Dynamic Optimisation using the Julia Programming Language

Nathan Davey

May 26, 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 Imperial College London Optimal Control Software (ICLOCS)

1.1.3 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 Mathematical Background

2.1 Dynamic optimisation problem formulation

Here we will outline the basic formulation of the dynamic optimisation problem. It is important to note that the main objective of dynamic optimisation is to find a set of optimal control inputs which results in the boundary constraints and system dynamics being satisfied, and a minimisation of the objective function. As such, we can break down the problem into 3 main components:

1. System dynamics
2. Boundary and path constraints
3. Objective function

The following discussion assumes the direct method i.e. transcription. Transcription is the process of converting a continuous dynamic optimisation problem into an array of discrete constraints (which ensure system dynamics are consistent with the state as explained in section **??**) which can be solved by a nonlinear programming (NLP) solver. The direct methods are, well, direct, because we just try and solve the problem by breaking it up into smaller constraint equations. Indirect methods, on the other hand, solve the problem by reformulating (analytically) the problem using a set of first-order necessary conditions for optimality, which can then be solved. Direct methods are currently of more interest as these conditions are often difficult to formulate, and indirect methods are more sensitive to the accuracy of the initial state guess [**?**]. Before we explore the direct method however, it may be beneficial to define what our dynamic optimisation problem will eventually become, a nonlinear programming problem.

2.1.1 Nonlinear programming problem

If we can reformulate our dynamic problem into a NLP problem, we can use a number of widely available solvers to find optimal solutions. The general formulation of the NLP problem is (as given by COIN-OR **??**)

$$\min_{x \in \mathbb{R}^n} \quad J(\boldsymbol{x}) \tag{1}$$

$$\text{s.t.} \quad \boldsymbol{g}^L \leq \boldsymbol{g}(\boldsymbol{x}) \leq \boldsymbol{g}^U \tag{2}$$

$$\boldsymbol{x}^L \leq \boldsymbol{x} \leq \boldsymbol{x}^U \tag{3}$$

Where

$$J : \mathbb{R}^n \to \mathbb{R}$$

is the objective function to be minimised, and

$$g : \mathbb{R}^n \to \mathbb{R}^m$$

are functions that represent nonlinear constraints. In this text we use bold font to denote the existance of an array of either variables or functions. In this particular case, $\boldsymbol{g}$ comprises of multiple constraint functions with multiple inputs and outputs. The target of dynamic optimisation is to discretize the dynamic optimisation problem into a set of constrained points (known as collocation points) which can be expressed as a series of dynamics constraints and solved as a nonlinear problem. The forces that govern the system thus become a discretized set of equality constraints that essentially say the system must behave in this way otherwise the physics would be violated. Another way to put it is rather than giving an input and calculating what the output would be based on extrapolation of the system dynamics through time marching (such as in a shooting method), we evaluate what the physics would have to be at each location in time to satisfy a given input.

We will now go on to explore how the dynamics optimisation problem becomes the NLP problem.

2.1.2 System dynamics

The system dynamics are of key importance in the dynamic optimisation problem, and are what essentially separate it from a standard NLP problem. By system dynmaics, we are referring to the governing equations which describe the evolution of the state over time. Simply put, if $\boldsymbol{x}$ is our state, $\boldsymbol{u}$ is our control input and t represents time we have

$$\frac{\partial \boldsymbol{x}}{\partial t}|_{t_k} = \dot{\boldsymbol{x}}|_{t_k} = \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{u})|_{t_k} \tag{4}$$

Essentially we are saying the change of state through time (derivative) is a function of both the state(s) and control(s) at that instance in time. Let's define our terms a little more thoroughly. Our state is denoted by $\boldsymbol{x}$, which in reality represents an array of states such that

$$x \in \mathbb{R}^m$$

Where m here is the number state discritzation points, otherwise known as mesh points. This is not to be confused with the number of collocation points, which is most cases is the same but in some is different. Collocation points are the points where system dynamics are enforced, but not all points where we approximate state have to comply with system dynamics.

If the number of mesh points is the same for each state it may be convenient to define $\boldsymbol{x}$ as

$$\boldsymbol{x} \in \mathbb{R}^{n \times m}$$

Where n is the number of states (e.g. distance, velocity etc.). In some cases (if the toolbox supports such a formulation) we may require each individual state to have a different mesh size. If this is the case, we can define $\boldsymbol{x}$ as

$$\boldsymbol{x} = \bar{x_i} \in \mathbb{R}^{m_i}, \quad i = 1, 2...n \tag{5}$$

Such that $m \in \mathbb{R}^n$ may contain a different value at each state index i. Here we use the overbar in $\bar{x}$ to represent a vector of states x. Supporting this formulation is indeed one of the aims of the toolbox being developed here and as such this shall be the assumed notation from now on.

