An EXPLORATION *and* ADVANCEMENT *of* Paradigms *for* PREDICTIVE CONTROL SYSTEMS Software Design and Construction

A Thesis Proposal

by TIMOTHY A.V. TEATRO

UNIVERSITY OF ONTARIO INSTITUTE OF TECHNOLOGY

Abstract

A course of research is proposed which advances theory and practice in predictive control systems software. The benefits of modern software techniques, such as functional and functional-reactive design patterns will be evaluated in the development of a reusable framework for parallel, distributable, research-grade model predictive control systems in modern C++. The work will be validated by the production of novel research in the field of control systems theory. Prior work which lead to this, is described along with a review of relevant literature and theoretical background in optimal control.

Contents

1 Introduction

... we must avoid the temptation to think of software as simply the language of the implementation. Control code ... is a dynamic system. It has an internal state, responds to inputs, and produces outputs. It has time scales, transients, and saturation points. It can also be adaptive and distributed. ... if we take this software dynamic system and couple it to the plant dynamics through the sensor and actuator dynamics, we have a composite system whose properties cannot be decided from the subsystems in isolation.

HELEN GILL AND JOHN BAY
Software-Enabled Control: Information Technology for Dynamical Systems [1, p. 4]

THE EXPERTISE of a control systems engineer is, of course, in the design of mechanisms to control machines to desired behaviour. However, that engineer will likely occupy most of his or her time on implementation, rather than the design. Very often, that implementation takes the form of computer software and as the infrastructure of the first-world leans ever more heavily on software driven electronics, our profession has spawned an entire branch dedicated to the engineering of software. But in academia, there is rarely much attention devoted to the engineering of *control systems software* in particular.

The contribution of my proposed thesis is threefold. One of my major interests is in developing software engineering patterns using modern programming paradigms for crafting readable, reliable, testable, correct and ***performant*** predictive control software. A significant contribution to the field will be:

(C1) the software design description—the general description of the software components and their interactions and the learned design principles.

Those designs will be implemented in a portable and reusable framework, with a contribution represented in:

(C2) ***software capital***, published through open source licensing with thorough documentation.

Definition 1. ***Software capital***, as coined by Dean Zarras in 1996 is—the cumulative technology that can be re-deployed to new situations. (See `http://goo.gl/y2EHPs`)

My primary motivation for developing the software is to enable novel research in the field of *optimal control* systems. Before now, the majority of my research has focused on aspects of optimal control theory, and software development has been a mere means for those activities. However, the challenge of implementing these systems well has stimulated my attention in the engineering of the software itself. Theoretical research in optimal control remains a paramount goal. Therefore, a continued and concurrent contribution of my research will be:

(C3) novel theoretical research in the field of optimal control systems.

In the remainder of this section, I will motivate and outline the scope of the work. I will demonstrate that the time is ripe for this research, as we approach a critical and opportune phase in our technological evolution.

1.1 Motivation

In the opening paragraphs of this chapter, I enumerated my proposed contributions by discussing the software I wanted to design (contributions (C1) and (C2)), in order to facilitate research in control systems (contribution (C3)). However, when discussing the motivation, the logical ordering is reversed. The software is what enables the control systems research, but progress in control systems is the primary motivator. So, I begin with a discussion of some directions of novel control systems research.

Definition 2. A software ***contract*** is a formal, precise and verifiable interface specification for software components, which extend the ordinary definition of abstract data types with preconditions, postconditions and invariants. The term is a conceptual metaphor with the conditions and obligations of business contracts. (See Wikipedia entry for *design by contract*, at `https://goo.gl/nfOWRN`)

American cognitive scientist and philosopher Daniel C. Dennett, in his book *Kinds of Minds* [2] describes the workings of the human mind:

> The task of the mind is to produce future, as the poet Paul Valéry once put it. A mind is fundamentally an anticipator, an expectation-generator. It mines the present for clues, which it refines with the help of the materials it has saved from the past, turning them into anticipations of the future. And then it acts, rationally, on the basis of those hard-won anticipations.

So, what will happen if you run down the sidewalk with a full bucket of water? *Your feet will get wet,* of course; the water will slosh out.[1] You have probably never been told that fact. You are not likely to find that fact through any reasonable amount of web-searching or inspection of literature. That fact was generated in your mind through a multiphysical simulation based on rules, intuitions, and information learned from past experiences.

[1] I borrow this example from Prof. Patrick Winston's lectures in artificial intelligence at MIT.

Notice how the models running in your brain that inform your intuition in the bucket example are different from the models you might use to make decisions about your mortgage. Your mind requires multiple models in order to make the decisions across multiple domains of experience.

Complex dynamical systems, such as those of mobile robots, have state spaces which are too varied and unpredictable to analyse thoroughly. Therefore, we have no ways to guarantee performance in a sufficiently wide set of circumstances. However, we know that resiliency can be achieved by just having a controller that processes information in the right way, refines models and adapts to use them. We know this because it is what we do. It is what you are doing right now—and what I am doing. Our brains are self-modelling, self-aware control systems for human apes, that are unrivalled in scope and resiliency by the artificial control systems of today. What if a robot could reason about its frame and its environment in the same way that we can?

I do not claim to know that brain-like control systems are the *only* way to achieve the sort of resiliency we desire from our machines. But animals—particularly the human animal—is the best example we know of that behaviour. Until we invent or discover something better, the best we can do is aspire to the standard set in our own biology.

Brain-like control systems are not the topic of the proposed work, but it is an important part of the motivation. A significant technology gap separates us from any plausible attempt at brain-like control systems. In the meantime, there is a lot to be understood about self-modelling control systems.

At the current state of the art, self-modelling mobile robots have been given the ability to learn how to move after suffering physical trauma, such as the loss of a limb [3]. Self modelling robots have even been given a primitive sense of ethics with regard to protecting the welfare of human beings in their environment [4]. This is a very young field of research that has been historically constrained by the computationally expensive nature of the calculations involved with modelling, and the search for control actions that produce desired simulation results.

My personal history in physical science biases me toward optimal control techniques, which are as mathematically beautiful as they are effective. The underlying structure of the optimal control problem is the purchase upon which all of modern physics is based. The most widely employed of optimal control techniques is Linear or Nonlinear Model Predictive Control (N/MPC). Despite the fact that optimal control techniques are not easily wielded by the novice, NMPC is a remarkably intuitive way to control nonlinear Multiple Input Multiple Output (MIMO) systems—especially with state or control space constraints—making them very popular.

The application of machine learning techniques to NMPC is a fairly recent development and there is a lot to explore. Practical results are extremely promising. For example, see Reference [5], where in researchers at University of Toronto describe automated, learning machines capable of strikingly precise manoeuvres over learned paths.

Software to make those sorts of techniques accessible to mainstream researchers or practitioners is not publicly available. I want to design a software framework that makes those sorts of algorithms very natural to develop and implement. I want to facilitate rapid development while constraining the space of possible designs as little as possible. More concisely, I want to minimise development effort while maximising flexibility—something much easier said than done.

It is widely understood that the major problems faced in the software engineering industry today, centre around parallelism and complexity management. Designing control systems to rival the human mind will require collision with both of those issues. My focus on software design and construction techniques will encompass our current best solutions to both issues.

The benefits of building a productive software enterprise upon a durable, high quality, generically programmed codebase are now widely recognised in the software industry. More to the point, the dangers and costs of not doing so, are changing corporate attitudes toward software assets. Companies such as Google, Apple, Microsoft, facebook and Bloomberg have taken pains to develop rules and guides for their developers and spent millions of dollars on training, with the focus of improving software quality. Books have risen to popular status in software engineering, describing techniques for developing clean and reliable code [6]–[9, for examples]. Because so much of the world infrastructure is built upon legacy code, a great deal of attention is also devoted to safely and reliably modernising existing code to conform to reflect modern principles and values [10], [11].

Progress in processor clock speeds has plateaued around the 2.5 GHz to 3.5 GHz range. In place of clock-speed, hardware progress has directed toward multiplication of cores within a single system. At the time of this writing, it is not uncommon for consumer grade laptops to have 4 to 8 Central Processing Unit (CPU) cores, with up to 16 thread processors, not including Graphics Processing Unit (GPU) cores. Modern supercomputing hardware is making use of GPUs with thousands of thread-processors. This state of hardware has created demand for parallel software design at a scope and scale that would have been unthinkable just twenty years ago.

Mobile computing, embedded and large-scale computing markets both maintain a high value on efficient software. Simultaneously, the need to distribute computation across many cores places importance on good design and code which is easy to reason about. (If you've ever tried to debug parallel software, you might understand the potential to lose thousands of man-hours to an obscure race condition.)

In [12, §5.2.1], Wills, Kannan, Sander, *et al.* summarise some of the limitations of controls software design practices. They emphasise one that I find particularly pertinent:

> The modules in a typical modern control system tend to be tightly coupled; that is, there are direct communication links between the various components that share information. This can make the system faster but may make changes to the system harder to achieve, particularly with respect to the constraints imposed by the data transfer protocol, as mentioned above. With the advent of ever-faster processors, tight coupling is not necessary to gain efficiency, particularly at the expense of reduced ability to update the control system quickly and easily. It would be much easier to make changes to the system if its components were less coupled and if functionally similar components could be easily interchanged in a "plug-and-play" fashion. This requires an open software infrastructure that supports a component-based control system design.

Tight coupling, objects sharing state, and functions and methods which depend on shared state are the root of all problems in parallel programming. If two routines enter the same piece of memory at the same time, one with the intent to mutate the memory, then a race condition is created. The execution path of the code will now depend on whether the memory is read before or after the mutation. A common solution is to lock the memory and to allow only sequential access to concurrent processes. This produces a bottleneck which can eliminate the benefits of concurrent execution, or even make it slower than an equivalent sequential routine.

Even in sequential software, the concept of a program's state becomes extremely complicated if mutable state is shared among many components of the software. This concept is well understood, and is the root of common admonishments against the use of global variables. If a function changes the state of a program outside of its scope, and apart from its return value, it is said to have *side-effects*. If a function's return value depends not only on the arguments but also on some external piece of state, the function is said to be *impure* or *referentially opaque*.

Side-effects and referential opacity are mechanisms for obfuscating software. They expand the multitude of possible states in which a program might exist, and worse, they obscure that fact from the programmer who uses these abstractions in an honest attempt to *clarify* their reasoning of the code.

A failure to reason about program state is the root of most problems in both sequential and concurrent programs. A novice might simply suggest we eliminate mutable state from our programs, and would immediately face ill-concealed laughter from many experts. Yet, the idea is one taken very seriously in computer science.

The concept of *funtional programming* is a collection of ideals, among which are immutable state, function purity and referential transparency. This idea is shocking to many. Take a moment to think about programming without using assignment. (That is, no '=' operator except for initialisation.) Without side effects, input and output, as most people know it, is impossible. It is often joked that the only effect of a purely functional program is to warm up the room.

Suffice to say that the functional programming paradigm represents a very different way of thinking about software. It leads to programming patterns which can be used to solve very difficult real-world programming problems. Programs and data structures designed in a functional style are susceptible of mathematically analytical forms of reasoning that are simply impossible with more conventional paradigms. Additionally, programs have natural modular separations because of the lack of coupling. Most modules of a functional program are truly generic and reusable.

The basis for many patterns in functional programming come out of a tight relationship between functional programming and pure mathematics. Many of the patterns are named after their counterparts in *category theory.* For examples, the *monoid* pattern, the *monad* pattern, the *functor* pattern and so on.

Functional programming is the basis of Functional Reactive Programming (FRP). The FRP framework is a set of design patterns and abstractions for modeling hybrid systems. In an FRP-program, events and signals drive the execution of the program, and not the flow control structures. The functional patterns, such as *monads*, *monoids* and *functors* are used to build a set of mutually compatible abstractions around time and event oriented data structures.

The functional programming and FRP abstractions transcend the language of implementation. However, the native language features will determine the fidelity with which the abstractions may be implemented. Some of the critical language features for a faithful realisation of functional programming include anonymous functions (lambda-functions), the ability to pass functions as parameters, the ability of functions to return functions and tail call recursion optimisation.

The major language of implementation in the control systems industry is C or C++. They are standardised, portable, widely known and are unrivalled in the potential to produce efficient machine code. Compilers are available for nearly any target platform from the embedded Atmel AVR to AMD64. They are designed as systems programming languages, meaning they integrate well with hardware and emphasise efficiency and flexibility. The C++ language is, for most intents and purposes, a superset of the C language, additionally offering zero-overhead or low-overhead abstractions. Historically, many of those abstractions have been geared toward Object-Oriented (OO) programming, but C++ is considered a multi-paradigm language.

Until recently, C++ lacked many features that are common in most modern languages. In the 2011 iteration of the C++ standard, ISO/IEC 14882:2011 (C++11) [13], the language underwent major revamping that continues to gain momentum. The steering committee for the International Organisation *for* Standardisation (ISO) C++ standard is gradually bringing the language to feature parity with other modern languages.

The proposed research is opportune, as C++ enters feature freeze for its 2017 standardisation as ISO/IEC 14882:2017 (C++17) [14]. The new version of the standard introduces language features that allow functional programming patterns to be safely, portably and naturally expressed. This is important because such techniques open code up for mathematical analysis using type algebra and the λ-calculus [15]. This introduction of features into C++ is a good reason to reevaluate how we implement control systems software.

