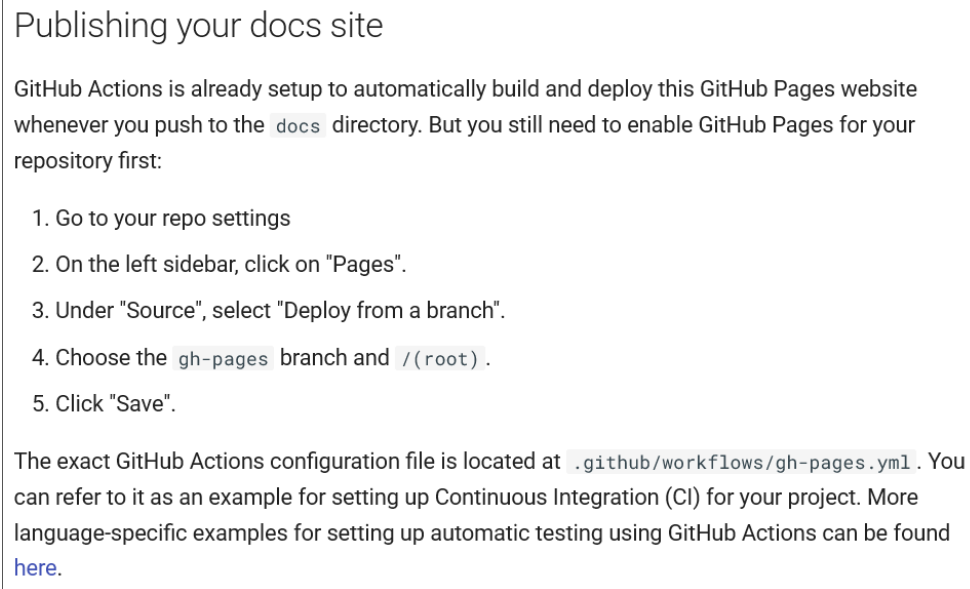


Figure 3.14: Instructions and further resources on how to set up GitHub Actions for continuous integration in our template.

3.6 Design and Implementation

The design document is the final piece of the puzzle that the developer should create before starting the actual initial implementation. In this section we discuss the documentations as well as some suggestions on the design itself.

3.6.1 Design Document

The original template by Smith [2] asked for two pieces of design documents: Module Guide (MG) and Module Interface Specification (MIS), both originally proposed by

Parnas [26][27]. The MG is a high-level view of the software structure and roles of each module, while the MIS is a detailed view of the input, output and transitions of the functions/methods in each module. In practice, our experience with SynthEddy found that while the MG provided a useful outline (Figure 3.15 as an example), the MIS was too much overhead for the developer. Moreover, the MIS was only relevant at the very beginning, since as the project progressed, the implementation diverged from the MIS, because rather than to first update the documentation, the first instinct of the developer was to modify the code.

| |
|--|
| 7.2.6 Shape Function Module (M7) Secrets: Shape function equations Services: Providing a list of shape functions to choose from. Allow setting currently active shape function and cutoff value to use. Implemented By: SynthEddy Type of Module: Abstract Object |
|--|

Figure 3.15: An example module as seen in the SynthEddy Module Guide.

For the domain expert, their involvement is already diminishing at this stage. They may need to review the module layout to get an overall understanding of the software design, but they are unlikely to go through a design as detailed as the MIS.

Thus, in the spirit of making a more lightweight process, we propose to drop the MIS entirely, and instead fully leverage the in-code documentation (such as Python [docstrings](#)) functionalities of modern programming languages and integrated development environments (IDEs). During the design phase, the developer should create code files for the planned modules, and populate the function headers along with their documentations. In a unified format, the function header documentations should include:

- A brief description of the function

- The parameters, types and default values (if applicable)
- The return and their types
- If in a class, the states that it modifies (transitions)
- The calculation that it performs, referring to requirements document if necessary

Since these files are now part of the formal documentation, they also include author and creation/modification date information at the top.

While some languages are not strongly typed, their documentations can include formatted type hints, which can be used by linters to catch potential bugs. Examples include [JavaScript in VSCode](#) and the [extension for Lua](#) by sumneko.

Figure 3.16 shows an example of such an in-code documentation in SynthEddy. Since it is good practice to include documentation of each function in the code anyway, expanding it to serve as part of the design document avoids redundancy and keeps the documentation up-to-date with the code. The result is that when the design phase ends, the developer already has a good code skeleton in-hand to start the implementation.

Another added benefit of this approach, which is becoming increasingly relevant these days, is that such detailed in-code documentation can provide better context to AI coding assistants such as [GitHub Copilot](#) and [Codeium](#), should the developer choose to use them. While we should always check and never blindly trust the suggestions/auto-completions from these tools, especially in the highly specialized field of research software, they can be a good starting point for the developer in the implementation phase. This is especially useful in the case when the domain expert

```

def sum_vel_chunk(
    centers: np.ndarray,
    sigma: np.ndarray,
    alpha: np.ndarray,
    x_coors: np.ndarray,
    y_coors: np.ndarray,
    z_coors: np.ndarray,
):
    """
    Calculate the velocity field due to each eddy within a chunk.

    Uses the intance model (IM1) in SRS as the core of the calculation.
    Choice of shape function is set in the shape_function module.

    Parameters
    -----
    centers : np.ndarray
        | Array of eddy centers.
    sigma : np.ndarray
        | Array of eddy length scales.
    alpha : np.ndarray
        | Array of eddy intensities.
    x_coors : np.ndarray
        | Array of x coordinates spanning the chunk.
    y_coors : np.ndarray
        | Array of y coordinates spanning the chunk.
    z_coors : np.ndarray
        | Array of z coordinates spanning the chunk.

    Returns
    -----
    np.ndarray
        | Array of velocity fluctuations due to each eddy within the chunk.
    """

```

Figure 3.16: A detailed docstring example in SynthEddy.

who is not that familiar with code takes up the responsibility of development.

