Dynamic Optimisation using the Julia Programming Language

Nathan Davey

May 20, 2020

Abstract

Some abstract

Nomenclature

α	b

1 Introduction

1.1 Motivation

1.1.1 Dynamic Optimisation

Optimisation problems are the focus of wide range of research fields, and have broad applications to almost any discipline. As such, effective tools for solving optimal problems are extremely desirable, and are employed in fields such as medicine, robotics and aerospace. Specifically, optimal problem solvers allow us to streamline the design an operation of, for example, a reusable spacecraft, or a walking robot, or the layout of a hospital. If the problem has a cost function and can be formulated subject to certain mathematical constraints, it can be optimised.

Dynamic optimisation is a subset of general optimisation problems, where the problem is best thought of as finding the optimal control input that minimises some cost function through statespace subject to system dynamics and constraints. An example would be finding the optimal thrust output over time from the earth to another body which maximises the final orbital radius (as seen in [**?**]).

Dynamic optimisation problems can be solved by the process of transcription. Transcription is the process of translating a continuous control problem into a discrete nonlinear programming problem (NLP) which can then be solved by an NLP solver. A general formulation of the dynamic optimisation problem can be found in section 2.1.

The focus of this project is thus to lay the foundations for a simple but powerful dynamic optimisation toolbox, such that users can spend more time on the problem formulation and results without needing a detailed background in optimisation. As history has shown, more can be achieved when time is spent letting the tools work on the problem rather than trying to make the problem work with the tools.

1.1.2 Julia

The Julia programming language is a product of the desire to have highly performant code in a dynamic, high level format i.e. having your cake and eating it. Julia has been designed with numerical analysis and computational science in mind [**?**], and aims to solve what is referred to as the two language problem. The two language problem is that, with the advent of rapid prototyping languages such as python and MATLAB, code can be written and tested quickly at the cost of scalability. Once the code has been written, core components are translated into a low level but performant languages, e.g. C/C++ or Rust. Julia bridges the gap by having easy to read and prototype code which is performant and even garbage collected. This is achieved by a well thought out architecture and a clever just-in-time (JIT) compiler [**?**]. While code must still be written with a certain level of awareness of lower level processes and implementations, and as such a certain style of programming must be adopted to achieve truly performant code, the main goals of Julia are delivered on to a more than satisfactory level. The gains from Julia by solving this problem is more than just faster transition to production. The ability to have performant, high level code can increase the shareability and modularity of code. For programs written natively in Julia, (providing the source code is easily accessible), users wishing to understand and adapt code packages no longer have to trace through difficult to understand C++ programs and try and guess which parts of code exist purely to speed up performance.

Although Julia takes heavy inspiration from other languages such as C, MATLAB. Python and Lisp (to name a few), through the implementations of its main paradigm it holds its own in the world of dynamic general programming languages. By choosing to diverge from the commonly practiced object oriented programming (OOP) paradigm, Julia presents an alternative and more intuitive interface using multiple dispatch and strong type interfaces.

What is very apparent in Julia is that every line of code has been scrutinised, and each feature thought about to great depth. The effect of this is that code feels like it makes sense, rather than a group of features bundled together into a programming language (see: PHP). In addition, Julia is a general purpose language. So general purpose that this very document has been typeset using Julia. The advantage of this is the ability to apply Julia variety of applications. If all the user wishes to do is simulate a spacecraft in orbit (for example), this benefit is of little consequence. But say the user now wants a live visualisation of the spacecraft. If there already exists a general purpose visualisation package in Julia, hours can be saved from having to develop an interface between your code and the outside world.

2 Background

2.1 Mathematics

2.2 Julia Packages

3 Project Ethos

The main aims of the project are derived from lessons learnt in ICLOCS, and are as follows:

1. Code structure must be modular
2. Code must be sufficiently verbose when handling errors
3. Code must verify data before computation

- JuDO

4. The final package must be user extensible and have the ability to support multiple solvers
5. Location of collocation points for each state must be independent of other states

- MorePolynomials

6. The MorePolynomials package must interface well with the existing Polynomials package

3.1 Code structure must be modular

It has been identified that the ICLOCS package has reached a feature saturation point, such that further expansion would require a large proportion of code to be rewritten. One of the primary aims of this project is to write code which is sufficiently extensible by leveraging features in Julia such as types and multiple dispatch that allow for greater code separation. Examples of this can be found in section **??**, where multiple dispatch and abstract typing allow us to utilise existing functions in the Polynomials.jl package while extending it to work with our own function definitions. This would be far more difficult in a traditional OOP based language.