1.2 A Brief History & Summary of Prior Progress

In early 2013, I began developing control software for an omnidirectional 3-wheeled mobile robot called the virtualME® by CrossWing Inc. Based on the NMPC algorithm, I first prototyped the software in GNU Octave/Matlab. However, the performance of the software was unacceptable for the online calculation required to control virtualME, with its modest on-board computing resources. So, I began writing a full implementation in C compliant with ISO/IEC 9899:1999 (C99) [16] in a sequential, imperative style.

The C99 version of the software worked efficiently. However, the software was ridged, making it difficult to perform research and development. Also, the needs of virtualME demanded a level of runtime flexibility not easily offered by sequential code. Another rewrite was inevitable.

Definition 3. By common convention, software is called ***ridged*** if a small change in one part of the code enjoins on the programmer a cascade of subsequent changes due to a high degree of interdependency. This is not a standardised term.

The C99 code was, however, sufficient for interesting research. In 2012 we published a conference paper in the proceedings of the Canadian Conference on Electrical and Computer Engineering [17] (See Appendix C). The paper attracted an invitation for an extended version in a special edition of the Canadian Journal of Electrical and Computer Engineering [18]. (Also, see Appendix C.)

Around this time, I had also begun to rewrite the software to control and maintain criticality of a Russian Water-Water Energetic Reactor (VVER) nuclear reactor. That work was published in the proceedings of the International Conference on Nuclear Engineering 23, in 2015 [19]. (See Appendix D.)

Also around this time, I noticed the unusual frequency with which I was duplicating my own effort, and quite attracted to the flexibility on offer from an object oriented design which took full advantage of polymorphism. If I could only define a series of interfaces between classes, making changes would be a matter of creating new implementations without making any serious design decisions.

I rewrote the code for virtualME, and while doing so, began to realise that I could generalise the interface so that the business of the NMPC controller need not be directly coupled with the lower level concerns of the robot. This separation of concerns meant that I could remove and reuse much of the controller code. This lead to software design that I presented in the 2016 Canadian Conference on Electrical and Computer Engineering [20] (Appendix E). I used this design to re-implement the controller for virtualME and presented the research to the executives at CrossWing Inc.

In the months after that, I began exploring theory of software design and programming paradigms in different languages. I wanted to engineer a framework that was capable of supporting my future research efforts, without knowing exactly the direction it would take. Meanwhile, I was searching the control systems literature for new and interesting directions. I became fascinated with the idea of *intelligent control* and started thinking about the fusion of those novel concepts with NMPC. The hope would be that such a fusion would improve on the versatility and resiliency of the NMPC control, while preserving the applicability of the rich theory and analytical techniques that come from optimal control.

As I studied and planned and tested ideas, I began to see how modern software development techniques and programming paradigms could ease the development process of these novel control techniques. I needed, once again, to redesign and rewrite my framework, with deep consideration of best practices.

In realising the monumental scale of the design and development task ahead—the computer science, software engineering, control systems engineering, numerical modeling and simulation, the distributed real-time computing—I discovered my PhD contribution.

1.3 Overview

SECTION 2—*Background* aims to provide a foundation in the theoretical aspects and locates the proposed work within the context of literature and history. The chapter opens with a review of literature in software architecture, modern techniques with the C++ language, optimal control systems and NMPC. The literature review establishes a foundation for the proposed work, while the *prior art* section describes work which has some overlap, or resemblance, to the proposed work. Gaps in the prior art are identified, which localises the proposed work in the state of the art. The chapter concludes with a review of the theory of optimal control and NMPC.[2]

[2] If you, the reader is made uncomfortable by the notation in this section, I *highly* recommend you browse Appendix A, which establishes and justifies the chosen conventions for notation in mathematics.

Section 3—*The Proposal* repeats the contributions (C1), (C2) and (C3), and describes them. The sections to follow describe objectives for each of those contributions. Because the proposal is quite preliminary, in lieu of a time-line I describe a *sequence of research milestones* that I expect to cross, and the *approach and methodology* by which to progress.

2 Background

In this section, the aim is to provide a sufficient set of descriptions to form a reasonable mental picture of the embodiment of the proposed research. This section opens with a literature review encompassing methods for software description of architecture and pattern based design which are necessary background for (C1). The contribution (C2) is motivated partially by the recent history of the C++ language, which is reviewed next, followed by functional programming and FRP. The remainder of the literature review covers various aspects of optimal control, which underlies all of the contributions, but is particularly expressed in (C3).

The next section, §2.2, describes work that potentially overlaps with the proposed contributions. The next section, §2.3, I specifically outline the gaps in literature and prior-art toward which I direct my efforts—localising my research goals and contributions within the field of engineering.

In the remaining sections, I provide a serviceable theoretical background in optimal control and NMPC which supports a detailed discussion of the proposal for (C3).

2.1 Literature Review

Software Architectural Description

The software design (as opposed to the code) will be documented in the form of

1. patterns,
2. contracts, and
3. idioms.

This will include, but are not limited to descriptions compliant with the IEEE standard for software design descriptions (IEEE 1016:2009) [21] and the ISO/IEC/IEEE standard for software architecture description (ISO/IEC/IEEE 420101:2011) [22].

Contracts are a communication tool. They are a precise and complete specification of behaviour visible to the user. They include a description of preconditions, postconditions and invariants. It is a specification of the behaviour of a routine without defining implementations.

Definition 4. A ***precondition*** is a condition on the parameters of a method, or data within the scope of the method that is expected to be true for the method to behave properly. They are obligations of the user of the method. ***Postconditions*** are a similar sort of expectation that must be true *after* the method runs. ***Invariants*** are conditions on parameters and data that are true before *and* after the method runs.

When software code is constrained by contracts, bugs are easier to define since a bug will be a departure from contractually defined behaviour. Well defined contracts make unit testing more straightforward (since it makes clear what to unit test). Clearly defined contractual behaviour combined with the lawful constraints of functional programming precipitates the opportunity for mathematical analysis and provable correctness.

When you were learning to read, you started by learning the letters of the language and the sounds they encode. Next, you begin sounding out words through inspection of the letters. When you became good at reading, you no longer had to concentrate on individual letters. Instead, you grouped them visually into whole words. Finally, in high-school you were taught the anatomy of various forms of literature to tell stories.

Software patterns are to software what archetypes are to literature. The programming language provides the dictionary of words, from which your code forms the sentences. But, the story is made of archetypes: the evil super-villain, the hero's small home town, the lone cowboy. The author is not constrained by archetypes, and is free to adapt and reinterpret them. But archetypes provide useful abstractions so that an expert reader is implicitly granted insight. Software design patterns similarly facilitate unwritten communication between program authors and the reader, and that is what design patterns are about.

Patterns are small design units, of the sort that could be represented with Unified Modeling Language (UML) [23] diagrams. They define a solution to common design problems. So, if you see one used, you also know something about the problem the programmer was trying to solve. The intent is communicated through the application of the pattern. For example, if I use the *abstract factory* pattern, (one of the original Gang *of* Four (GoF) patterns [24]), and I write a class called `WidgetFactory`, the nomenclature makes clear to readers of my code what is and how they should use the `WidgetFactory`. It is self-documentation for appropriately literate readers.

Because software patterns are a communication tool based on common solutions to common problems, they are useless as a personal fabrication. They must be identified in the structure of production software. A significant phase of research will require me to survey existing code, looking for these types of patterns. I will document, and make rigorous definitions for the patterns and their use in control systems programming.

Functional Programming

Probably the most common and venerable introduction to functional programming is John Hughes classic paper "Why Functional Programming Matters" [25]. A bit of history is to be found in David Turner's "Some History of Functional Programming Languages" [26].

Systematic approaches to explaining functional programming often begin the chronology in the 1930s with Alonzo Church and the λ-calculus [15], [27]. The λ-calculus is a mathematical theory to generalise untyped functions and make rigorous the rules for composition and substitution. This naturally requires a way of describing functions anonymously, giving rise to the *lambda* binding notation. For example, $\lambda x.x^2$, binds free occurrences of x to x^2. This is the origin of the convention to name anonymous functions *lambdas* in programming.

Church attempted to demonstrate that functions could be used to encode numerals, and could therefore be a formal basis for all mathematics.

LISP and Algol 60 are often cited as the first languages to implement concepts from the λ-calculus. But in 1966, Peter Landin published a landmark paper called "The Next 700 Programming Languages" [28] in which he gives semantics for ISWIM, a language based directly on Church's λ-calculus.

In 1977, John Backus's in his Turing award speech [29] describes an algebra of programs and is often credited with popularising the notion of functional programming. His talk directed attention to the work of Church, Landin and others, inspiring the invention and academic development of a series of functional languages. In late 1980s a committee was convened to integrate decades of work in functional programming into a single language: *Haskell*.

A thorough review of these seminal works will give the reader little idea of what functional programming looks like today in practice. Sources for this are numerous, and language dependant, but in general I have found most useful sources to be instructional material for the Haskell language. Haskell is the primary language for academic research in the field. A great deal of modern work involves the use of category theory to develop principled and lawful abstractions and patterns that improve efficiency, push application boundaries and ensure safe parallelism and distributed computation.

A book I am fond of is Allen and Moronuki's *Haskell Programming from first principles* [30], but there are numerous others which I have yet to consume. I started with it, because it appears to be one of the more recent introductory volumes. I have found a useful introduction to category theory in Spivak's *Category theory for the sciences* [31].

A great deal of work in functional programming and FRP is indexed in the publications listing of the Yale Haskell group at

`http://haskell.cs.yale.edu/publications/`

The group was founded by Paul Hudak (1952–2015) who was a co-founder of FRP. The publications are from only one group, but they were very productive, and cite a lot of relevant literature.

Because I wish to target control applications which often use embedded hardware that is potentially modest, my work will be implemented in C++. There is very little formal work on functional programming in C++. A book by Ivan Čukić is in development for Manning Publications [32] and its expected to be released later this year. Based on several conference talks from professionals at companies like Netflix, Facebook, Twitter and Google, there is functional C++ code being written and used.

Functional Reactive Programming

In 1997 Conal Elliott and Paul Hudak published what was to become a popular paper entitled "Functional reactive animation" [33]. In it, they describe *Fran* (Functional Reactive Animation), a domain specific language embedded in Haskell for abstracting the composition of computerised graphical animations. This was the birth of the patterns and paradigm that would become known simply as Functional Reactive Programming (FRP) [34], [35]. Since then, FRP has been ported to create other domain specific languages. For examples, *FVision* for visual tracking [36], *Frob* for Robotics [37], [38] and at least 6 other domain specific implementations of the paradigm.

The FRP paradigm is a set of abstractions for making time varying signals and events first class objects. These abstractions are based on a rich substrate of mathematics and is shown to model hybrid systems exactly in the limit of small time-sampling. Functional abstractions based on category theoretical concepts facilitate automatic conversion of non-FRP functions to work on behaviours and events. This allows the user to think at a high level of abstraction, near the mathematical description of elements in the problem and solution domains.

My focus will be on hybrid systems modeling in distributed environments. In particular, the research Hudak, Peterson, Nilsson and others. Key works include [39]–[41]. John Peterson specifically addresses issues of parallelism in [42]. Performance issues that affect real-time applications are addressed in [39] and references therein.

The richest source of literature I have found on the implementation of FRP in C++ is Louai Al-Khanji's Master's thesis, "Expressing Functional Reactive Programming in C++" [43]. In the thesis he describes a C++ implementation of the paradigm which self-admittedly is lacking in refinement.

There are now several publicly available modern C++ libraries for implementing FRP, such as Sodium[1], C++React[2], and David Sankel's sfrp[3]. David Sankel's FRP implementation has an interesting history. He designed the program semantics for controlling a robot arm *before* realising that he had created a functional reactive design. He tells that story in one of his popular talks from BoostCon 2014[4]. His entire personal library, including C++ headers for functional programming and FRP are available on his bitbucket repository [44].

[1] `github.com/SodiumFRP/sodium`

[2] `github.com/schlangster/cpp.react`

[3] `https://goo.gl/qoqlQP`

[4] `youtu.be/tyaYLGQSr4g?t=38m32s`

ISO/IEC Standard C++

The history of the C++ language began with its original creator Bjarne Stroustrup in 1979 as "C with classes". The language was not compiled, but transpiled into C and then compiled using which ever C compiler was available to the user. It was not until 1998 that the language was standardised by committee under the ISO and International Electrotechnical Commission (IEC) and published as ISO/IEC 14882:1998. Between 1998 and 2011, the C++ language was largely static, with only a minor update to the standard (with no new features) in 2003. A major update in 2011 brought C++ into the modern era, facilitating styles of programming pioneered by languages such as Python and Haskell. At the time of this writing, the current standard is ISO/IEC 14882:2014 (C++14) [45]; and C++17 is feature frozen and en route to publication in 2017. The standards committee currently plans new releases every 3-years, with every other release considered *major*. C++11 was a major release, and C++14 as comparatively minor, but still important. C++17 is expected to be the most drastic departure from *class* C++ (that is, C++98) yet.

Bjarne Stroustrup's compendium, *The C++ Programming Language* [46], is a very accessible plain language discussion of the key parts of the C++11 standard, with an introduction to the language in the first chapter. The time constrained reader wanting a primer on modern C++ in smaller volume will appreciate Stroustrup's *A Tour of C++* [47]. That book is basically the first chapter of [46], augmented so as to be self contained.