3.6.2 Modularity and Performance Advices

A natural way of organizing the software is to mirror the models listed in the requirements document, i.e. if something has a physical significance, then we are inclined to construct it into a module as a class or object. This is inline with the object-oriented

programming (OOP) paradigm, and supports the idea of encapsulation and information hiding. Doing so comes with the benefit of making the software structure more intuitive to a different reader (including the domain expert), and also future proofs it to be more easily extendable. However, it may come with a performance cost. Not only does an overhead exist in creating and calling objects, but it may also be harder to optimize to work with existing libraries and tools.

This is a hard lesson learned in the SynthEddy project, as mentioned previously (Section 2.2.4). While the initial design (Figure 3.17) was very easy to understand, as each “eddy” was an encapsulated object within the “flow field”, faithfully recreating the physical system, the sheer number of eddies and the number of time each eddy need to be called made the software impractically slow and memory hungry.

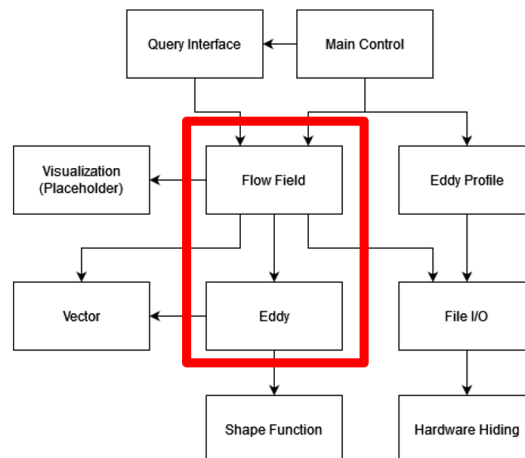


Figure 3.17: The use hierarchy showing the original modular design of SynthEddy, with each “eddy” being an object use by the “flow field”.

This is the reason why we ask questions like Q4.1-4.2 in the meeting. If we have determined that the scale of the problem is large enough that may lead to performance issues, then a proof of concept (PoC) may be necessary to further examine this aspect

and experiment with different design choices.

Our eventual choice is to break the perfect encapsulation, so that we can arrange eddy data and write heavily called functions in a way that supports broadcast operations by NumPy, a library that is optimized for vector/matrix operations. The eddy “class” then became the eddy “library” module that houses such functions without holding data itself. Similar measures may be needed if the use of such optimizable libraries or tools are required.

To salvage some “future-proofing merits” of this new design when the encapsulation is gone, we suggest writing example functions with detailed explanations so that future developers can take advantage of the optimization the same way you have done, such as what we have done for the shape functions in SynthEddy, shown in Figure 3.18.

3.7 User Guide and “Advertising”

As mentioned in the literature review (Section 1.4.1), the domain expert may lack the interest to commit much time and energy into the software development process, as they see little benefit beyond the immediate publication. This limits the accessibility of the software to other potential users, leading to less publicity of the software, and thus a downward spiral.

3.7.1 Publications

As developers, we understand that the “rite of passage” in the software world is not always the citation count, but also the number of users, GitHub stars and forks, etc.

```
def gaussian(dk, sigma):
    """Gaussian shape function"""
    return np.where(
        dk < cutoff,
        C * np.exp(-HALF_PI * dk**2),
        0
    )
    # The function is structured in this way so that it can be broadcasted by NumPy
    # over many eddies efficiently.
    # Mathematically, this function is equivalent to:

    #      ⌈ C * e^(-π/2 * dk^2), if dk < cutoff
    # q(dk) = ⌊
    #      ⌋ 0, elsewhere

    # where C = 3.6276, HALF_PI = π/2
    # These are defined as constants at the top of this file. You can use different
    # values or define new ones.
    # Use pre-calculated constants when possible because this function is called many
    # times.
    # It is faster not to recalculate these values every time.

    # Your inputs must include dk and sigma (sigma), even if some shape functions
    # may not use sigma.

    # You can choose what shape function and cutoff value to use in query arguments.
```

Figure 3.18: An example function with detailed explanation in SynthEddy.

While we can certainly communicate this to the domain expert, it is also prudent that we take some actions to satisfy their academic needs.

An inspiration can be drawn from some recent computer science publications, where the paper and the software were amalgamated into a single, inseparable entity. For example, in the GitHub repository of v2e [28], a software for converting conventional videos into “fake” event-based camera videos (which is needed for certain machine learning scenarios), the README file includes the following section:

Citation

If you use v2e, we appreciate a citation to the paper below. See the v2e home page for further background papers.

Y. Hu, S-C. Liu, and T. Delbruck. v2e: From Video Frames to Realistic DVS Events. In 2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW).

URL: <https://arxiv.org/abs/2006.07722>, 2021

To reproduce the experiments of the paper, please find [this repository](#).