3.2 Code must be sufficiently verbose when handling errors

4 Results and Discussion

4.1 User interface

4.1.1 Error handling

4.2 Benchmarking and performance gains

5 Next steps

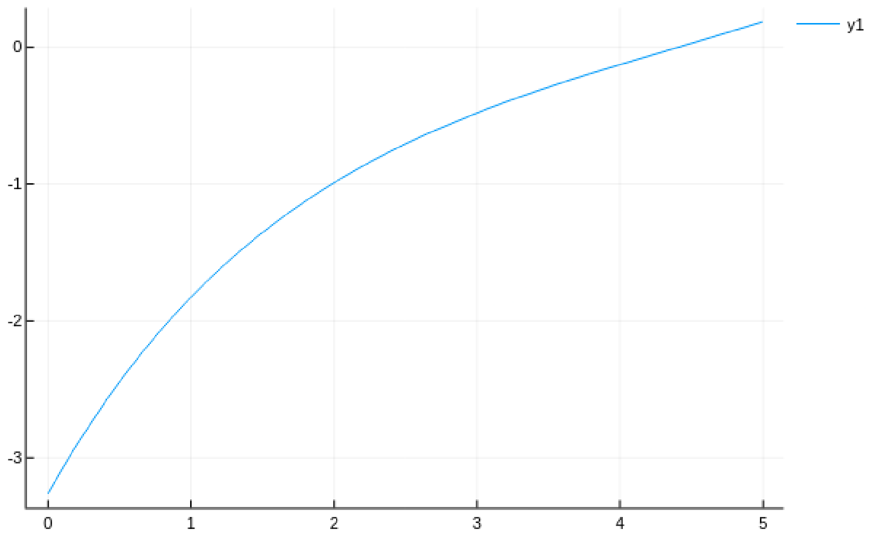