Because a core value of the C++ committee is backward compatibility with previous language versions, the current emphasis of the committee is on making the language simpler by adding the right features. As early as 1994 (and probably earlier), Bjarne Stroustrup wrote

> Within C++, there is a much smaller and cleaner language struggling to get out.
>
> *The Design and Evolution of C++. p.207*

and later added,

> And no, that smaller and cleaner language is not Java or C#
>
> *FAQ item "Did you really say that?" on Dr. Stroustrup's website*

In order to evolve the language by adding features without removing any, the new features must be strategically chosen to bring about shifts in programming style. New features, used together idiomatically, form replacements for older techniques, without creating direct redundancies. Code compliant with C++98 will compile on a C++14 compiler, but an informed programmer could distinguish *Modern C++* by inspection, despite the backward compatibility.

Modern code will not only make use of new the new language features, but will portray a style that has evolved over decades of C++ use by millions of programmers, computer scientists and software engineers. Such practices are often learned *on the job*, but are also promulgated in prominent texts, such as Scott Meyers *Effective Modern C++* [48]. However, the most monolithic effort to document and standardise best practices is the *C++ Core Guidelines* project [49]. Those guidelines are supported by the C++ Guideline Support Library [50].

Aside from good general programming practices, lightweight abstractions and intuitive Application Programming Interfaces (APIs), the proposed work will focus heavily on developing thread-safe and parallel algorithms and patterns. The C++11 standard was the first to openly support threading in the language (via the memory model) and in the C++ standard library. The popular text on that subject is Anthony Williams' book *C++ Concurrency In Action* [51]. The C++14 standard made only small additions to concurrency[5]. A common criticism of the C++ standard for concurrency is that the interface for `std::promise` does not offer composability. This means that they can be processed in order of completion. Asynchronous event handling mechanisms in C# and other languages are composable. The need for composability of synchronisation events has been widely recognised. The Microsoft Windows API has `WaitForMultipleObjects` and Unix has the venerable `select` call.

[5] Such as `shared_timed_mutex` and `shared_lock`.

Optimal Control and Nonlinear Systems

There are several well established and classic texts in optimal control theory. I began with an older volume by Kirk simply titled *Optimal Control Theory* and subtitled *an introduction* [52]. There is no emphasis on programming code, but algorithms for solving optimal control problems are detailed. The book is very practical and calculation driven. It contains a sturdy introduction to the calculus of variations. The text focuses on indirect methods using the formalism of Pontryagin.

Though the calculus of variations is not a necessary for solving optimal control problems, it is an inseparable part of the field and its literature. The reason that, historically, so much effort has been put into developing techniques with this calculus is that it provides extremely useful intellectual abstractions that aid in analysis and understanding.

A valuable text focusing only on the *Calculus of Variations* is from Gelfand and Fomin. The original was written in Russian and used as a course textbook, however the book was translated to English and revised several years later by Richard A. Silverman [53].

Another important treatise in optimal control is Stengel's *Optimal Control and Estimation* [54]. This book, an aged but highly regarded graduate level text, may easily be the preferred introduction for anyone with a classical education in (linear) control systems theory. Starting from very naïve principles, Stengel builds a theory of optimal control. Developing from that, a theory of estimation which he combines into two chapters on *stochastic optimal control.*

Despite the fact that this section is dedicated to optimal control, Vidyasagar's *Nonlinear Systems Analysis* [55] requires special mention. The analytical techniques it describes are entirely relevant to optimal control of nonlinear systems. The reader benefits greatly from the intuitive grasp the author imparts in every page. I fully expect that a good deal of background in analysis of nonlinear systems in my final thesis will be inspired heavily from Vidyasagar's writing.

An excellent overview of numerical methods for optimal control is found in a survey paper by Rao [56]. Rao's survey lacks depth (for practical reasons), but is a valuable index for the original literature expounding the various techniques. This paper is well paired on a reading list with Cannon's review of efficient algorithms for NMPC [57]. Though Cannon's paper is centred on NMPC, and not more broadly on optimal control, his key focus is on numerical techniques for the minimisation problem which is core to both.

NMPC

As an introduction to the recent state of research, I know of no better source than David Mayne's recent survey [58]. It is remarkably comprehensive and contains 170 references to books, survey works and seminal papers in all major sub-fields of model predictive control research. The paper briefly introduces the core N/MPC problem, but is far from an introduction to the topic. For a more introductory review of NMPC, I direct the reader to the introduction written by Findeisen and Allgöwer [59], or James B. Rawlings' tutorial overview in the June 2000 issue of IEEE Control Systems Magazine [60]. A more comprehensive introduction is provided in some of the textbooks in the field. For example, Grüne and Pannek in their standard monograph [61].

The Model Predictive Control (MPC) technique was first conceptualised several times independently between the 1960s and 1980s [61], [62], but was not able to find widespread use due to the modest computational technology of those decades—even with completely linear(ised) problems. In the 1980s, the process control industry, with its characteristically gradual plant dynamics, took hold of the method following the seminal paper by Richalet at others [63]. Three decades later, even consumer grade computing hardware can implement nonlinear-MPC for things like unmanned aircraft [64], mobile robots [18], automotive vehicles [65] and other systems which rapidly traverse their state spaces. A brief but compelling history of the early development of MPC is given by Camacho and Bordons in [62].

When the process industry embraced MPC, the finite horizon meant that no guarantees of stability existed. It was simply an act of empirically driven-design that horizon sizes were made large enough to give confidence in stability. Around 2000, Mayne, Rawlings, Rao, *et al.* developed the analysis (using Lyapunov theory) to achieve nominal stability under appropriate constraints [66].

Stability in NMPC is a difficult problem and an ongoing area of research. The stability problem stems from the fact that a finite sized preview horizon cannot provide foresight to navigate around constraint regions, leading to cyclical integral curves in the vector field (that is, f_c or f_+). If the characteristic size of the constrained regions are small compared to the horizon size, they do not pose a practical problem for stability. But analytical tools that can inform our concerns are difficult to wield for non-experts, and those tools are limited, and often inject undesirable constraints into the problem.

A solution is to employ navigational path planning techniques to plot a desired route through state-space, and then use the NMPC algorithm to track the desired path. This shifts the responsibility of cycle avoidance away from the controller and into a higher level planner which can supervise the progress of the system.

Numerical Methods for NMPC

Conventional approaches to solving NMPC Optimal Control Problem (OCP_N) can be divided into the categories of *direct* or *indirect methods.* Indirect methods use the calculus of variations to determine first-order optimality conditions of the original control problem, leading to a two-point boundary value problem. Direct methods call for a discretisation of the problem. Once discretised, it can be transcribed in a straightforward manner to a NonLinear Program[ming] (NLP) problem. Many mature, efficient and well known strategies exist for solving NLPs.

Findeisen and Allgöwer outline various NMPC techniques in [59], as well as Cannon in [57]. More detailed, but still brisk is Rao [56].

Both direct and indirect methods will require numerical differential equation solvers and numerical integrators that can be found in any text on numerical-methods. However, the approach to finding the solution to the optimisation problems differ vastly in approach.

2.2 Prior Art

THERE IS NO ABUNDANCE OF WORK with a focus on software design for model predictive control systems. As far as I have been able to find, there are no textbooks, or recent papers.

In 1994, Jobling, Grant, Barker, *et al.* surveyed the impact of oo-programming in control system design [67]. A lot has changed about the styles and attitudes of software developers since the mid-1990s, but those efforts are very much in the spirit of my proposed work.

In 1993, Boasson wrote an historical perspective on the evolution of control systems software architecture [68]. The work is obviously dated, but again shares the spirit of the proposed work, and is sure to lend insight. That date places the work before the rise of enterprise-oo, which is often considered to be a distortion of classical principled oo design.

Despite the lack of communication about design principles in the control application, there have indeed been endeavours to make generic software frameworks for control, such as the Defence Advanced Research Projects Agency (DARPA) Software Enabled Control project, or the National Institute of Standards and Technology (NIST) The Real-time Control Systems Architecture.

The DARPA Software-Enabled Control project [1], [69] probably represents the largest independent overlap with my proposed work. However, some key components of the software are proprietary, and not publicly available. It was part of a military project, which is made obvious by the many papers with application to military ground vehicles and aircraft.

A similar project to produce a general framework for control came from the Intelligent Systems Division of the NIST[6]. The project, entitled the *The Real-time Control Systems Architecture* has software support called the Real-Time Control Systems Library. There is a deep hierarchy of supportive technologies, such as the NIST Reference Model Architecture for Intelligent Systems Design, and the NIST Neutral Messaging Language, all of which couple with the library.

[6] Details at https://www.nist.gov/intelligent-systems-division

Those technologies offered by NIST are no longer developed or maintained, and are considered *legacy* projects. They are also remarkably complex which presents a significant barrier to entry. I am proposing something with a sufficiently small cognitive footprint that an enthusiastic graduate student could have a small project running in a week or so.

The software in the components of the NIST architecture and the publicly available components of the DARPA project will be the subjects of my search for design patterns. While they may be useful as inspiration, I must emphasise again that I have no intention of developing design tools for control systems. Rather, I want to develop frameworks to support research and development of new control techniques based on predictive models.

In 2001, Peterson, Hager, and Serjentov produced a case study using FRP to control their robot in the 2000 Robocup competition [70]. The produced a subset of FRP using C++ template metaprogramming techniques to create what they describe as a “powerful, extensible programming environment for robotics”. Some similar research was produced in 2002 by Dai, Hager, and Peterson [71] where they explicitly describe their work as a *domain specific embedded language* within C++.

With regard to (C3), there is some work done with NMPC controllers which are able to dynamically adjust or swap predictive models during operation. For example, in 2016 Zhang, Sprinkle, and Sanfelice, and related work developed a computationally aware controller that could vary the fidelity of the model to reduce computational burden [72, and related work]. Also in 2016, Ostafew, Schoellig, and Barfoot produced a NMPC controller which is able to learn more accurate models [5]. The concept of switching components, such as the predictive model, the minimisation technique and even the constraint techniques is something I wish to explore, and so there is overlap with these papers.

2.3 Literature Gaps

With regard to (C1) and (C2), there is very little literature which directly addresses the problem of engineering the predictive control systems software. There is a small variety of software available for designing and implementing fairly straightforward NMPC controllers, but those tools are designed to hide implementation details and minimise implementation choices, and I am proposing a research framework which necessarily exposes those decisions.

In the field of software engineering, interest has been building for functional and functional reactive designs. The limitations of the C++ language prior to 2011 made it awkward to produce truly functional architecture, so progress was limited in many fields. The benefits of these paradigms were clearly seen in more dynamic fields such as web development, cloud computing and in asynchronous support libraries for desktop software (such as those found in Microsoft Windows).

With regard to (C3), there is a lack of good literature regarding choices in implementation details of NMPC controllers. I wish to study the impact of some of those choices.

Enabled by the highly dynamical nature of the proposed framework, there is a lot that can be be understood about re-configuring NMPC controllers during operation. This is a fresh and open area of research.

In §2.4, I will describe techniques for discouraging the controller from certain behaviours using potential fields in the cost function (see Eq. (2.20)). The more direct way to do this is to proscribe undesired states using constraint techniques in the mathematical optimisation[7]. Techniques which penalise behaviour using the cost function, are called *soft constraints*, while directly proscribed behaviours are called *hard constraints*.

[7] For example, using Lagrange multipliers, or NLP

In textbooks and elementary literature, you will find discussions comparing hard and soft constraints at a naïve level. There may be discussion of the fact that hard constraints are computationally more expensive, or that soft constraints do not provide guarantees against the undesired behaviour. There will be little or no discussion of techniques for constructing potential fields.

2.4 Optimal Control & NMPC

Nonlinear MPC has, at its core, a numerical optimal control problem. In this section I shall focus on theory and methods for solving optimal control problems as they arise in the context of online NMPC.

We will move swiftly into the discrete flavour of optimal control, but the origins of the problem are rooted in classic and continuous formulations. There are also several techniques for discretising the problem, so it will be useful to have a fully continuous case for reference.

Optimal control is quite different from the classical control techniques which use model information only at design time to choose a rule which maps each state to a control value to obtain desired behaviour. Optimal control formulates the control problem in terms of a cost (or reward) functional that maps a state space trajectory and corresponding control space trajectory to a real number which quantitatively appraises the paths in terms of desirable behaviour. A mathematical model of the system is used to estimate a state-space trajectory that will result from a given control plan. We may then seek the state-space/control-space trajectory pair which minimise the cost.

A common model for a nonlinear system is

$$\dot{\boldsymbol{x}} = \boldsymbol{f}_{\mathrm{c}}(t,\ \boldsymbol{x}(t)), \quad \boldsymbol{x}(0) = \boldsymbol{x}_0, \quad t \geq 0, \tag{2.1}$$

where $\boldsymbol{f}_{\mathrm{c}} : \mathbf{R} \times \boldsymbol{X} \longrightarrow \boldsymbol{X}$ is a time varying nonlinear map from a vector of state at time t, $\boldsymbol{x}(t) \in \boldsymbol{X}$, to the state velocity $\dot{\boldsymbol{x}}$, forming an initial value problem. When $\boldsymbol{f}_{\mathrm{c}}$ is specified, time-integration will yield an integral curve leading from the initial state. The problem is often visualised with $\boldsymbol{f}_{\mathrm{c}}$ representing a vector field (that is, a velocity field) and the integral curves are the trajectories to which the vectors are locally tangent at every point.