Likewise we may use the same definition for our control variables.

$$\boldsymbol{u} = \bar{u_i} \in \mathbb{R}^{m_i}, \quad i = 1, 2...n \tag{6}$$

Our system dynamics function is how we specify the dynamics (in many cases physics) of the system being optimised, and as above is represented as an array of functions for each state derivative. Given we can calculate all state derivatives as a function of the states at one particular time instance, we can have a function mapping that outputs all the state dynamics for a particular state at collocation points given by a time vector $\bar{t} = t_1, t_2...t_{m_i}$

$$\boldsymbol{f} = f_i(\boldsymbol{x}(\bar{t}), \boldsymbol{u}(\bar{t})) : \mathbb{R}^{n \times m_i} \to \mathbb{R}^{m_i}, \quad i = 1, 2...n$$

Where n here is the number of both states and controls, and m_i is the number of collocation points for each state index i. The keen eye will note that for each individual function evaluation, the input array is assumed to have dimensions $n \times m_i$, while in equation 5 and 6 we observe that these arrays do not necessarily have a rectangular representation (rather they are more arrays of varying length vectors). We resolve this by interpolating the values of x and u at some time t.

Putting this all together we have

$$\boldsymbol{x} \begin{cases} \bar{x_1} = & [x_{11}, x_{12}...x_{1m_1}] \\ ... & \\ \bar{x_n} = & [x_{n1}, x_{n2}...x_{nm_n}] \end{cases}$$

$$n = \text{number of states}, m_i = \text{mesh size}, i = 1, 2...n$$

$$\boldsymbol{u} \begin{cases} \bar{u_1} = & [u_{11}, u_{12}...u_{1m_1}] \\ ... & \\ \bar{u_n} = & [u_{n1}, u_{n2}...u_{nm_n}] \end{cases}$$

$$n = \text{number of controls}, m_i = \text{mesh size}, i = 1, 2...n$$

$$\boldsymbol{f}\begin{cases}\dot{\tilde{x}}_1 = & f_1(\boldsymbol{x}(t_1), \boldsymbol{u}(t_1)...\boldsymbol{x}(t_{m_1}), \boldsymbol{u}(t_{m_1})) \\ ... & \\ \dot{\tilde{x}}_n = & f_n(\boldsymbol{x}(t_1), \boldsymbol{u}(t_1)...\boldsymbol{x}(t_{m_n}), \boldsymbol{u}(t_{m_n}))\end{cases}$$

$$n = \text{number of states}, m_i = \text{number of collocation points}, i = 1, 2...n$$

It is important to note that at these collocation points, the system dynamics are enforced i.e. we can verify that the system satisfies the laws of physics (if the problem is physical). However, between collocation points we do not enforce such laws, and solutions must be interpolated, from which we derive a certain amount of error from approximation. Additionally, oscillations can occur between collocation points if the system is significantly ill-defined.

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 mesh 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

When describing the dynamic optimisation problem, the use of verbose error handling and user feedback is extremely useful in guiding the user to both a solvable and well constructed problem. This was particular apparent in the formulation of problems in the ICLOCS package. Often errors in the user defined problem (UDP) would result in error traces that lead deep into the core of the code, providing little help to the user who's only course of action would be to trace the flow of data through the sourcecode of ICLOCS. The aim of this new package is to catch errors elegantly, and give the user contextual feedback.

3.3 Code must verify data before computation

Closely related to well implemented error handling is the verification of the data in the first place. The phrase garbage in garbage out is relevant here. If we ensure the quality of the UDP before any computation, we can ensure the speed and freedom from errors in the results. The reverse is true, that if no verification takes place, the UDP could result in hard to debug errors embedded in the source code, which is not where we want them. By leveraging Julia's explicit type system, we can remove potential conversion errors further down the line by internally specifying the types of the components of the UDP. Further pre solve check can ensure the quality and compatibility of the UDP with internal solver interfaces. This also has the added benefit of speeding up computation when the compiles is aware of the types at runtime.

3.4 The final package must be user extensible and have the ability to support multiple solvers

This is particular important for the longevity of the project. Bespoke programs could be created for each individual solver, but a more ideal solution would be to create a generic framework from which additional solvers can be plugged in. This is highly dependent on the nature of each solver, and additional solver specific syntax may need to be included in the UDP (e.g. if the solver is not black box, additional derivatives may need to be provided unless numerically calculated). However, this goal is more focused towards ease of future development, rather than the user interface.