This way, to reproduce or build upon the research is not only possible, but also encouraged, and the software is more likely to be used by other researchers in the field. On the other side, if someone wants to know more about the software and how it has been used in practice, they can refer to the paper. A win-win scenario for both the software usage and academic recognition can be achieved.

3.7.2 Quick Start

Beyond the domain experts, other potential users are not going to have the luxury of “first-party tech support” from the developer. To encourage potential users to try out the software, we can also draw some inspiration from the Self-Hosting community, where people host software on their own servers instead of relying on commercial services. A common theme among many successful self-hosting oriented software is to include a “Quick Start” section in the documentation, such as in the case of [Jellyfin](#) (a media server), [Nginx Proxy Manager](#) (a reverse proxy with GUI), and [Vaultwarden](#) (a password manager). In that community, a quick start is often provided in the form of a Docker Compose file, so potential users can easily spin up a containerized version of the software, often with the recommended configurations.

While Docker and Makefile are go-to choices for quick trials in the software realm, we should recognize that our domain expert and other potential users, should they

come from a different background, may not be familiar with command line tools or Linux environments. More likely, they have Windows or MacOS machines, and are more comfortable with “clicking” things.

For compiled languages like C++, this can be easily solved by providing a pre-compiled binary for download, but it is a bit more challenging for interpreted languages like Python or JavaScript, as they usually require a runtime environment (e.g. conda/pip, Node.js). In the case of SynthEddy, we provided a step-by-step guide to setting up a conda environment to run the software (Figure 3.19), which the domain expert, who had never used Python before, was able to successfully follow.

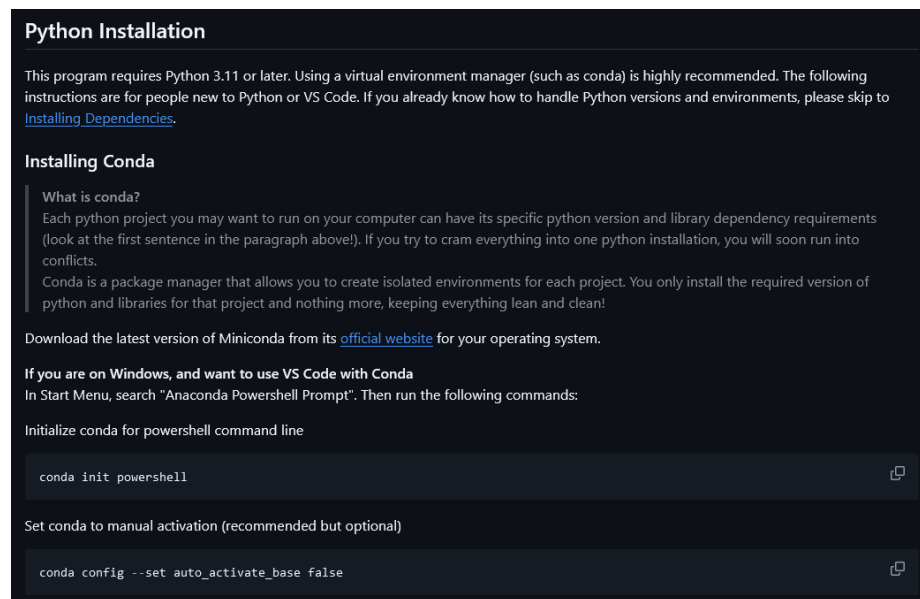


Figure 3.19: An excerpt of the installation guide of SynthEddy.

However, an even easier approach that we learned later (at the cost of some flexibility), is to package the entire environment with the release, and provide a batch script to run the software. This has been the case of many recent open-source AI tools written in Python for a wider, less technical target audience, such as [GPT-SoVITS](#)

(a text to speech tool) used by many video creators.

As a part of the quick start, we should also provide some example inputs that represent the typical use cases, but with expected resource usage within what is feasible on a personal computer.

After these preparations, the domain expert, as a typical user themselves, can be invited to test drive the software, with minimum interference from the developer, which helps validate and improve both the software usability and the documentation. This also help the developer avoid being trapped in the endless “tech-support” as mentioned in some developer’s feedback in the literature review (Section [1.4.1](#)).

Chapter 4

Preliminary Experiment Design

Given the timeframe of this thesis work, we are unable to conduct a full experiment to evaluate the effectiveness of the proposed framework. Instead, we will propose an experiment design that can be used to validate our proposal in the future if time and resources permit. We were, however, able to perform a preliminary validation of the proposed developer-domain expert meeting, which we will discuss more later in this chapter (Section [4.3.1](#)).

4.1 Overview

The experiments will attempt to answer the following Research Questions (RQ) regarding our proposed framework:

- RQ1: Does it ease communication between domain experts and developers, especially when there is a knowledge gap between the two parties?



- RQ2: Can the information gathering process capture most of the immediately relevant information needed for developing the first version of the software/Minimum Viable Product (MVP).
- RQ3: Can a successful piece of software can be developed without requiring significant time investment or prior software knowledge from the domain experts?
- RQ4: Does it minimize back-and-forth revisions/rewrites?

We will discuss the **Built-in Evaluation** (Section 4.2) as part of the framework itself to help us gauge its effectiveness. **Case Studies and Focus Groups** (Section 4.3) would be suitable for more in-depth evaluation, while **Mock Meetings** (Section 4.3.1) can be used to obtain more feedback if commitments to full case studies are hard to come by. Finally, we also consider **Comparison to Other Projects** (Section 4.4) that use different framework (or lack thereof) using more objective metrics like code churn rate and the implications.