A controlled system model will extend (2.1) with a control input, $\boldsymbol{u}$:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}_{\mathrm{c}}(t,\ \boldsymbol{x}(t),\ \boldsymbol{u}(t)), \quad \boldsymbol{x}(0) = \boldsymbol{x}_0, \quad t \geq 0. \tag{2.2}$$

Depending on the specific dependency of $\boldsymbol{f}_{\mathrm{c}}$ on $\boldsymbol{u}(t) \in \boldsymbol{U}$, this introduction allows us to control the shape of the integral curves or, more to the point, the shape of the underlying vector field if $\boldsymbol{u}$ can be specified as a function of $\boldsymbol{x}$: $\boldsymbol{u}(\boldsymbol{x}(t))$. The foundational problem of control systems in this context is to find a law giving the values for the elements of $\boldsymbol{u}$ to ensure that all possible integral curves represent desirable behaviour.

The desirability of a particular solution must be quantifiable as a *cost function(al),* based on the trajectories through state and control space the solution represents. For example, if I am trying to find a minimum-time path, a cost functional could simply be $\int_{t_0}^{t_{\mathrm{f}}} \mathrm{d}t = t_{\mathrm{f}} - t_0$. But that could (or will) lead to solutions which ride the limits of the control actuators. Perhaps what you really want is a *minimum-effort* solution: $\int_{t_0}^{t_{\mathrm{f}}} \big(\boldsymbol{u}(t)\big)^{\mathsf{T}} \boldsymbol{u}(t)\ \mathrm{d}t$. Anything with a direct relationship to the choices of state and control may be penalised in this way. The term in the integrand responsible for accumulating cost over the trajectory is called the *running cost*, and is usually denoted with a capital (and sometimes calligraphic) L.[8] In many formulations, it is desirable (for reasons to be discussed) to place an additional penalty on the location of the terminus of the state-space trajectory, denoted $e(\boldsymbol{x}(t_{\mathrm{f}}))$.

[8] The custom of using L for the running cost commemorates the Italian-French mathematician and astronomer Joseph-Louis Lagrange, who revolutionised physics by reformulating (Newtonian) mechanics in terms of this sort of functional minimisation.

Definition 5. *The **Optimal Control Problem (OCP)** is to solve for the free variables $\boldsymbol{x}(t)$ and $\boldsymbol{u}(t)$ that satisfy the minimisation,*

$$\min_{\boldsymbol{x}(t),\ \boldsymbol{u}(t)} \left[\int_{t_0}^{t_{\mathrm{f}}} L(t,\ \boldsymbol{x}(t),\ \boldsymbol{u}(t))\,\mathrm{d}t + e(\boldsymbol{x}(t_{\mathrm{f}})) \right], \tag{OCP}$$

subject to

$$\begin{aligned} \boldsymbol{x}(t=0) - \boldsymbol{x}_0 &= 0 && && (2.3)\\ \dot{\boldsymbol{x}}(t) - \boldsymbol{f}_{\mathrm{c}}(\boldsymbol{x}(t),\, \boldsymbol{u}(t)) &= 0 && t_0 \le t \le t_{\mathrm{f}} && (2.4)\\ g_k(\boldsymbol{x}(t),\, \boldsymbol{u}(t)) &= 0 && t_0 \le t \le t_{\mathrm{f}} && (2.5)\\ h_k(\boldsymbol{x}(t),\, \boldsymbol{u}(t)) &\le 0 && t_0 \le t \le t_{\mathrm{f}} && (2.6)\\ r(\boldsymbol{x}(t_{\mathrm{f}})) &\le 0. && && (2.7) \end{aligned}$$

The constraint (2.3) merely ensures that the solution trajectory starts at the systems known present state. Equation (2.4) ensures that the solution is consonant with the dynamical description in our model. The constraints (2.5) and (2.6) are algebraic expressions constraining the solution space. For example, if the control input is limited to a ball in the control space, it may be expressed here. The final constraint, (2.7), ensures that the terminus of the solution trajectory will lie in a subset of the state-space. This is key to some techniques to guarantee stability in NMPC. At minimum, (2.3) and (2.4) are part of every well posed OCP, while the others are at the discretion of the designer. Optimal control techniques offer and interesting choice to control designers, in that constraints may be express in *hard* or *soft* varieties. If the constraint is expressed through active enforcement of (2.5)–(2.7), then it is a *hard* constraint. However, the cost function can be chosen so as to anathematise state-control configurations without the computational overhead of hard constraints. This comes at the cost of certainty in the proscription of such regions, and so these constraints are called *soft.*

Nonlinear Model Predictive Control

Since our goal is a computer algorithm, we must sample the state and control trajectories on discrete intervals. This is true regardless of how we approach the problem, since it is a limitation imposed by digital hardware. Choosing a constant sampling interval t_Δ, the system state at a time $t_k = kt_\Delta$ is notated with the short hand $\boldsymbol{x}(k\, t_\Delta) = \boldsymbol{x}^k$. The model from (2.2) takes the form of a recurrence relation

$$\boldsymbol{x}^{k+1} = \boldsymbol{f}_{+}(\boldsymbol{x}^k,\, \boldsymbol{u}^k), \quad \boldsymbol{x}^0 = \boldsymbol{x}_0, \quad k = 0, 1, 2, \dots \tag{2.8}$$

where the *state transition map*, $\boldsymbol{f}_{+} : X \times U \longrightarrow X$, assigns a successor state $\boldsymbol{x}^{k+1}$ from a given state vector $\boldsymbol{x} \in X$ and control vector $\boldsymbol{u} \in U$. The domains X and U may be arbitrary metric spaces, with metrics $d_X(\boldsymbol{x}_1, \boldsymbol{x}_2)$ and $d_U(\boldsymbol{u}_1, \boldsymbol{u}_2)$. For the time being, it is harmless to imagine that $X = \mathbf{R}^n$ and $U = \mathbf{R}^m$ for $n, m \in \mathbf{Z}_+$, with the standard Euclidean metric $d_X(\boldsymbol{x}_1, \boldsymbol{x}_2) = \|\boldsymbol{x}_1 - \boldsymbol{x}_2\|$, (and alike for d_U), but this need not be the case.

Given a *control sequence* $\{\boldsymbol{u}^0, \boldsymbol{u}^1, \dots, \boldsymbol{u}^{N-1}\} \in U^N$ for $N \in \mathbf{Z}_+$, we can forecast a corresponding (discrete) state-space trajectory $\boldsymbol{x}_{\boldsymbol{u}}^k$ based on the model. Starting from an initial position $\boldsymbol{x}^0$, simply iterate (2.8) recursively, using a previously obtained value. That is, starting from $\boldsymbol{x}^0$ we obtain $\boldsymbol{x}^1 = \boldsymbol{f}_{+}(\boldsymbol{x}^0, \boldsymbol{u}^0)$, and then $\boldsymbol{x}^2 = \boldsymbol{f}_{+}(\boldsymbol{x}^1, \boldsymbol{u}^1)$, and so on until $\boldsymbol{x}^N = \boldsymbol{f}_{+}(\boldsymbol{x}^{N-1}, \boldsymbol{u}^{N-1})$. In this context, the integer N is referred to as the *prediction horizon*, or simply as *the horizon.*

Definition 6. *The* ***NMPC Optimal Control Problem (OCP$_N$)*** *is to solve for the free variables in the sequences* $\boldsymbol{x} = \{\boldsymbol{x}^0, \boldsymbol{x}^1, \dots, \boldsymbol{x}^{N-1}\}$ *and* $\boldsymbol{u} = \{\boldsymbol{u}^0, \boldsymbol{u}^1, \dots, \boldsymbol{u}^{N-1}\}$

that satisfy the minimisation,

$$\min_{\substack{\boldsymbol{x} \in X^{N+1} \\ \boldsymbol{u} \in U^{N+1}}} \left[\sum_{k=0}^{N-1} L(\boldsymbol{x}^k,\ \boldsymbol{u}^k) + e(\boldsymbol{x}^N) \right] \tag{OCP$_N$}$$

subject to

$$\boldsymbol{x}^0 - \boldsymbol{x}_0 = 0 \tag{2.9}$$

$$\boldsymbol{x}^{k+1} - \boldsymbol{f}_+(\boldsymbol{x}^k, \boldsymbol{u}^k) = 0 \quad k = 0, \dots, N-1 \tag{2.10}$$

$$g_k(\boldsymbol{x}^k, \boldsymbol{u}^k) = 0 \quad k = 0, \dots, N-1 \tag{2.11}$$

$$h_k(\boldsymbol{x}^k, \boldsymbol{u}^k) \leq 0 \quad k = 0, \dots, N-1 \tag{2.12}$$

$$r(\boldsymbol{x}^N) \leq 0 \tag{2.13}$$

Here, the constraints play the same roles (now in discrete form) as they did for Definition 5, the definition of the OCP.

It is not uncommon for Definitions 5 and 6 to be augmented with an algebraic problem to be solved simultaneously with the differential problem. Such problems have an additional set of free variables for the algebraic problem. It adds complexity without insight, and is not pertinent to the chosen examples, it is not discussed further.

The objective of the minimisation in OCP$_N$ is our discrete cost function which is commonly notated as $J_N : X^N \times U^N \longrightarrow \mathbf{R}$. Because the model constraint (2.10) unambiguously maps a control sequence to a state-space trajectory, a large class of numerical methods for NMPC eliminate the state-space variables and free variables in the minimisation problem. For those methods, (OCP$_N$) might be rewritten as

$$\boldsymbol{u}_\star = \arg\min_{\boldsymbol{u} \in U^N} J_N(\boldsymbol{x}_{\boldsymbol{u}},\ \boldsymbol{u}). \tag{2.14}$$

The ornamental '$\star$' denotes values relating to the solution of the optimisation problem. So $\boldsymbol{u}_\star$ is the optimised control sequence (that is, the one which minimises the cost function), while $\boldsymbol{x}_\star$ would represent the corresponding trajectory through state-space.

In the NMPC algorithm, $\boldsymbol{u}_\star$ is computed repeatedly and the first element, $\boldsymbol{u}_\star^0$, is executed before solving the problem again with a newly measured $\boldsymbol{x}^0$. A useful notational tool is the feedback law: $\boldsymbol{u}(t) = \boldsymbol{\mu}(\boldsymbol{x}(t))$. In this context, the feedback law $\boldsymbol{\mu}_c : X \longrightarrow U$ maps the current state to the appropriate control[9]. Let us summarise the NMPC algorithm more precisely:

[9] If the feedback law can be expressed in closed form, then simple substitution,

$$\boldsymbol{f}_c(t, \boldsymbol{x}(t), \boldsymbol{\mu}_c(\boldsymbol{x}(t))) = \boldsymbol{f}_c'(t, \boldsymbol{x}(t)),$$

can transform (2.2) back to (2.1), justifying my comment that choosing a control law can be thought of as manipulating the shape of the underlying vector field.

Algorithm 2.1 (The general NMPC algorithm)**.**

```
1: repeat                                                        ▷ NMPC loop
2:     Measure or estimate the state $\boldsymbol{x}^j \in X$
3:     Set $\boldsymbol{x}_0 = \boldsymbol{x}^j$ and solve OCP$_N$ for those initial conditions
4:     Define the feedback control value $\boldsymbol{\mu}(\boldsymbol{x}^j) \coloneqq \boldsymbol{u}_\star^0 \in U$ and execute
5: until control process is inactivated
```

The Cost Function

The choice of the cost function impacts the behaviour and performance of the controller in the most thorough sense. The cost function is how the designer expresses the notions of good and bad behaviour. But since this function is core the optimisation problem—the most computationally intensive portion of the controller—a balance is to be struck between a thorough quantification of behavioural characteristics and the practical performance of the controller.

The cost function is commonly expressed as a sum of *running cost terms* over the horizon, with an additional term depending only on the terminal state, called the *terminal cost*:

$$J_N(\boldsymbol{x},\ \boldsymbol{u}) \coloneqq \sum_{k=0}^{N-1} L(\boldsymbol{x}^k,\ \boldsymbol{u}^k) + e(\boldsymbol{x}^N). \tag{2.15}$$

The Running Cost

The function $L : X \times U \longrightarrow \mathbf{R}$ appearing in (2.15) is the *running cost* and expresses the operational costs of traversing state-space-time.

Without loss of generality, the controllers described here are designed to track a reference path in state space:

$$\boldsymbol{x}_{\text{ref}} = \left\{\boldsymbol{x}_{\text{ref}}^0,\ \boldsymbol{x}_{\text{ref}}^1, \dots, \boldsymbol{x}_{\text{ref}}^{N-1}\right\}.$$

To track the above path is to minimise the error function $\tilde{\boldsymbol{x}} = \boldsymbol{x}_{\text{ref}} - \boldsymbol{x}_{\boldsymbol{u}}$. This desire quantified quadratically in the running cost function as

$$L\left(\boldsymbol{x}_{\boldsymbol{u}}^k, \cdot\right) \coloneqq \left(\tilde{\boldsymbol{x}}^k\right)^{\mathsf{T}} \boldsymbol{Q}\, \tilde{\boldsymbol{x}}^k \tag{2.16}$$

where $\boldsymbol{Q}$ is positive-definite matrices of weighting coefficients. The positive-definiteness of $\boldsymbol{Q}$ ensures that, all other things being equal, $\|\boldsymbol{x}_1\| > \|\boldsymbol{x}_2\| \iff L(\boldsymbol{x}_1, \cdot) > L(\boldsymbol{x}_2, \cdot)$.

If the the elements of the reference sequence are all equal (that is to say the reference is static) then the tracking controller becomes a set-point controller.

Minimising the tracking error in the way described, surprisingly to some, may lead to very undesirable behaviour. Plain error minimisation without further regard will drive actuators to their limits because the *only* desire expressed in formulation is adherence to the reference trajectory. Even if you constrain your optimisation to respect the limits of your actuators, the controller will tend to extremes—casually running at the operational limits you prescribe in the constraints.