3.5 Location of mesh points for each state must be independent of other states

By isolation of each state, we can vary the density of mesh points for each state. As such, states which can be approximated by a smaller mesh size do not occupy unnecessary space in memory due to other states requiring a larger mesh size to achieve an accurate approximation. This can be achieved by interpolating the state at collocation points when required.

3.6 The MorePolynomials package must interface well with the existing Polynomials package

As Polynomials.jl is a rapidly developing package, ensuring compatibility allows for easy integration of future features. As Julia is an ever evolving language, by building off prexisting packages we can make the MorePolynomials.jl packages appealing for users already familiar with the framework, while reducing developer load on are part as Polynomials.jl is updated with Julia updates. The disadvantage being we are then at the mercy of the Polynomials.jl package developers. If they decided to change their framework entirely, our package may also need to change. This is not necessarily an issue however as we can specify backwards compatibility.

4 Results and Discussion

4.1 User interface

4.1.1 Error handling

4.2 Benchmarking and performance gains

5 Documentation

6 Next steps

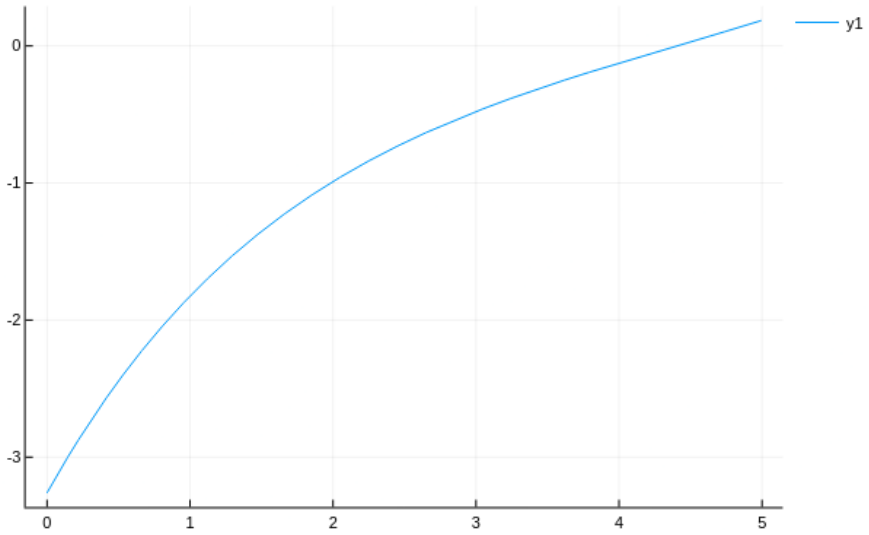

Figure 1: a

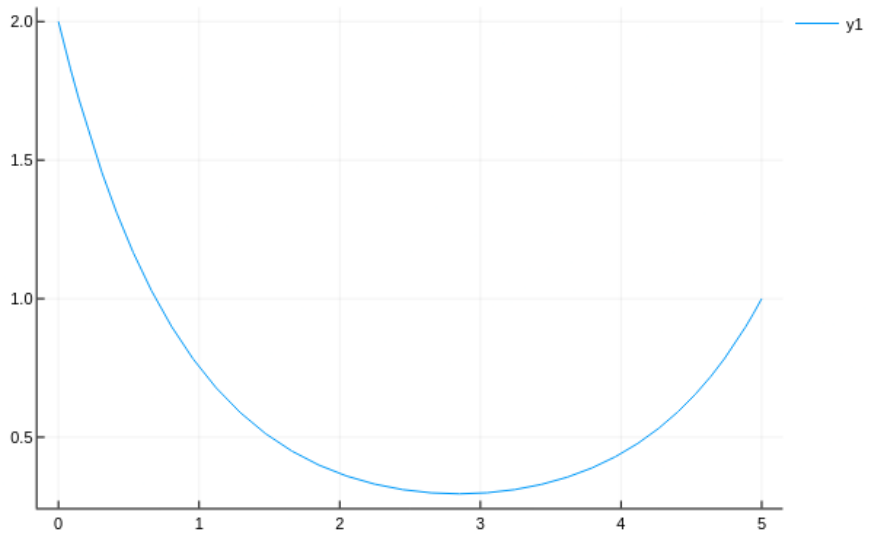

Figure 2: a

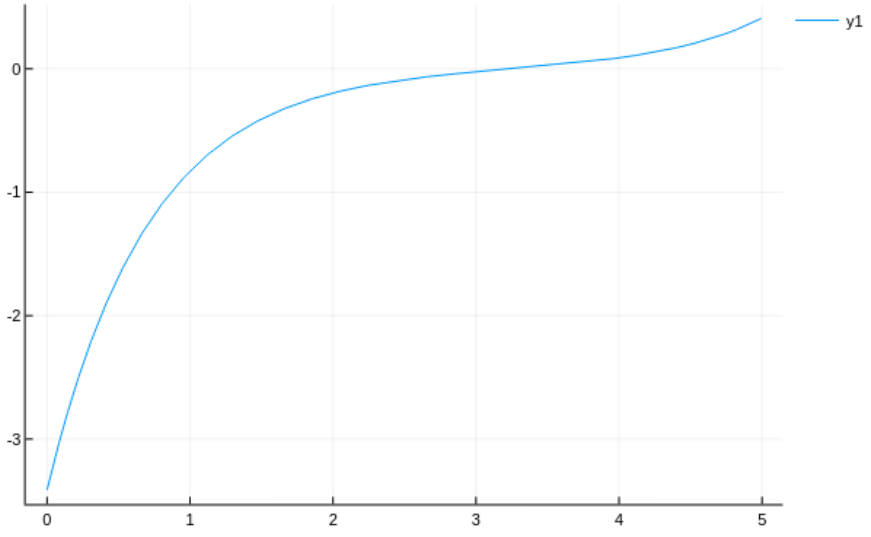

Figure 3: a

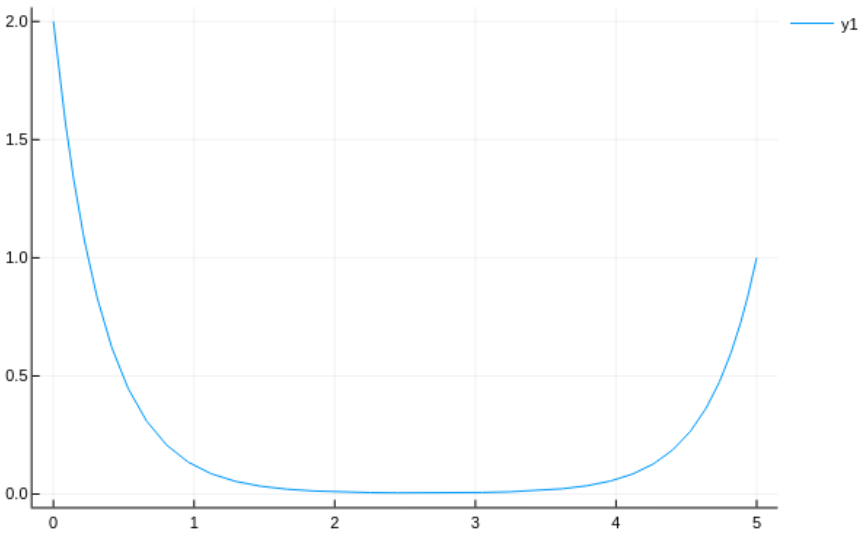

Figure 4: a

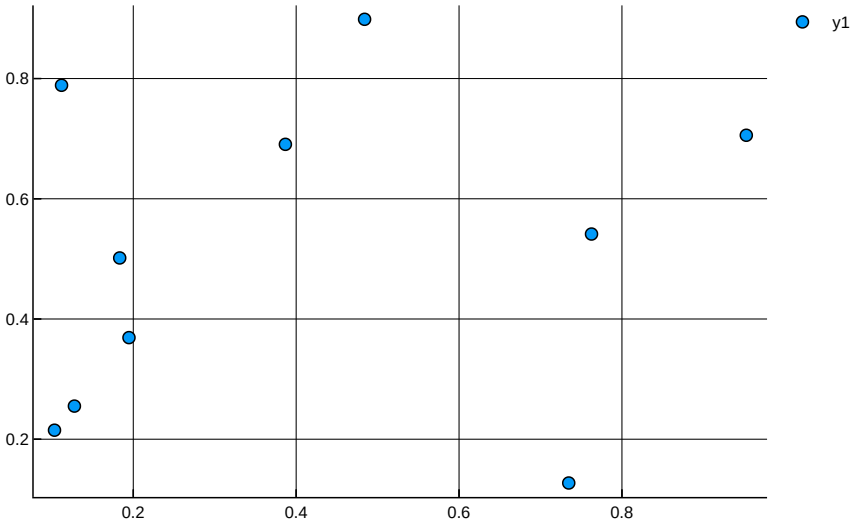

Figure 5: a