4.2 Built-in Evaluation

The [GitHub repository template](#) allows us to embed some process evaluation questions (listed below) for the domain experts and developers that use our framework, as presented in Figure 4.1.


Built-in Evaluation: System Testing



The developers and domain experts are encouraged to evaluate the software development process at various phases. This may help them catch any misunderstandings or shortcomings not brought up with issue-reporting mechanisms. It also helps us to improve our development framework.

To developers: You may want to hide this page after the software is released (do not delete the file from the repository).

Questions for Domain Experts

 **Tip**

Please edit this page (upper right corner) to add your answers to the questions below.

BIQ2.1

If the software passes all the system tests listed, would you be confident to use it in your research and publications?

- yes/no

Figure 4.1: Example Built-in Evaluation page at the end of testing documents in the GitHub template.

The questions are asked at the end of the major milestones in the development process of the minimum viable product (MVP), after any issues about the relevant writings have been raised. Answering “no” to any of the yes/no questions indicates a major problem; a major problem means a failure of our process, because the problem cannot be quickly addressed with individual issues, and may require additional time to be allotted from both parties for further discussions, a situation that we hope to avoid.

Even without considering the evaluation of our framework, these questions provide

immediate feedback to the project teams, which should help them improve their software development process. For any open-source project that adopts our framework, the response is public and can then be polled and used to evaluate the effectiveness of our framework. Since the information is public, the researchers likely do not need ethics approval.

The Built-In Questions (BIQ) are as follows. The Research Questions (RQ, as in Section 4.1) that each group of BIQ traces to are listed in parentheses:

- At the end of the information gathering meeting note (RQ1, RQ2):
 - For the domain experts:
 - * BIQ1.1: Based on all the questions asked and information you have provided, are you confident that the developer will have a good understanding of the problem domain and the model to implement the software? (yes/no)
 - * BIQ1.2: (optional) If you think there are major information gaps that you wish to discuss further, please list them as bullet points.
 - For the developers:
 - * BIQ1.3: Based on all the information you have collected, do you have a good picture of what the domain expert wants and your work ahead? (yes/no)
 - * BIQ1.4: (optional) If there are aspects not covered in the template that you need to know more about, please list them as bullet points.
- At the end of the requirements document (RQ2):
 - For the domain experts:

- * BIQ2.1: Do you think this requirements document is a good representation of what you want the software to do? (yes/no)
 - * BIQ2.2: (optional) If there are any major misunderstanding or omissions, please list them as bullet points.
- At the end of the system testing documentation (RQ2):
 - For the domain experts:
 - * BIQ3.1: If the software passes all the system tests listed, would you be confident to use it in your research and publications? (yes/no)
 - * BIQ3.2: (optional) If you do not think the tests cover all the important aspects, please list any concerns as bullet points.
 - After the minimum viable product is implemented, at the end of the user guide (RQ3):
 - For the domain experts:
 - * BIQ4.1: Were you able to get the software up and running with only the resources provided in the user guide? (yes/no)
 - * BIQ4.2: (optional) If you had to look for additional resources to get the software running, or encountered major obstacles, please list them as bullet points. These may be...
 - I had to Google for...
 - I asked for help from the developer to...
 - It was not working on my computer at all because...

Note that there are no built-in questions about the design document, as we expect the involvement of the domain experts to be minimal at this stage.

We will collect the responses from all open-source projects that adopt our framework. For each question, the ratio of “yes” to total responses will be calculated, as well as the number of bullet points listed for the optional questions if the previous question was answered “no”. A high ratio of “yes” from any question indicates that the likelihood of encountering a major hiccup at that stage is low. If the number of bullet points listed is high, it may indicate that critical information is missing from the template, and we may need to identify common themes in the responses to improve our template.

4.2.1 Threat to Validity

Although we cannot require all project teams that come across and adopt our framework to answer these questions (unless they are specifically recruited for our experiment, which we will discuss later), we hope that the usefulness of these questions will encourage the project teams to answer them. But nonetheless, the response is entirely voluntary. In addition to a potentially low response rate, we run the risk of a biased result. To save time, the domain experts and developers may be more inclined to answer “yes” to the yes/no questions so that they can move on to the next phase of their projects, unless they feel the problem is severe enough to warrant a “no” answer and more detailed feedback.

4.3 Case Study and Focus Group

More focused efforts can be directed toward a few selected projects that are willing to participate in a more in-depth evaluation. For these projects, we will be in direct communication with the project teams, similar to how the series of case studies were conducted by Carver et al. [22]. One such opportunity find candidate projects is on the CAS 741 course (Development of Scientific Computing Software) at McMaster University, with additional ethics approval. A pilot study can be conducted with limited project teams, before expanding to a large data collection exercise.

Our goal is to follow each project team from the initial contact between the domain expert and the developer, to the release of MVP, and possibly further versions with added features or improvements. We will collect incremental feedback from both parties via a series of questions (aside from the built-in questions) to both gauge the effectiveness of our framework, and to better understand the dynamics and interactions between the domain experts and developers in general. The full list of questions will be detailed in Section 4.3.2, which covers every phase of the development process. We also propose a shortened version of such experiment (Section 4.3.1) that focus on the beginning (information gathering) to potentially gather data from larger pool of participants.