Solutions which balance the tracking error against the control effort must be explicitly called for. A quadratic penalty on control effort, in addition to the tracking error terms will balance needless aggression in the controller against tracking accuracy. (Assuming the notion of *needless* aggression means anything in your application.)

An individual element in the control sequence is, in general, a time varying function $\boldsymbol{u}^k(t)$ with a duration equal to t_Δ. Since the cost and cumulative effect of a control is computed by integration, $\boldsymbol{u}^k(t)$ should locally Lebesgue integrable. Let $\mathbf{L}^s_{\infty,\,\text{loc}}\Omega$ denote the set of all locally Lebesgue integrable functions that map Ω to $\mathbf{R}^s$. Then the control space is expressed as $U = \mathbf{L}^m_{\infty,\,\text{loc}}[0,\ t_\Delta)$. In such cases, the control effort penalty would take the form,

$$L(\cdot,\ \boldsymbol{u}^k) \coloneqq \cdots + \frac{1}{t_\Delta}\int_0^{t_\Delta} \left(\boldsymbol{u}^k(t)\right)^{\mathsf{T}} \boldsymbol{R}\, \boldsymbol{u}^k(t)\,\mathrm{d}t. \tag{2.17}$$

where, as it was with $\boldsymbol{Q}$, the matrix of coefficients, $\boldsymbol{R}$, should also be positive definite.

I previously wrote that the reader may harmlessly think of U as simply $\mathbf{R}^m$, and the attentive reader may be wondering about the apparent dissonance I am creating with this discussion of U as a function space. This tension is broken when we implement the so called *zero-order hold* for the control sequences, in which controls are held constant between sampling nodes[10]. The choice eases mathematical rigour and reduces the computational burden of the final algorithm. The set of all constant functions mapping the interval $[0,\, t_\Delta)$ to $\mathbf{R}^m$ are a subset of $\mathbf{L}^m_{\infty,\,\mathrm{loc}}[0,\, t_\Delta)$, so the choice is algebraically sound. Under the zero-order hold condition, the contribution of the control effort to the running cost simplifies from (2.17) to

$$L(\cdot, \boldsymbol{u}^k) \coloneqq \cdots + \left(\boldsymbol{u}^k\right)^{\mathsf{T}} \boldsymbol{R}\, \boldsymbol{u}^k. \tag{2.18}$$

[10] If the sampling interval is sufficiently short, then the zero-order hold is a very practical choice to make. More elaborate choices are indicated when the sampling interval is longer, or the physics of the underlying mechanism do not allow for constancy between sampling nodes.

While the zero-order hold is a very common convention, the explorations of the section on *numerical methods of* NMPC, §5, will describe alternative ways to discretise the problem.

IN MOST applications of NMPC, the running cost consists only of the terms described in Equations (2.16) and (2.18). In summary,

$$L(\boldsymbol{x}^k_{\boldsymbol{u}},\, \boldsymbol{u}^k) \coloneqq \left(\tilde{\boldsymbol{x}}^k\right)^{\mathsf{T}} \boldsymbol{Q}\, \tilde{\boldsymbol{x}}^k + \left(\boldsymbol{u}^k\right)^{\mathsf{T}} \boldsymbol{R}\, \boldsymbol{u}^k. \tag{2.19}$$

In some applications, it is useful to disproportionately discourage paths through a particular portion of state-space. Take, as an example, the case of a self-driving car which may want to move slowly near blind corners (thus discouraging states of high velocity near those areas). These regions of state-space may be cast as inauspicious with a scalar field which is accumulated in the running cost. The field $\Phi : X \longrightarrow \mathbf{R}$ takes on high (positive) values in regions of X which the designer wishes to ward against. If Φ is finite, it does not preclude trajectories from entering those regions with large Φ, but it does discourage such trajectories as solutions of cost minimisation. We therefore call such constraints *soft constraints.* This brings us to our final and most comprehensive formulation of the running cost,

$$L(\boldsymbol{x}^k_{\boldsymbol{u}},\, \boldsymbol{u}^k) \coloneqq \left(\tilde{\boldsymbol{x}}^k\right)^{\mathsf{T}} \boldsymbol{Q}\, \tilde{\boldsymbol{x}}^k + \left(\boldsymbol{u}^k\right)^{\mathsf{T}} \boldsymbol{R}\, \boldsymbol{u}^k + \Phi(\boldsymbol{x}^k). \tag{2.20}$$

the value of which is increased by deviations from a reference path, non-zero control action and placement in state space regions characterised by large values of Φ.

The Terminal Cost

The terminal cost (or terminal penalty) from (2.15) is the function $E : X \longrightarrow \mathbf{R}$ which depends only on the terminal state $\boldsymbol{x}^{N-1}$. It is used to insinuate the trajectory towards a particular state. The term conventionally shares the quadratic form with the error and control penalties in the running cost. Specifically,

$$e(\boldsymbol{x}) \coloneqq (\boldsymbol{y} - \boldsymbol{x})^{\mathsf{T}} \boldsymbol{S}\, (\boldsymbol{y} - \boldsymbol{x}) \tag{2.21}$$

with $\boldsymbol{y} \in \Omega \subseteq X$ and weighting matrix $\boldsymbol{S}$ sharing the same properties as $\boldsymbol{Q}$. A lot of stability analysis of N/MPC centres around choosing the terminal set Ω, and the terminal target $\boldsymbol{y}$ need not have any relationship to $\boldsymbol{x}_{\mathrm{ref}}$.

It should be superficially clear that the terminal cost in term is minimised when the optimal state trajectory $x_\star$ ends a closely as possible to y. It is useful to further appreciate that since $x_\star$ is constrained by the state succession relationship, the influence of this term is propagated backward through the trajectory during the minimisation. Speaking qualitatively, this gives the appearance of a somewhat rigidly connected set of points forming the state trajectory. If one were to pinch the end of the ridged line and push it around, the path would give some hindrance as the effect of the constraint resists forbidden motions.[11]

[11] Later on, we will see that this effect I characterise as rigidity, is actually the influence of Lagrange multipliers working to constrain the system during the mathematical optimisation.

Traditionally, N/MPC schemes with guaranteed stability for nonlinear systems impose conditions on the allowed regions for terminus of the state-space trajectory, or other such demanding hypotheses on the system which make the on-line computation of the open loop optimal control difficult. Softly constraining the terminus with the quadratic error penalty eases the calculations, but sacrifices guarantees of stability.

3 The Proposal

In the introductory chapter, §1, I enumerated three primary contributions of my thesis:

(C1) the ***Software Design Description***.

(C2) the *software capital*, and

(C3) novel theoretical research in the field of optimal control systems.

Definition 7. A ***Software Design Description*** is, as outlined in IEEE 1016:2009 is a description of a software product that *can be produced to capture one or more levels of concern with respect to its design subject. These levels are usually determined by the design methods in use or the life cycle context; they have names such as "architectural design," "logical design," or "physical design.* I will use the term to describe, mainly, the concerns of what is done within the software components and their interrelationships, as opposed to *how* anything is done.

In this chapter, I shall attempt to itemise and prioritise some milestones, principles and guidelines for the proposed research. It is important to keep in mind that, at any given stage of research, the forward direction will be determined by intermediate findings. At this point, it is impossible to predict those directions, and making plans that are too specific may, at best, be a misuse of attention and energy. So, I will focus on describing, in general terms, parameters and values that will guide and inform my decisions along the way.

3.1 Research Objectives for the Software Design

ITEM (C1) in my list of contributions is the architectural description of the software framework, the SDD. It can, in places, be difficult to decouple (C1) from (C2), which embodies the actual code. The architecture consists of descriptions that comport with the IEEE standard for software design descriptions (and possibly the ISO/IEC/IEEE standard for software architecture description), and can be implemented in any *Turing complete* language. But the language can have a large impact on the form of patterns and contracts.

For example, in a strongly typed language like C++ or Java, interface specifications are an important component of polymorphism. In *duck typed* languages like

Python, interfaces are meaningless, and inheritance is only used to share implementation between classes.

The SDD will outline the software components, their contractual obligation to the user and their interrelationships. However, the design of any given component should be flexible and open to easy modification and adaptation. So, I expect that some of the strongest contributions will be in the form of patterns and reusable design solutions for common problems in control systems software.

3.2 Research Objectives for the Software Capital

```
    Python 3.5.2
    Type "help", "copyright", "credits" or "license" for more information.
>>> import this

    The Zen of Python, by Tim Peters

    Beautiful is better than ugly.
    Explicit is better than implicit.
    Simple is better than complex.
    Complex is better than complicated.
    Flat is better than nested.
    Sparse is better than dense.
    Readability counts.
    Special cases aren't special enough to break the rules.
    Although practicality beats purity.
    Errors should never pass silently.
    Unless explicitly silenced.
    In the face of ambiguity, refuse the temptation to guess.
    There should be one-- and preferably only one --obvious way to do it.
    Although that way may not be obvious at first unless you're Dutch.
    Now is better than never.
    Although never is often better than *right* now.
    If the implementation is hard to explain, it's a bad idea.
    If the implementation is easy to explain, it may be a good idea.
    Namespaces are one honking great idea -- let's do more of those!
```

Tim Peters
The Zen of Python

I open this section with a poem by the great Tim Peters, the software engineer and computer scientist of `timsort`[1] fame. (And if you did not know about it, yes, you can import the **`this`** module in python, and it prints out Peters' poem.) The poem, called *The Zen of Python*, expresses some values of software craftsmanship that comport well with the goals of the software component of the proposed research. Those values largely emphasise readability and clarity over clever shortcuts.

[1] timsort is the algorithm used in the standard sorting commands in Python, Java SE 7, on Android and in GNU Octave.

As I learn and understand more about functional programming, I realise that there may be value in code that is *hard to explain*, seemingly in contrition to Peters' dictum. But the difficulty in explaining functional designs is not from needless complexity. It comes from a cultural barrier. As students, we simply are not taught to think and understand functional patterns in programming and one incurs that overhead, at least once, when seeing it for the first time. Once someone has overcome the learning curve, functional designs can be very clean and understandable when compared to equivalent procedural code. (For a practical but anecdotal investigation, see the DARPA sponsored study by Hudak and Jones in *Haskell vs. Ada vs. C++ vs. Awk vs. ... An Experiment in Software Prototyping Productivity* [73].)

The oo-paradigm organises code to encapsulate the moving parts. Functional code reduces the number of moving parts. When implemented puritanically, neither of these paradigms is a panacea. The best way to think about predictive control systems is certainly found in a balance between these two philosophies. I wish to explore that balance and describe some rules and patterns which merge the best from each.

Regardless of the principles I uncover in the course of the proposed research, the following list of design goals will remain at high priority:

Compliant. It should comply strictly with ISO/IEC 14882:2017 and the C++ Core Guidelines [49][2]. This will include use of the GSL wherever appropriate.

[2] From the GitHub project description: *The C++ Core Guidelines are a set of tried-and-true guidelines, rules, and best practices about coding in C++*

Practical. Code which is high quality and production ready. It should be efficient enough for real-world use. It should be straightforward to use, read and maintain. It will conform to a formatting style that optimises for readability. My metric for this item will be this question: *can a clever and well motivated graduate student get a project up-and-running in less than a week?*

Unit Tested. The entire codebase, including examples, should be covered by unit tests. If a user begins a new design based on the example code, they can adapt the tests as needed. Of course, 100 % test coverage is a rarely achieved goal, but it is always a good goal to have.

Lightweight. The framework itself should be lean enough that a control system designer can, if they are able to cope with the numerics, deploy the framework on modest hardware, such as Arduino, Raspberry PI, Intel Edison, or TI Beagle Bones. Emphasis will be placed on abstraction that is zero-overhead or low-overhead, but never at the cost of clarity.

Portable. It should compile with the latest GNU Compiler Collection (GCC) C++ compiler and Clang++ compilers. The code will compile without warnings using the `--pedantic` compiler flag.

Pattern-oriented. Emphasis is placed on using patterns in order to make the code more easily understood. The framework needs to be readily extensible by control systems researchers. The use of patterns makes the code more expressive, and suggests good techniques and practices to those extending the code.

Minimal mutable state. In the context of software, a mutable state is a feature of routines that store objects and change values during runtime. Whenever you write code with an assignment outside of initialisation, you are modifying state. I am relatively certain the product of my research will not be without mutable state. But there are good reasons to avoid needless use of state.

Mutable state introduces complexity and opportunities for bugs, especially in parallel applications. Control systems researchers who are extending the code should not have to worry about subtleties of the framework obstructing their research. Where mutable state is sufficiently sensible, I will attempt to abstract it away from the API, so it does not complicate parallelism or distribution of the computational tasks.

Truly stateless programs are a feature of *pure functional programming*. However, it is not practical as a public solution, and a major part of the proposed research is discovering a balance that is practical within the context of modern C++ and the control systems community.

The code will be documented, and a suite of examples will serve as illustrative documentation, templates or starting points for users. Those examples will include common problems from optimal control textbooks. Some of the selected textbook questions include:

- Proportional-Integral-Derivative (PID) control of a damped, driven harmonic oscillator
- NMPC control of a train-like vehicle with throttle and break control. [52, p. 5]
- NMPC control of artificial satellite altitude adjustment, as in [74, p. 1]

3.3 Research Objectives in Control Systems Theory

The following subsections each describe (briefly) an avenue for research in the theory of optimal control and NMPC, with a view toward (C3). However, many of these projects also provide sample code and documentation for contributions (C1) and (C2).