Figure 1: a

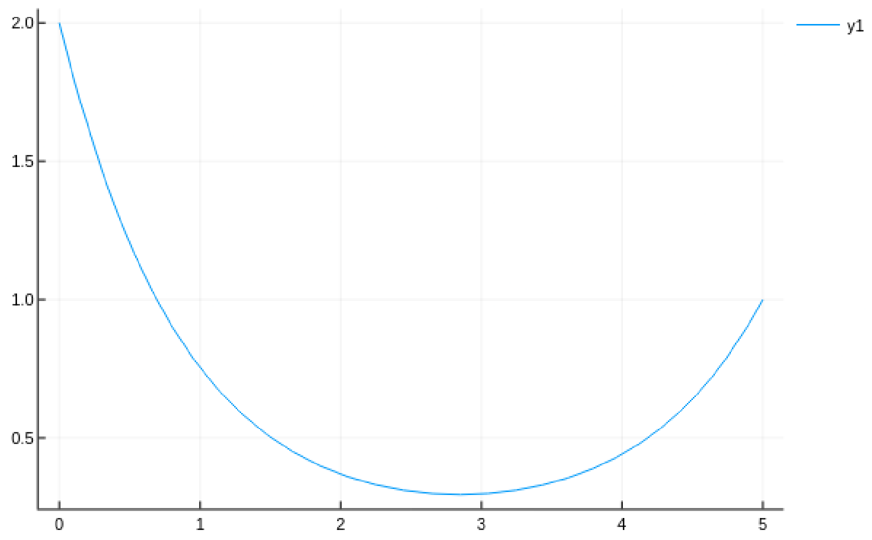

Figure 2: a

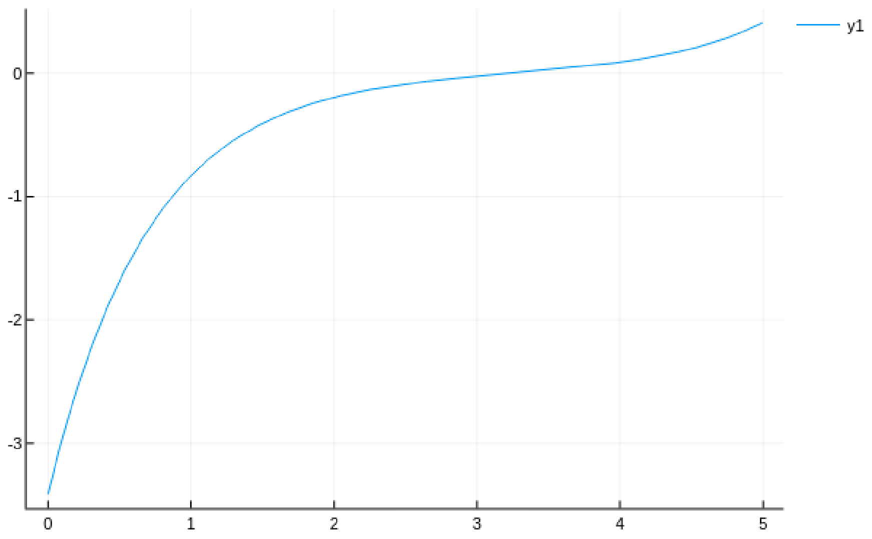

Figure 3: a

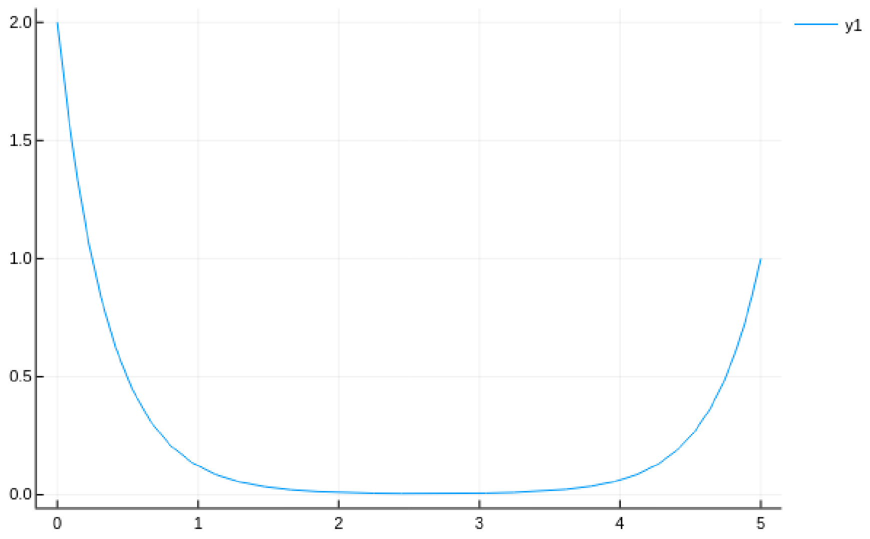

Figure 4: a

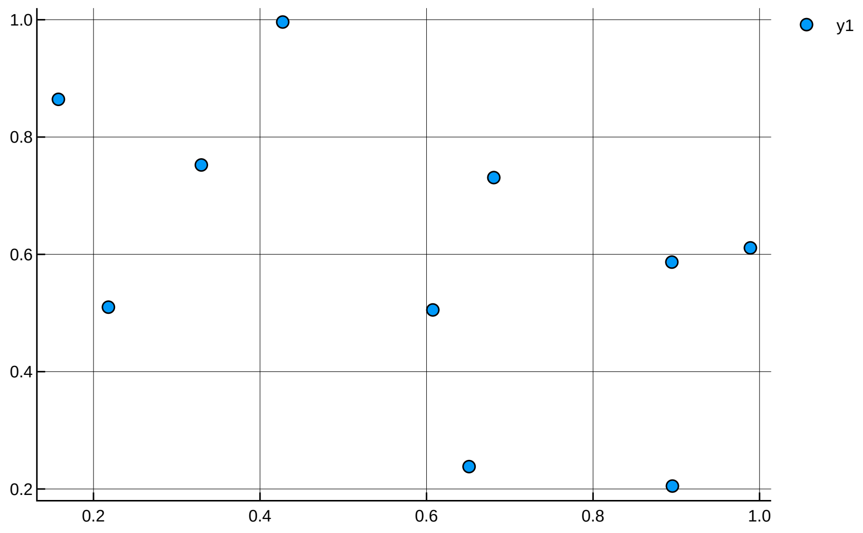

Figure 5: a