4.3.1 Mock Meeting

We realize that participating project teams for our case studies may be hard to come by. Unlike the studies by Carver et al. [22] where the researchers contacted already formed, mature teams, we need to target potential candidates that have not yet started the software development work, and ask them to commit months of their

effort to use our framework. The willingness by the domain experts, availability of the developers, and the timing of the project must all align to make this work.

To work around this challenge, we first propose a series of mock information gathering meetings according to practices detailed in Section 3.2. This meeting is a new process introduced by our framework and not present in previous works. We believe the information gathering meeting is a critical step in the success of a research software project.

Despite calling them “mock meetings,” the participants will still be real domain experts with problems at hand, and real developers who are interested in writing research software. The only difference is that (in most cases) they will not proceed to actually develop the software after the meeting, as it is much easier to recruit participants who can spare just a few hours of their time to prepare and conduct the meeting, review the meeting notes, answer BIQ1.1-1.4, and have a discussion with the organizers about their experience.

This does not preclude the possibilities that some participants, with their schedules permitting, will be interested in continuing the project (or just to finish the requirements document). In which case, they will be asked to participate the full case study until at least the release of an MVP.

Trial Run and Threat to Validity

This mock meeting format has already been tested by the author of this thesis, acting as the domain expert of his previous trajectory research project [4]. The mock developer, who did not have previous knowledge of the meeting template, was asked to direct the meeting and complete the meeting notes. Due to time constraints, the

meeting lasted less than an hour with most of the questions covered. The mock developer reported confidence in gathering the necessary information for the project.

This test run did come with the threat to validity that, while the author pretended to let the other party direct the meeting, he nevertheless was familiar with the questions and what “good answers” would look like. The mock developer also identified that the author’s above average ability to explain technical details in layman’s terms (as he had previous experience in making educational videos) was a significant factor in the success of the meeting. This makes us more interested in conducting more mock meetings with a variety of pairings between real domain experts and developers.

4.3.2 Full Experiment Questions

The following is a list of questions for the full-scale experiment. Part 1 (Team Background) and 2 (Information Gathering) are also applicable to the mock meetings. Some of these questions are inspired by [29]. The brackets after the question numbers indicate if the question is intended for only the domain experts (DE), only the developers (Dev), or otherwise both. These questions are supplementary to the Built-In Questions in the framework (Section 4.2), instead of replacing them.

Team Background

These questions are to be asked at the very beginning of the experiment. Besides providing demographic information for the experiment, they help us identify if a knowledge gap exists between the domain expert and the developer in a team (RQ1), and the level software engineering expertise from the domain expert (RQ3).

- FEQ1.1: What is your current position/title/degrees?

- FEQ1.2 [DE]: What is your research domain?
- FEQ1.3 [DE]: How long have you been working on this research?
- FEQ1.3 [DE]: Did you have any related publications or other substantial work prior to this point?
- FEQ1.4 [Dev]: What prior knowledge/experience do you have in the research domain or its broader field of study (biology, physics, mechanical engineering...), if any?
- FEQ1.5: What software developing/programming experience do you have, if any? As in..
 - Previous projects and their size (lines of code, number of contributors, etc.)
 - Programming Languages and years of experience.
- FEQ1.6: Are you familiar with any software development tools (e.g. Git, GitHub, Docker, etc.)?
- FEQ1.7: Are you familiar with any general models or practices in software development (e.g. waterfall, agile, etc.)?
- FEQ1.8: In the project participating our experiment, how many domain experts and developers are there? Are there any overlaps?

Information Gathering

These questions are to be asked sometime following the initial technical meeting between the domain experts and developers, after any lingering issues are addressed

regarding the meeting notes. They allow us to examine if the communication is successful (RQ1) and the amount of information passed on to the developer (RQ2).

- FEQ2.1 [DE]: What challenges did you encounter when trying to convey your research problem to the developer, if any?
- FEQ2.2 [Dev]: What challenges did you encounter when trying to understand the needs of the domain expert and recording them in the meeting notes, if any?
- FEQ2.3 [Dev]: After following our meeting template, are there any aspects that you think are missing/lacking?
- FEQ2.4: Did you face any obstacles raising and addressing issues regarding the meeting notes after the meeting?
- FEQ2.5: What is your overall opinion on the information gathering meeting + meeting note format? Do you think it saves time, or can it be improved?

Software Requirements and Tests

After the requirements document and system tests are drafted by the developer and reviewed by the domain expert, the following questions shall be asked. These will help us evaluate if the information gathered in the previous phase is indeed sufficient for the developer to proceed with the software development (RQ2), or if the domain expert and developer still need more back and forth before coming to a mutually agreed upon document (RQ4).

- FEQ3.1 [Dev]: Was the information gathered previously sufficient for you to draft the requirements document and system tests? If not, what additional information did you request from the domain expert?

- FEQ3.2 [Dev]: Was there any significant piece of information missing due to the domain expert not being able to realistically provide it (research not done yet, lack of data, etc.)? Were you able to work around it (e.g. document assumptions/educated guesses, leaving as inputs, etc.)?
- FEQ3.3 [Dev]: What disagreements between you and the domain expert ran into when drafting the requirements document and system tests, if any? Did you find common ground and how?
- FEQ3.4 [DE]: At this stage, did you still need to spend any significant amount of time helping the developer to revise the requirements document or coming up with test cases? If so, what were the major issues?
- FEQ3.5 [DE]: Do you think these documents are a good format to communicate and arrive on the same page with the developer in terms of what to build and how to test it? If not, what format would you prefer or what changes would you suggest?