Analysis of Hard and Soft Constraints

While designing the controller for virtualME[3] I created a useful software design for handling artificial potential fields. The software still needs refinement and extension before it is complete.

[3] For details on the virtualME controller, see Appendix C.

One of my interest lies in analysis and design of artificial potential fields, and the possibility of switching constraints between hard and soft flavours during runtime. Since soft constraints are computationally lighter, this can be a route to increasing efficiency and reliability.

Robotic Reflexes

With the NMPC algorithm in dynamic state spaces, or with dynamically configuration artificial potential fields, there is always a concern that the optimisation step may take too long, when the system must react very quickly to avoid disaster. Search based optimisation strategies are faster when the starting *guess* is close to the optimum. The techniques for providing better guesses is called *hot-starting*.

I would like to try using a neural network to give a robotic system a behaviour which resembles our human reflexes. The artificial potential field around the robot can be sampled on a mesh. That mesh can be used as in input to the neural network, which can either hot-start the optimisation step, or even circumvent the optimisation step providing a suitable approximation to the NMPC solution.

Comparison Among Gradient Based Search Strategies

When I was struggling with the original C99 controller, there was a study I was particularly interested in performing. The framework I am designing would make it nearly trivial.

In this study, I would like to extend my already existing gradient based controller (see Appendix C) with a new optimiser. I would like to build an *adaptor*-class[4] for

[4] See the *adaptor pattern* in the GoF book [24].

the GNU Scientific Library (GNU-SL) optimisation routines. It will then be trivial to switch between them.

It would be interesting to explore which routines may be most effective when paired with certain types of artificial potential field. This relates to the analysis of hard and soft constraints, described in the previous subsection.

Comparison Between Gradient and NLP Optimisation

The choice between gradient based and NLP based algorithms is one faced by control designers each time they implement an NMPC controller. Despite that fact, there seems to be a lack of literature comparing the two in a rigorous way.

It is a near certainty that one technique will have advantages over the other in certain conditions. One of my research curiosities is in the possibility of switching between the two representations of the problem, leveraging the advantages of each.

Dynamic model adaptation

Functional software design makes it extremely natural to compose functions. This means that optimisation techniques such as *genetic algorithms* are relatively straightforward to implement.

I would like to design a supervisory algorithm which evolves the predictive model, to learn, improve and adapt the model under dynamic conditions. This research parallels the work done by Bongard, Zykov, and Lipson in [3].

3.4 Sequence of Research Milestones

1. Once I have identified relevant patterns, I can continue designing the final architecture. The patterns will provide a layer of communication through nomenclature, while the more ridged components are expressed through contracts.

2. In the next phase, common textbook example questions will be implemented as examples for use. Since the solutions to these problems are well known and documented, they are good tests. And, their implementation as example code is a useful form of documentation that allows users to focus on implementation without the additional complexity of an unfamiliar problem.

   Some of the selected textbook questions include:

   - PID control of a damped, driven harmonic oscillator
   - NMPC control of a train-like vehicle with throttle and break control. [52, p. 5]
   - NMPC control of artificial satellite altitude adjustment, as in [74, p. 1]

3. At this stage, I should reasses the research avenues for (C3), (as described in §3.3), and determine an appropriate order for implementing those.

4. In the final stage of development, I want to re-implement the virtualME controller and augment it with an algorithm to learn and improve the system model, as described in §3.3.

3.5 Approach and Methodology

I am developing software to further my own research goals, as per §3.3. I have also laid out in several simple example problems as part of the documentation and testing. I have outlined comprehensive unit-test coverage as a goal in §3.2.

This is an ideal situation for a tight cycle of development driven by test-first principles, such as the Test Driven Development (TDD) model of Beck, as expounded in his 2003 book *Test-driven Development: By Example* [8]. The TDD method has proven very effective, and is often considered to be a component of the *Agile software development* and *Extreme programming* development methods, and others.

The benefits of TDD include

- allowing us to catch bugs before the cause problems,
- future-proofs against new bugs in an evolving codebase
- projects an image of dependability and a mechanism for proving it
- facilitates safe re-factoring when users adapt it to their needs.

There are many popular books prescribing TDD, such as Martin's *Clean Code: A Handbook of Agile Software Craftsmanship* [7], or the aforementioned Beck [8] or his later *Implementation Patterns* [9]. But most importantly, there are also academic works advocating for the method. For example, the HydroShare case study in *Software Engineering for Science* by Carver, Hong, and Thiruvathukal [75], or the book *Software Engineering Quality Practices* by Kandt [76]. Kandt's book expounds the use of *extreme programming*, of which TDD is an integral part.

Most pertinently, a 2005 study commissioned by Canada's National Research Council found that TDD can lead to higher productivity [77]. In particular, they conclude

> Our main result is that Test-First programmers write more tests per unit of programming effort. In turn, a higher number of programmer tests lead to proportionally higher levels of productivity. Thus, through a chain effect, Test- First appears to improve productivity. We believe that advancing development one test at a time and writing tests before implementation encourage better decomposition, improves understanding of the underlying requirements, and reduces the scope of the tasks to be performed.

I also want to investigate the fusion of Design By Contract (DbC) [78] with TDD in C++. The C++17 standard has no native support for contract specification, though there are some implementations where some have adapted `assert` for those purposes. However, I think TDD provides a mechanism to define and test contractually defined behaviour. Each *precondition*, *postcondition* and *invariant* can be the subject of a unit-test.

4 Conclusion

The complexity of control systems at and beyond the state of the art is producing proportionally complex software. Management of software complexity is an ongoing issue within the software engineering industry. New software development techniques and architectural styles are being practiced to improve software quality as a means of controlling complexity.

The preceding document has outlined a course of research expanding modern control techniques *and* software designs that enable more rapid progress is NMPC control systems research. I desire to take lessons from the leading edge of the software industry and from computer science to discover techniques and best practices for designing software for NMPC that is scalable, distributable and parallelisable.

For the proposal, I outline three main areas of contribution:

(C1) the Software Design Description.

(C2) the software capital, and

(C3) novel theoretical research in the field of optimal control systems.

In a test-driven development process, I wish to develop a modern framework using functional, reactive programming techniques and the modern C++17 language. The product of that endeavour covers the first two contributions.

The proposed framework is to be the basis for several research projects, *in potentiâ*. For example, one such project investigates the use of artificial neural networks to accelerate the optimisation stage of the algorithm, based on a sampling of the potential field which simulates obstacles in state space. That project, and the others hold novelty in the field of control systems research and represent progress in (C3). They additionally demonstrate the usefulness of the framework which underlies them.

Appendix A

A Brief Discussion for the Purpose of Establishing Notation

We could, of course, use any notation we want; do not laugh at notations; invent them, they are powerful. In fact, mathematics is, to a large extent, invention of better notations.

Richard P. Feynman (1918–1988)
The *great* physicist, my personal hero

Unless otherwise indicated, the mathematics in this document is typeset with strict adherence to the ISO/IEC standard for mathematical signs and symbols (ISO/IEC 80000-2:2009) [79]. Undocumented deviations should be considered a bug. To save the reader the trouble of finding and reading the standard, I shall summarise herein with particular care to point out anything not in common convention in North America.

Quantities which are not variable across time or context (such as immutable constants of mathematics) are set in upright text. For examples, $\exp(1) = \mathrm{e}$ and not the italic e; the ratio of a circle's diameter to its circumference (in flat space) is $\mathrm{\pi}$ and not π; the imaginary unit, defined by $\mathrm{i}^2 = -1$, is not the italic i, and so on. Variables, parameters, contextual constants, running numbers and alike are set in italicized text. For example, $\sum_i a_i = 2 + 4\mathrm{i}$ where i is a counter while i is the imaginary unit.

This rule of italics vs. upright is universally conventional for functions. This is why $\sin(x)$ is correct and the commonly seen italic $sin(x)$ is widely recognized as an error. (Some LaTeX users, particularly those who tend to forget backslashes such as the one in `\sin(x)`, are common offenders.)

Vector and matrix quantities follow the convention of italics vs. upright, and are notated in bold. In addition to being bold, it will generally (but not necessarily) be the case that vectors will be lowercase, while matrices are uppercase:

$$(\boldsymbol{A}\boldsymbol{x})^{\mathsf{T}}\,\boldsymbol{y} = \boldsymbol{x}^{\mathsf{T}}\,(\boldsymbol{A}\boldsymbol{y}). \tag{A.22}$$

(Note, this is the foundational identity of the matrix-transpose of $\boldsymbol{A}$. That is, the matrix $\boldsymbol{A}^{\mathsf{T}}$ that makes the Left Hand Side (LHS) inner product equal to the Right Hand Side (RHS) inner product, for all suitably shaped $\boldsymbol{x}$ and $\boldsymbol{y}$, is by definition the transpose of $\boldsymbol{A}$.)

The $n \times n$ identity matrix is a matrix quantity, (so is set in bold and uppercase), but it is defined for matrices independently of context. As such, its symbol should be bold and upright: $\mathbf{I}_n$[1].

[1] This marks a deviation from the ISO/IEC standard which discloses a bold italicised $\boldsymbol{I}$ for this purpose. I view this as an oversight in the standard since it departs from the convention of printing mathematical constants in Roman. I have found good examples where others make this same deviation.

I find the following convention to be the most awkward to North American eyes. I have had reviewers mistake it for poor typesetting despite the great care I take with my documents. *In following with the convention that context independent objects are upright, the differential operators are typeset upright.* For example:

$$\frac{\mathrm{d}f}{\mathrm{d}x} = \partial_x f = \mathrm{e}^{\mathrm{i}x} \Longrightarrow f(x) = -\mathrm{i}\,\mathrm{e}^{\mathrm{i}x} + c,$$

not,

$$\frac{df}{dx} = \partial_x f = \mathrm{e}^{\mathrm{i}x} \Longrightarrow f(x) = -\mathrm{i}\,\mathrm{e}^{\mathrm{i}x} + c.$$

The differences between the upright partial ∂ and *italic* partial ∂ seem trivially subtle. But the upright infinitesimal $\mathrm{d}x$ avoids confusion with the common notation for a distance metric $d(\boldsymbol{x}, \boldsymbol{y})$. The expression $dd/d\boldsymbol{x}$ is nonsense while $\mathrm{d}d/\mathrm{d}\boldsymbol{x}$ is at least meaningful, even if it is distracting.

A good notation should do some work for you. It frees the mind to bring full attention to bare on the real problem, and even carry you part of the way. The legendary mathematician Bertrand Russell is reputed to have said that *a good notation has a subtlety and suggestiveness which at times make it almost seem like a live teacher.*

In the spirit of the previous paragraph, I feel compelled to comment that I dislike this dot: $\boldsymbol{a} \cdot \boldsymbol{b}$ in the context of the vector *dot* product. The great applied mathematician William Gilbert "Gil" Strang even calls it unprofessional [80, p. 108], though I feel that may be overstating it. Worse than useless, that dot distracts from what really is happening. We have been well trained to understand how matrices multiply. Let us simply write the dot product (or inner product) for two column vectors $\boldsymbol{a}^\mathsf{T}\boldsymbol{b} = \sum_j a_j b_j$. It is unsurprising then that the outer product (or rank one product, or *tensor* product) is $\boldsymbol{a}\boldsymbol{b}^\mathsf{T}$.

The matrix multiplication really lets us connect with the underlying linear algebra. It does work for us, even when we extend linear algebra to functions (that is, functional analysis). Let us think about two functions of a real variable, $f(t)$ and $g(t)$. Let us also discretise by evaluating on a mesh of t that is sampled with interval T. More concisely, $t = kT$ with $k = 0, 1, 2, \ldots$. We can then think of f as a column of values $f_k = f(kT)$. The inner product notation now suggests a convenient form for the inner product of two functions:

$$f^\mathsf{T} g = \sum_j f_j\, g_j.$$

Now, work your way back to the continuous case by allowing $T \longrightarrow 0$, and the above sum condenses to the integral:

$$\lim_{T \longrightarrow 0} \sum_j f_j\, g_j = \int_{-\infty}^{\infty} f(x)\, g(x)\, \mathrm{d}x = \langle f,\ g \rangle.$$

It is short work from here to derive the Fourier or Laplace transformations, and we were well set up for it using the right notation. (If you are unsure about getting Fourier or Laplace out of this, consider taking the inner product of a function of interest with each member of a set of orthogonal basis functions, such as the complex sinusoids. You are decomposing the function vector in a Hilbert space of functions just as one would find the x-, y- and z-components of a vector in Euclidean 3-space.)

I digress for a moment to appreciate that combining the above definition of the inner product with the transpose identity (A.22) can be used to derive integration by parts:

$$(\boldsymbol{A}\,\boldsymbol{x})^{\mathsf{T}}\,\boldsymbol{y} = \boldsymbol{x}^{\mathsf{T}}\,(\boldsymbol{A}\,\boldsymbol{y})$$
$$\int_{-\infty}^{\infty} \frac{\mathrm{d}f}{\mathrm{d}t}\,g(t)\,\mathrm{d}t = \int_{-\infty}^{\infty} f(t)\left(-\frac{\mathrm{d}g}{\mathrm{d}t}\right)\,\mathrm{d}t.$$

The reader can expect two notations for the differential operation. I could hardly be accused of rebellion for adopting the common notation of Leibniz,

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mathrm{d}}{\mathrm{d}x}\,f,$$

for the derivative of y with respect to x. I will also exercise the more compact notation for the same thing:

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \partial_x y.$$

The former notation has the disadvantage of becoming optically obscure if x were to have super/sub-scripts, so in such cases I will fall back to Leibniz.