Design and Implementation

For the sake of the experiments (RQ3), we should clarify that the domain experts involvement is entirely optional at this stage until a release is ready for them to trial. The questions are more focused on the developers, if they can smoothly proceed with the software development (RQ4), but we will still record the involvement of the domain expert, if any.

- FEQ4.1 [Dev]: Were you able to design and implement the software solely based on everything you have so far, or did you need to consult the domain expert for

additional information? If so, about what?

- FEQ4.2 [Dev]: Did you ever need to go back and revise the requirements document or system tests during design or implementation? If so, what were the issues?
- FEQ4.3 [Dev]: Did you ever need to significantly change the software design after you started the implementation? If so, what were the issues?
- FEQ4.4 [DE]: Were you in any way involved in the design and implementation process (e.g. read the design document, discussed architecture with the developer, etc.)? If so, why did the necessity arise and did you find it helpful?

Released Product

After a Minimum Viable Product (MVP) is released accepted by the domain expert, the following questions shall be asked. These questions evaluate if the software is ultimately considered “successful” (RQ3), and whether there were major issues needed to be ironed out (RQ4) before that point.

- FEQ5.1 [DE]: Were you satisfied with the first version of the software provided to you, or did you ask for more revisions before accepting it as an MVP? What were the issues if any?
- FEQ5.2 [Dev]: Did the domain expert ask for any revisions before accepting the MVP? If so, were they minor changes or did you need to significantly rework the software or even the requirements document?
- FEQ5.3 [DE]: Is the released software true to your requirements such that it will aid your research? If not, what deviates from your expectations?

- FEQ5.4 [DE]: Did you find the software easy to use? What would you suggest improving?
- FEQ5.5 [DE]: Do you think another researcher with similar knowledge as you, but who is otherwise not affiliated with you or the developer, would be able to use the software solely based on the user guide provided?

Sustainability

These questions are about what happens after the MVP is released. Given what possible timeframe that the experiment may run, we might not be able to touch on this phase.

- FEQ 6.1 [DE]: What additional features or improvements did you request from the developer after the MVP was released? Were they implemented to your satisfaction?
- FEQ 6.2 [Dev]: When additional request came, was the whole software, from requirements to design to implementation, more or less prepared for the changes? If not, what major rework was needed?
- FEQ 6.3: Did the software gain any additional recognition beyond the domain expert? If so, what feedback was received, if any?

Overall

These questions are to be asked at the end of the experiment to gain some summarizing insights from the participants.

- FEQ7.1: What was the biggest challenge during the whole development process?
What would you change to remove or mitigate the obstacles?
- FEQ7.2: What is your opinion on various pieces of documents? Do you think they are helpful or was the overhead too onerous?
- FEQ7.3: What is your overall opinion on the framework? Do you think it saves time or there can be improvements/more effective ways?
- FEQ7.4: Would you use this framework (with some improvements if necessary) again for future projects or recommend it to others? Why or why not?

4.3.3 Extended Experiment

If our framework and experiment catches enough interests from potential participants, we can expand the experiment with controlled variables to evaluate the effectiveness of specific parts within our framework. For example, some groups may be asked to meet in a more ad-hoc manner without using our information gathering meeting template. Other groups may be asked to skip the requirements document (or make a much simpler version without formally documenting the models), or the Module Guides, etc. This would allow us to further determine if any of our specific practices are indeed effective, or if they are overheads without clear benefits and can be simplified or removed.

4.4 Comparison to Other Projects

While it can be hard to get a straightforward measurement of the software quality, especially for research software with a relatively small user base, we can still try to compare our process to software developed by traditional means. To compare projects, we propose code churn rate, which is the ratio of lines modified/deleted to total lines written in a codebase. Sometimes this metric is bracketed within a certain time frame or between two builds to investigate if there are frequent rewrites. A low churn rate between requirements being drawn up and the MVP being accepted by the domain expert can indicate that the requirements are well understood to begin with, and there is little back and forth or communication breakdowns between the domain expert and the developer.

It has been shown that metrics such as code churn may be used when software quality cannot be directly measured [30]. One study has found that high cumulative code churn negatively impacts maintainability [31]. This is especially relevant for us, to prepare for the eventuality that the original developer may leave the project, or if the project is adapted by another team in the future.

The GitHub template includes a [script](#) that can quickly calculate the churn rate with instructions provided. It can also easily be used on other open-source codebases. For the experiment, we will compare the churn rate for the first release version of the software that uses our framework, the previous framework proposed by Smith et al. [3] that we aim to improve upon, and more general research software projects that do not use any framework. We wish to examine if there is a statistically significant difference (reduction) in the churn rate (RQ4) between different development processes.

For projects with texted based documentations (e.g. Markdown and LaTeX) in

their repository, we can also expand the churn rate comparison to the requirements and design documents. However, such projects would be harder to come by, as not all projects are documented as rigorously during their development.