The notation with the *del*, ∂, is popular in field theory (and related areas of physics) and I believe is based on the notation of Oliver Heaviside which is the same except using a D in place of the del.

The set of real numbers is denoted by $\mathbf{R}$. The symbol $\mathbf{R}_+$ is a common notation for the positive real numbers. It is also a common short hand for the additive group of real numbers, $(\mathbf{R}, +)$. I will surely reference the positive real numbers much more frequently, so the compact notation $\mathbf{R}_+$ is reserved for that. Similarly, $\mathbf{Z}_+$ will represent positive integers, which some may refer to as natural numbers. Still others would not recognize these as natural numbers as there is no general agreement on whether natural numbers should be the positive integers, or the non-negative integers. To avoid confusion, I notate the non-negative integers as $\mathbf{Z}_{\geq 0} = \{0\} \cup \mathbf{Z}_+$, and avoid the phrase *natural numbers* everywhere but this paragraph.

The Kronecker-delta, $\delta^n_m = \delta_{nm}$, (which evaluates to 1 if and only if $n = m$ and 0 otherwise), is upright. So too, is the Dirac-delta, $\delta(x)$, which is zero everywhere except at $x = 0$ with $\int_{-\infty}^{\infty} \delta(x)\,\mathrm{d}x = 1$.

In this document, there will be references to discretely sampled trajectories through (flat) spaces of arbitrary dimension. For example, a trajectory through a 7-dimensional Euclidean space. These sampled trajectories will be collected as a time series into a structure that looks a lot like a matrix: a row of column-vectors forming a rank-2 array. These will typically be notated with a lowercase bold italicized Latin symbol, such as $\boldsymbol{x}$, or $\boldsymbol{u}$—breaking the convention of uppercase for matrices. This departure from the convention indicates that these structures are most usefully thought of as a time series of column-vectors. Take such an entity $\boldsymbol{a} \in \mathbf{R}^{n\times N}$. The k-th column-vector in the series will be notated as $\boldsymbol{a}^k$ with $0 \leq k \leq N-1$. Since these are vector quantities, there should be no confusion with exponentiation. The superscript notation frees the subscript position for notational embellishments. This sort of notation is not strange, it is simply borrowed from tensor notation. In tensor notation, there is a difference between super-scripts (which indicate covariant components) and subscripts (which indicate contravariant components). However, we will be working in flat spaces where the two are equivalent. However, I would rather not even confuse the issue with such concerns. Simply keep in mind that superscripts on vectors indicate a column-number in a series of column vectors.

Parentheses, brackets and braces will be used with consistency. Parentheses, $(\cdot)$, will be used as traditional delimiters.

Braces, $\{\cdot\}$, identify sets. For example, $\{a_1,\ a_2, \ldots, a_N\}$. I also make use of the ranged brace notation, which expresses that last set as $\{a_k\}_{k=1}^{N}$.

Brackets, $[\,\cdot\,]$, will be used to enclose composite structures such as vectors and matrices, and they will also be used as outer delimiters for tall set operators, such as $\sum$, $\prod$, $\int$, and $\bigcup$. For example, the definition of the Euclidean metric:

$$d(\boldsymbol{a},\, \boldsymbol{b}) = \|\boldsymbol{a} - \boldsymbol{b}\| = \left(\sum_i \left[a_i - b_i\right]^2\right)^{1/2} \tag{A.23}$$

Of course, that definition for the Euclidean metric is based on the Euclidean, or l^2-norm:

$$\|\boldsymbol{x}\| = \left(\sum_i x_i^2\right)^{1/2}.$$

On a normed space Ω, the norm is written $\|\cdot\|_\Omega$. Similarly, the metric will be written $d_\Omega(\cdot,\, \cdot)$. Of course, that norm and metric may or may not be Euclidean, and the presence of that subscript should incite to question.

The class of n-times continuously differentiable functions is denoted $\mathbf{C}^n$. The derivatives should be bounded so that the supnorm,

$$\|f\|_{\mathrm{C}^k(\Omega)} = \sum_{n=0}^{k} \sup_{x\in\Omega} \left\|\frac{\mathrm{d}^n f}{\mathrm{d}x^n}\right\|,$$

can complete the Banach space, which has important analytical consequences.

A.1 conventions *for* matrix calculus

Two conventions exist for organising the calculus on matrix and vector quantities. Namely, these are the *numerator*- and *denominator*-layouts. The two layouts are related (mostly) by transposition. For this document I adopt the numerator layout, which may be exemplified in the following five cases.

1. For a scalar valued function of a vector, $y : \mathbf{R}^n \longrightarrow \mathbf{R}$, the gradient is a row-vector:

$$\frac{\partial y}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \frac{\partial y}{\partial x_2} & \cdots & \frac{\partial y}{\partial x_n} \end{bmatrix}.$$

2. For a vector valued function of a scalar, $\boldsymbol{y} : \mathbf{R} \longrightarrow \mathbf{R}^n$, the derivative is a column-vector:

$$\frac{\partial \boldsymbol{y}}{\partial x} = \begin{bmatrix} \partial y_1/\partial x \\ \partial y_2/\partial x \\ \vdots \\ \partial y_n/\partial x \end{bmatrix}.$$

3. For a vector valued function of a vector, $\boldsymbol{y} : \mathbf{R}^n \longrightarrow \mathbf{R}^m$, the derivative is the matrix:

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \frac{\partial y_m}{\partial x_2} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}.$$

4. For a scalar valued function of a matrix, $y : \mathbf{R}^{m\times n} \longrightarrow \mathbf{R}$, the derivative is a matrix:

$$\frac{\partial y}{\partial \boldsymbol{X}} = \begin{bmatrix} \frac{\partial y}{\partial X_{11}} & \frac{\partial y}{\partial X_{21}} & \cdots & \frac{\partial y}{\partial X_{m1}} \\ \frac{\partial y}{\partial X_{12}} & \frac{\partial y}{\partial X_{22}} & \cdots & \frac{\partial y}{\partial X_{m2}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y}{\partial X_{1n}} & \frac{\partial y}{\partial X_{2n}} & \cdots & \frac{\partial y}{\partial X_{mn}} \end{bmatrix}.$$

5. And finally, for a matrix valued function of a scalar, $Y : \mathbf{R} \longrightarrow \mathbf{R}^{m\times n}$, the derivative (which is only defined in numerator-layout) is a matrix:

$$\frac{\partial \boldsymbol{Y}}{\partial x} = \begin{bmatrix} \frac{\partial Y_{11}}{\partial x} & \frac{\partial Y_{12}}{\partial x} & \cdots & \frac{\partial Y_{1n}}{\partial x} \\ \frac{\partial Y_{21}}{\partial x} & \frac{\partial Y_{22}}{\partial x} & \cdots & \frac{\partial Y_{2n}}{\partial x} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial Y_{m1}}{\partial x} & \frac{\partial Y_{m2}}{\partial x} & \cdots & \frac{\partial Y_{mn}}{\partial x} \end{bmatrix}.$$

A quantity simply stated to be a *vector* should be considered a column-vector unless otherwise indicated.

Appendix B

List of Acronyms, Initialisms and Selected Jargon

API Application Programming Interface

C99 ISO/IEC 9899:1999. (See reference [16].)

C++11 ISO/IEC 14882:2011. (See reference [13].)

C++14 ISO/IEC 14882:2014. (See reference [45].)

C++17 ISO/IEC 14882:2017. (See reference [14].)

C++98 ISO/IEC 14882:1998. (See reference [81].)

CPU Central Processing Unit

DARPA Defence Advanced Research Projects Agency

DbC Design By Contract

FRP Functional Reactive Programming

GCC GNU Compiler Collection

GNU-SL GNU Scientific Library

GoF Gang *of* Four. Named in reference to the four authors of the book *Design Patterns: Elements of Reusable Object-Oriented Software* [24].

GPU Graphics Processing Unit

GSL C++ Guideline Support Library

IEC International Electrotechnical Commission

IEEE Institute of Electrical and Electronic Engineers

IEEE 1016:2009 the IEEE standard for software design descriptions

ISO/IEC/IEEE 420101:2011 the ISO/IEC/IEEE standard for software architecture description

ISO International Organisation *for* Standardisation

ISO/IEC 80000-2:2009 the ISO/IEC standard for mathematical signs and symbols

MATLAB MathWorks® MATLAB® (SOFTWARE)

MIMO Multiple Input Multiple Output

MPC Model Predictive Control

N/MPC Linear or Nonlinear Model Predictive Control

NIST National Institute of Standards and Technology

NLP NonLinear Program[ming]

NMPC Nonlinear Model Predictive Control

OCP Optimal Control Problem

OCP$_N$ NMPC Optimal Control Problem

OO Object-Oriented

PID Proportional-Integral-Derivative. (Controller)

SDD Software Design Description. As described in the IEEE standard for software design descriptions.

TDD Test Driven Development

UML Unified Modeling Language

VVER Water-Water Energetic Reactor. NB: the 'V's come from the from the Russian word for water, 'водо', where the letter 'в' is prodounced as an English *v*. The full Russian phrase is 'Водо-Водяной Энергетический Реактор' (ВВЭР), or Anglo-phoenetically, *Vodo-Vodyanoi Energetichesky Reaktor.*

Appendix C

Our Robotics Publications

This appendix indexes copies of the following two articles:

1. T. A. V. Teatro and J. M. Eklund, “Nonlinear model predictive control for omnidirectional robot motion planning and tracking,” English, in *2013 26th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, IEEE, May 2013, pp. 1–4, ISBN: 978-1-4799-0033-6. DOI: 10.1109/CCECE.2013.6567713

2. T. A. V. Teatro, J. M. Eklund, and R. Milman, “Nonlinear model predictive control for omnidirectional robot motion planning and tracking with avoidance of moving obstacles,” *Canadian Journal of Electrical and Computer Engineering*, vol. 37, no. 3, pp. 151–156, 2014, ISSN: 0840-8688. DOI: 10.1109/CJECE.2014.2328973. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6978004

If you are reading a hard copy of this document, an individually stapled copies of these documents are placed hereafter. Readers of an electronic copy of this document may find them in in PDF form in the same directory.

Appendix D

Our Reactor Control Publication

This appendix indexes a copy of the following article:

- T. A. V. Teatro, P. McNelles, and J. M. Eklund, “A formulation of rod based nonlinear model predictive control of nuclear reaction with temperature effects and xenon poisoning,” in *23rd Int. Conf. on Nucl. Eng.*, Chiba, Japan: JSME, 2015

If you are reading a hard copy of this document, an individually stapled copy of this document is placed hereafter. Readers of an electronic copy of this document may find it in in PDF form in the same directory.

Appendix E

Our Software Design Publication

This appendix indexes a copy of the following article:

- T. A. V. Teatro and J. M. Eklund, “An object oriented framework for a generic model predictive controller,” in *2016 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, IEEE, May 2016, pp. 1–4, ISBN: 978-1-4673-8721-7. DOI: 10.1109/CCECE.2016.7726652

If you are reading a hard copy of this document, an individually stapled copy of this document is placed hereafter. Readers of an electronic copy of this document may find it in in PDF form in the same directory.

References

[1] H. Gill and J. Bay, “Introduction to the sec vision,” in *Software-Enabled Control: Information Technology for Dynamical Systems*, T. Samad and G. Balas, Eds., Wiley-IEEE Press, 2003, ch. 1, pp. 3–8, ISBN: 978-0-471-23436-4.

[2] D. C. Dennett, *Kinds Of Minds: Toward An Understanding Of Consciousness*. Basic Books, 1997, ISBN: 0465073514, 9780465073511.

[3] J. Bongard, V. Zykov, and H. Lipson, “Resilient machines through continuous self-modeling.,” *Science (New York, N.Y.)*, vol. 314, no. 5802, pp. 1118–21, Nov. 2006, ISSN: 1095-9203. DOI: 10.1126/science.1133687.

[4] A. F. T. Winfield, C. Blum, and W. Liu, “Towards an ethical robot: internal models, consequences and ethical action selection,” in *Advances in Autonomous Robotics Systems*, M. Mistry, A. Leonardis, M. Witkowski, and C. Melhuish, Eds., Cham: Springer International Publishing, 2014, pp. 85–96, ISBN: 978-3-319-10401-0. DOI: 10.1007/978-3-319-10401-0_8.

[5] C. J. Ostafew, A. P. Schoellig, and T. D. Barfoot, “Robust constrained learning-based nmpc enabling reliable mobile robot path tracking,” *International Journal of Robotics Research*, vol. 35, no. 13, pp. 1547–1563, 2016, ISSN: 0278-3649. DOI: 10.1177/0278364916645661.

[6] S. McConnell, *Code Complete*, 2nd. Microsoft Press, 2004, ISBN: 0735619670, 9780735619678.

[7] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2009, ISBN: 0136083250, 9780136083252.

[8] K. Beck, *Test-driven Development: By Example*. Addison-Wesley Professional, 2003, ISBN: 0321146530.

[9] ——, *Implementation Patterns*, 1st. Addison-Wesley Professional, 2007, ISBN: 9780321413093.

[10] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, and E. Gamma, *Refactoring: Improving the Design of Existing Code*, 1st. Addison-Wesley Professional, 1999, ISBN: 978-0201485677.

[11] M. C. Feathers, *Working Effectively with Legacy Code*, 1st. Prentice Hall, 2004, ISBN: 0076092025986.