Chapter 5

Conclusion

In this thesis, we examined the pain points existing in today’s research software development process as shown by several previous studies. These include:

- Lack of software engineering knowledge by the domain experts (researchers), especially areas like testing and project management [1] [8].
- “Feature creep” as the research evolves and hard to define requirements [8][9][6].
- Poor documentations and little interest in making the software sustainable, due to lack of recognition and outside user feedback [8].
- Knowledge gap and communication breakdown between the developer and the domain expert, making it difficult for the developer to grasp the requirements [9].

As part of the work toward this thesis, a substantial piece of research software, namely SynthEddy, was presented and discussed. Working with a domain expert in turbulent flow research, we developed SynthEddy and learned some valuable lessons

in the process, some echoing the aforementioned pain points. For example, the early communications between the developer and the domain expert (who was not that well-versed in software engineering) focused more on getting the theory right. This gave the developer a false understanding of what would be the typical problem scale when in use. This information was never well corroborated, leading to initial performance issues and major redesigns.

These experiences led us to propose an improved research software development framework in the form of a GitHub template, with its guiding practices discussed in Chapter 3, which in summary are:

- Taking a pragmatic stance overall, recognizing that the research is evolving and not everything can be known initially.
- Focusing on the Minimum Viable Product (MVP), and gather the right information that would directly contribute to the MVP from the domain expert, including:
 - The problem domain and theory itself.
 - Scale of the problem and typical use cases.
 - Availability of test cases and data.
- The above allows us to plan for decreasing involvement of the domain expert as development progresses to avoid taking too much of their time. They would be more involve in the requirements phase and less so in the design and implementation phases.
- Spec for the exploratory nature of the research, and design around available test cases.

- Measures to ease accessibility for potential future users, and gain more academic recognition.

This framework, tailored toward smaller research teams with limited resources, focuses on facilitating communication between the developer and the domain expert, potentially arriving at requirements and minimum viable product faster, and cutting down overhead in the development process, which would otherwise discourage the use of a document driven approach. Possible experiments to evaluate our framework are laid out in Chapter 4.

5.1 Potential Adoption

We do recognize that the adoption of such a framework is entirely voluntary in the present days, and most likely out of internal motivations instead of any external academic needs. The current academic landscape usually does not require the release and scrutiny of any software that aids the research as part of the publication process. One may argue that this is a bizarre reality as more and more research projects outside of computer and software realm are becoming increasingly dependent on software tools to produce their results, raising questions of their reproducibility. The act of publishing a research paper itself (as opposed to keeping the knowledge proprietary) is also more or less inline with the spirit of open-source software, with peer-review and future works similar to reporting bugs and forking in the software world. Nonetheless, unless there is a big push from the academic community to change the status quo, the adoption of such a framework may be slow in the foreseeable future.

We do believe there exists a particular group of stakeholders that would benefit from the work presented in this thesis: research organizations, such as laboratories and

university departments. The reason is that individual researchers, such as graduate students, come and go, but the tools they use for their works, namely the research software, can be relevant for years. In the case of the author previous trajectory research project, if there were an agreed-upon standard for him to develop, document and pass on the software, later researchers could have a much easier time to adapt it to their needs, instead of having to contact the author years later, with the risk of not getting a response. Thus, we implore such organizational stakeholders to take a deeper look at either ours or similar frameworks, and consider adopting them as a standard practice for research projects with substantial software components.

5.2 Future Work

The most direct future work coming from this thesis is to evaluate our proposed framework by recruiting real research project teams and following their journey through the software development process. This would allow us to gather real data on the effectiveness of our framework, and potentially lead to a more refined version of it.

While this thesis focuses on the developer and domain expert pairing scenario, another possible direction is to explore more on the case where the developer is the domain expert themselves. This is increasingly more relevant these days as rapid advancement in tools to assist software development has made it easier than ever for people with diverse backgrounds to dip their toes into software development to suit their specific needs. If their goal is to develop sustainable software, what would be the most effective practices with the least overhead? This need to take into consideration that they may be developing and learning on an ad-hoc basis at the same time. It could be a waterfall approach, or an agile one, or something even less rigid, as there

is no longer a separation between the developer and the domain expert.

Bibliography

- [1] J. Hannay, C. MacLeod, J. Singer, H. P. Langtangen, D. Pfahl, and G. Wilson, “How do scientists develop and use scientific software?” *Software Engineering for Computational Science and Engineering, ICSE Workshop on*, vol. 0, pp. 1–8, 05 2009.
- [2] S. Smith, “Software engineering for science,” in *Software Engineering for Science*, J. C. Carver, N. P. Chue, and G. K. Thiruvathukal, Eds. New York: CRC Press, 2016, ch. 1, pp. 1–36.
- [3] S. Smith, C. W. Schankula, L. Dutton, and C. K. Anand, “A software engineering capstone course facilitated by github templates,” *arXiv preprint arXiv:2410.12114*, 2024.
- [4] A. P. Du, “A comparative study between 6 degree-of-freedom trajectory model and modified point mass trajectory model of spinning projectiles,” Master’s thesis, Embry-Riddle Aeronautical University, 5 2021. [Online]. Available: <https://commons.erau.edu/edt/594/>
- [5] T. Storer, “Bridging the chasm: A survey of software engineering practice in