[12] T. Samad and G. J. Balas, Eds., *Software-Enabled Control: Information Technology for Dynamical Systems*. Wiley-IEEE Press, 2003, ISBN: 978-0-471-23436-4.

[13] ISO/IEC 14882:2011, *Information technology—programming languages—C++*, 2011.

[14] ISO/IEC 14882:2017, *Information technology—programming languages—C++*, 2017.

[15] A. Church, “The calculi of lambda-conversion,” in *Annals of mathematics studies*, Princeton University Press, 1941, ISBN: 0691083940, 9780691083940.

[16] ISO/IEC 9899:1999, *Information technology—programming languages—C*, 1999.

[17] T. A. V. Teatro and J. M. Eklund, “Nonlinear model predictive control for omnidirectional robot motion planning and tracking,” English, in *2013 26th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, IEEE, May 2013, pp. 1–4, ISBN: 978-1-4799-0033-6. DOI: 10.1109/CCECE.2013.6567713.

[18] T. A. V. Teatro, J. M. Eklund, and R. Milman, “Nonlinear model predictive control for omnidirectional robot motion planning and tracking with avoidance of moving obstacles,” *Canadian Journal of Electrical and Computer Engineering*, vol. 37, no. 3, pp. 151–156, 2014, ISSN: 0840-8688. DOI: 10.1109/CJECE.2014.2328973. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6978004.

[19] T. A. V. Teatro, P. McNelles, and J. M. Eklund, “A formulation of rod based nonlinear model predictive control of nuclear reaction with temperature effects and xenon poisoning,” in *23rd Int. Conf. on Nucl. Eng.*, Chiba, Japan: JSME, 2015.

[20] T. A. V. Teatro and J. M. Eklund, “An object oriented framework for a generic model predictive controller,” in *2016 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, IEEE, May 2016, pp. 1–4, ISBN: 978-1-4673-8721-7. DOI: 10.1109/CCECE.2016.7726652.

[21] IEEE 1016-2009, *Standard for information technology—systems design—software design descriptions*, 2009.

[22] ISO/IEC/IEEE 42010, *Systems and software engineering—architecture description*, 2011.

[23] ISO/IEC 19501, *Unified modeling language specification*, 2005.

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, Nov. 1994, ISBN: 0201633612.

[25] J. Hughes, “Why functional programming matters,” *The Computer Journal*, vol. 32, no. 2, pp. 98–107, Feb. 1989. DOI: 10.1093/comjnl/32.2.98.

[26] D. A. Turner, “Some history of functional programming languages,” in, Springer Berlin Heidelberg, 2013, pp. 1–20. DOI: 10.1007/978-3-642-40447-4_1.

[27] A. Church, “An unsolvable problem of elementary number theory,” *American Journal of Mathematics*, vol. 58, no. 2, Apr. 1936, ISSN: 00029327. DOI: 10.2307/2371045.

[28] P. J. Landin, “The next 700 programming languages,” *Communications of the ACM*, vol. 9, no. 3, pp. 157–166, Mar. 1966, ISSN: 00010782. DOI: 10.1145/365230.365257.

[29] J. Backus, “Can programming be liberated from the von neumann style?: a functional style and its algebra of programs,” *Communications of the ACM*, vol. 21, no. 8, pp. 613–641, Aug. 1978. DOI: 10.1145/359576.359579.

[30] C. Allen and J. Moronuki, *Haskell Programming from first principles*. 2016. [Online]. Available: http://haskellbook.com/.

[31] D. I. Spivak, *Category theory for the sciences*. Cambridge, Massachusetts: The MIT Press, 2014, ISBN: 978-0-262-02813-4.

[32] I. Čukić, *Functional Programming in C++*. Manning Publications, 2017, ISBN: 9781617293818. [Online]. Available: https://www.manning.com/books/functional-programming-in-cplusplus.

[33] C. Elliott and P. Hudak, “Functional reactive animation,” *ACM SIGPLAN Notices*, vol. 32, no. 8, pp. 263–273, 1997.

[34] Z. Wan and P. Hudak, “Functional reactive programming from first principles,” *Acm sigplan notices*, 2000.

[35] H. Nilsson, A. Courtney, and J. Peterson, “Functional reactive programming, continued,” in *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell’02)*, Pittsburgh, Pennsylvania, USA: {ACM} Press, Oct. 2002, pp. 51–64.

[36] J. Peterson, P. Hudak, A. Reid, and G. Hager, “Fvision: a declarative language for visual tracking,” in *Proceedings of PADL’01: 3rd International Workshop on Practical Aspects of Declarative Languages*, Jan. 2001, pp. 304–321.

[37] J. Peterson and G. Hager, “Monadic robotics,” in *Proceedings of DSL 99: Usenix Conference of Domain-Specific Languages*, Oct. 1999.

[38] J. Peterson, G. Hager, and P. Hudak, “A language for declarative robotic programming,” in *International Conference on Robotics and Automation*, 1999.

[39] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson, “Arrows, robots, and functional reactive programming,” in *Summer School on Advanced Functional Programming 2002, Oxford University*, ser. Lecture Notes in Computer Science, vol. 2638, Springer-Verlag, 2003, pp. 159–187.

[40] H. Nilsson, J. Peterson, and P. Hudak, “Functional hybrid modeling,” in *Proceedings of PADL’03: 5th International Workshop on Practical Aspects of Declarative Languages*, Springer Verlag LNCS 2562, Jan. 2003, pp. 376–390.

[41] A. Courtney, H. Nilsson, and J. Peterson, “The {y}ampa arcade,” in *Proceedings of the 2003 {ACM SIGPLAN} {H}askell Workshop ({H}askell’03)*, Uppsala, Sweden: {ACM} Press, Aug. 2003, pp. 7–18.

[42] J. Peterson, V. Trifonov, and A. Serjantov, "Parallel functional reactive programming," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 1753, Springer Verlag, 2000, pp. 16–31.

[43] L. Al-Khanji, "Expressing functional reactive programming in c++," Master's Thesis, University of Oulu, 2015.

[44] *David sankel's sbase c++ library*, https://bitbucket.org/camior/sbase.

[45] ISO/IEC 14882:2014, *Information technology—programming languages—C++*, 2014.

[46] B. Stroustrup, *The C++ Programming Language*, 4th. Addison-Wesley, 2013, ISBN: 0321563840, 9780321563842.

[47] ——, *A Tour of C++*. Addison-Wesley, 2013, ISBN: 0133549003, 9780133549003.

[48] S. Meyers, *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. Sebastopol, CA: O'Reilly Media Inc., 2014, ISBN: 1491908424, 9781491908426.

[49] H. Sutter and B. Stroustrup, *C++ Core Guidelines*, http://github.com/isocpp/CppCoreGuidelines.

[50] *C++ Guideline Support Library*, http://github.com/Microsoft/GSL.

[51] A. Williams, *C++ Concurrency in Action: Practical Multithreading*. Manning, 2012, p. 503, ISBN: 1933988770.

[52] D. E. Kirk, *Optimal Control Theory: An Introduction*. Dover Publications, 2004, ISBN: 0486434842.

[53] I. M. Gelfand, S. V. Fomin, and R. A. Silverman, *Calculus of Variations*, ser. Dover Books on Mathematics. Dover Publications, 2000, ISBN: 9780486414485.

[54] R. F. Stengel, *Optimal Control and Estimation*, ser. Dover books on advanced mathematics. Dover Publications, 1986, ISBN: 9780486682006.

[55] M. Vidyasagar, *Nonlinear Systems Analysis*, 2nd. SIAM, 2002, ISBN: 0-89871-526-1.

[56] A. V. Rao, "A survey of numerical methods for optimal control," *Advances in the Astronautical Sciences*, vol. 135, no. 1, pp. 497–528, 2009.

[57] M. Cannon, "Efficient nonlinear model predictive control algorithms," *Annual Reviews in Control*, vol. 28, no. 2, pp. 229–237, Jan. 2004, ISSN: 13675788. DOI: 10.1016/j.arcontrol.2004.05.001.

[58] D. Q. Mayne, "Model predictive control: recent developments and future promise," *Automatica*, vol. 50, no. 12, pp. 2967–2986, 2014, ISSN: 00051098. DOI: 10.1016/j.automatica.2014.10.128.

[59] R. Findeisen and F. Allgöwer, "An introduction to nonlinear model predictive control," in *21st Benelux Meeting on Systems and Control*, Veidhoven, 2002, pp. 119–141.

[60] J. B. Rawlings, "Tutorial overview of model predictive control," *IEEE Control Systems Magazine*, vol. 20, no. 3, pp. 38–52, Jun. 2000, ISSN: 02721708. DOI: 10.1109/37.845037.

[61] L. Grüne and J. Pannek, *Nonlinear Model Predictive Control: Theory and Algorithms*, 1st. Springer, 2011, p. 370, ISBN: 0857295004.

[62] E. F. Camacho and C. Bordons, *Model Predictive Control*, 2nd. London: Springer-Verlag London Ltd., 2007, ISBN: 978-1-85233-694-3.

[63] J. Richalet, A. Rault, J. L. Testud, and J. Papon, "Model predictive heuristic control," *Automatica*, vol. 14, no. 5, pp. 413–428, Sep. 1978, ISSN: 00051098. DOI: 10.1016/0005-1098(78)90001-8.

[64] J. M. Eklund, J. Sprinkle, and S. S. Sastry, "Implementing and testing a nonlinear model predictive tracking controller for aerial pursuit/evasion games on a fixed wing aircraft," in *American Control Conference*, IEEE, 2005, pp. 1509–1514, ISBN: 0-7803-9098-9. DOI: 10.1109/ACC.2005.1470179.

[65] M. A. Abbas, "Non-linear model predictive control for autonomous vehicles," PhD thesis, UOIT, 2011. [Online]. Available: http://hdl.handle.net/10155/206.

[66] D. Q. Mayne, J. B. Rawlings, C. V. Rao, and P. O. M. Scokaert, "Constrained model predictive control: stability and optimality," *Automatica*, vol. 36, no. 6, pp. 789–814, Jun. 2000, ISSN: 00051098. DOI: 10.1016/S0005-1098(99)00214-9.

[67] C. P. Jobling, P. W. Grant, H. A. Barker, and P. Townsend, "Object-oriented programming in control system design: a survey," *Automatica*, vol. 30, no. 8, pp. 1221–1261, 1994. DOI: 10.1016/0005-1098(94)90106-6.

[68] M. Boasson, "Control systems software," *IEEE Transactions on Automatic Control*, vol. 38, no. 7, pp. 1094–1106, Jul. 1993, ISSN: 00189286. DOI: 10.1109/9.231463.

[69] T. Keviczky, R. Ingvalson, H. Rotstein, O. R. Natale, G. J. Balas, and A. K. Packard, "Final report: an integrated multi-layer approach to software enabled control: mission planning to vehicle control," *DARPA Software-Enabled Control Program F33615-99-C-1497*, 2004.

[70] J. Peterson, G. Hager, and A. Serjentov, "Composable robot controllers," in *Proceedings 2001 IEEE International Symposium on Computational Intelligence in Robotics and Automation*, IEEE, 2001, pp. 149–154, ISBN: 0-7803-7203-4. DOI: 10.1109/CIRA.2001.1013188.

[71] X. Dai, G. Hager, and J. Peterson, "Specifying behavior in c++," in *Proceedings 2002 IEEE International Conference on Robotics and Automation*, vol. 1, IEEE, 2002, pp. 153–160, ISBN: 0-7803-7272-7. DOI: 10.1109/ROBOT.2002.1013354.

[72] K. Zhang, J. Sprinkle, and R. G. Sanfelice, "Computationally aware switching criteria for hybrid model predictive control of cyber-physical systems," *IEEE Transactions on Automation Science and Engineering*, vol. 13, no. 2, pp. 479–490, Apr. 2016, ISSN: 1545-5955. DOI: 10.1109/TASE.2016.2523341.

[73] P. Hudak and M. P. Jones, "Haskell vs. ada vs. c++ vs. awk vs. ... an experiment in software prototyping productivity," Department of Computer Science, Yale University, New Haven, CT, Research Report YALEU/DCS/RR-1049, Oct. 1994.

[74] R. Vinter, *Optimal Control*. Birkhäuser Boston, 2010, ISBN: 978-0-8176-4990-6. DOI: 10.1007/978-0-8176-8086-2.

[75] J. Carver, N. P. C. Hong, and G. K. Thiruvathukal, *Software Engineering for Science*. Taylor & Francis, CRC Press, 2016, ISBN: 9781498743853.

[76] R. K. Kandt, *Software Engineering Quality Practices*. Auerbach Publications, 2006, ISBN: 9781420031102.

[77] H. Erdogmus, M. Morisio, and M. Torchiano, "On the effectiveness of the test-first approach to programming," *IEEE Transactions on Software Engineering*, vol. 31, no. 3, pp. 226–237, Mar. 2005. DOI: 10.1109/TSE.2005.37.

[78] B. Meyer, "Applying 'design by contract'," *IEEE Computer*, vol. 25, no. 10, pp. 40–51, Oct. 1992. DOI: 10.1109/2.161279.

[79] ISO 80000-2:2009, *Quantities and units—part 2: Mathematical signs and symbols to be used in the natural sciences and technology*, 2009.

[80] W. G. Strang, *Introduction to Linear Algebra*, 4th. Wellesley, MA: Wellesley-Cambridge Press and SIAM, 2009, ISBN: 978-0980232714.

[81] ISO/IEC 14882:1998, *Information technology—programming languages—C++*, 1998.

Index