- scientific programming,” *ACM Computing Surveys (CSUR)*, vol. 50, no. 4, pp. 1–32, 2017.
- [6] S. Killcoyne and J. Boyle, “Managing chaos: Lessons learned developing software in the life sciences,” *Computing in Science & Engineering*, vol. 11, no. 6, pp. 20–29, 2009.
- [7] J. Segal, “End-user software engineering and professional end-user developers.” Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2007.
- [8] G. Pinto, I. Wiese, and L. F. Dias, “How do scientists develop scientific software? an external replication,” in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 582–591.
- [9] J. Segal and C. Morris, “Developing scientific software,” *IEEE Software*, vol. 25, no. 4, pp. 18–20, 2008.
- [10] I. Wiese, I. Polato, and G. Pinto, “Naming the Pain in Developing Scientific Software,” *IEEE Software*, vol. 37, no. 04, pp. 75–82, Jul. 2020. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/MS.2019.2899838>
- [11] R. F. Da Silva, H. Casanova, K. Chard, I. Altintas, R. M. Badia, B. Balis, T. Coleman, F. Coppens, F. Di Natale, B. Enders *et al.*, “A community roadmap for scientific workflows research and development,” in *2021 IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS)*. IEEE, 2021, pp. 81–90.
- [12] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

- [13] R. F. da Silva, H. Casanova, K. Chard, I. Altintas, R. M. Badia, B. Balis, T. Coleman, F. Coppens, F. Di Natale, B. Enders, T. Fahringer, R. Filgueira, G. Fursin, D. Garijo, C. Goble, D. Howell, S. Jha, D. S. Katz, D. Laney, U. Leser, M. Malawski, K. Mehta, L. Pottier, J. Ozik, J. L. Peterson, L. Ramakrishnan, S. Soiland-Reyes, D. Thain, and M. Wolf, “A community roadmap for scientific workflows research and development,” in *2021 IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS)*, 2021, pp. 81–90.
- [14] S. Smith, L. Lai, and R. Khédri, “Requirements analysis for engineering computation: A systematic approach for improving reliability,” *Reliable Computing*, vol. 13, pp. 83–107, 02 2007.
- [15] C. Liu, “Bridge chloride exposure predictor,” 9 2024. [Online]. Available: <https://github.com/CynthiaLiu0805/BridgeChlorideExposurePredictor>
- [16] J. Carette, S. Smith, D. Szymczak, J. Balaci, B. MacLachlan, M. Niazi, S. Crawford, D. Scime, D. Chen, A. Hunt, T.-Y. Wu, M. Bilal, and B. Bosman, “Drasil.” [Online]. Available: <https://jacquescurette.github.io/Drasil/>
- [17] S. Smith, “Template for capstone projects.” [Online]. Available: <https://github.com/smiths/capTemplate>
- [18] T. Christie, “MkDocs - Project documentation with Markdown,” 2014. [Online]. Available: <https://www.mkdocs.org/>
- [19] M. Donath, “Material for MkDocs - A powerful documentation framework on top of MkDocs,” 2016. [Online]. Available: <https://squidfunk.github.io/mkdocs-material/>

- [20] A. P. Du, N. Holyev, and S. Smith, “SynthEddy - Synthetic eddy for turbulent flow simulation,” 12 2024. [Online]. Available: <https://github.com/omltcat/turbulent-flow>
- [21] R. Poletto, T. Craft, and A. Revell, “A new divergence free synthetic eddy method for the reproduction of inlet flow conditions for LES,” *Flow, Turbulence and Combustion*, vol. 91, pp. 1–21, 10 2013.
- [22] J. C. Carver, R. P. Kendall, S. E. Squires, and D. E. Post, “Software development environments for scientific and engineering software: A series of case studies,” in *29th International Conference on Software Engineering (ICSE’07)*, 2007, pp. 550–559.
- [23] S. Balaji and M. S. Murugaiyan, “Waterfall vs. v-model vs. agile: A comparative study on sdlc,” *International Journal of Information Technology and Business Management*, vol. 2, no. 1, pp. 26–30, 2012.
- [24] U. Kanewala and T. Yueh Chen, “Metamorphic testing: A simple yet effective approach for testing scientific software,” pp. 66–72, 2018.
- [25] C. Liu, “Requirements validation process for research software: A case study with bridge chloride exposure prediction software,” 2024.
- [26] D. L. Parnas, “On the criteria to be used in decomposing systems into modules,” *Commun. ACM*, vol. 15, no. 12, p. 1053–1058, Dec. 1972. [Online]. Available: <https://doi.org/10.1145/361598.361623>
- [27] —, “On the criteria to be used in decomposing systems into modules,”

- Commun. ACM*, vol. 15, no. 12, p. 1053–1058, Dec. 1972. [Online]. Available: <https://doi.org/10.1145/361598.361623>
- [28] Y. Hu, S. C. Liu, and T. Delbruck, “v2e: From video frames to realistic DVS events,” in *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. IEEE, 2021. [Online]. Available: <http://arxiv.org/abs/2006.07722>
- [29] A. Dong, “Questions to developers,” 2020. [Online]. Available: <https://github.com/smiths/AIMSS/blob/master/StateOfPractice/Methodology/Questions%20to%20Developers.pdf>
- [30] J. Munson and S. Elbaum, “Code churn: a measure for estimating the impact of code change,” in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, 1998, pp. 24–31.
- [31] C. Faragó, P. Hegedűs, and R. Ferenc, “Cumulative code churn: Impact on maintainability,” in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2015, pp. 141–150.
- [32] H. Martinsson and V. Svanqvist, “Technology stack selection: Guidelines for organisations with multiple development teams,” 2022.