

Dynamic Optimisation using the Julia Programming Language

Nathan Davey

May 31, 2020

Abstract

Some abstract

Nomenclature

\boldsymbol{b}	Boundary constraint functions
\boldsymbol{f}	System dynamics functions
\boldsymbol{g}	Constraint functions
\boldsymbol{h}	Path constraint functions
\boldsymbol{u}	Controls
\boldsymbol{x}	States
i	State or control index
J	Objective function
k	Location of a collocation point
N	Total number of collocation points
t	Time

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 Ipopt from COIN-OR ??, a popular NLP solver)

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

$$\text{s.t. } \mathbf{g}_L \leq \mathbf{g}(\mathbf{x}) \leq \mathbf{g}_U \quad (2)$$

$$\mathbf{x}_L \leq \mathbf{x} \leq \mathbf{x}_U \quad (3)$$

Where

$$J : \mathbb{R}^n \rightarrow \mathbb{R},$$

is the objective function to be minimised,

$$\mathbf{x} \in \mathbb{R}^n$$

is an array of decision variables and

$$\mathbf{g} : \mathbb{R}^n \rightarrow \mathbb{R}^m,$$

are functions that represent nonlinear constraints. In this text we use bold font to denote the existence of an array of either variables or functions. In this particular case, \mathbf{g} comprises of multiple constraint functions with multiple inputs and outputs.

The nonlinear constraint may also take the form (which is more common in the syntactic formulation of dynamic optimisation problems)

$$\mathbf{g}(\mathbf{x}) \leq 0 \quad (4)$$

Solvers such as Ipopt additionally replace nonlinear inequality constraints with an equality constraint and a slack variable [?] such that equation 2 becomes $\mathbf{g}(\mathbf{x}) - s = 0$ and $\mathbf{g}^L \leq s \leq \mathbf{g}^U$. As such it makes sense to also include in our formulation the additional equality constraints

$$\mathbf{g}(\mathbf{x}) = 0 \quad (5)$$

The nonlinear constraints may take the form of either equation 4 or 5, or both.

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 dynamics, we are referring to the governing equations which describe the evolution of the state over time. Simply put, if x is our state, u is our control input and t represents time we have

$$\frac{\partial x}{\partial t}|_{t_k} = \dot{x}|_{t_k} = f(x, u)|_{t_k} \quad (6)$$

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 the continuous time variable x , which in reality represents a number of continuous states such that

$$x \in \mathbb{R}^n,$$

where n is the number of states (e.g. distance, velocity etc.). In order to transcribe the problem to a format compatible with our NLP problem, we must discretise this continuous states into a set of decision variables and dynamics constraints. As such, we can represent the state as a function of a set of decision variables. Let's take \tilde{x} to be some interpolating function which interpolates a state for a given number of inputs. We then have

$$\mathbf{x} = [\tilde{x}_i(x_i, t)], \quad x_i \in \mathbb{R}^{m_i}, \quad \tilde{x} : \mathbb{R}^{m_i+1} \rightarrow \mathbb{R}, \quad i = 1, 2 \dots n, \quad (7)$$

such that t is the evaluation time of the interpolating function, $m \in \mathbb{R}^n$ is the number of decision variables for each discretised state and may contain a different value at each state index i , and x_i is the decision variable. As illustrated in section 2.3, function \tilde{x} can be initialised without knowledge of the specifics of t . The number of decision variables is not to be confused with the number of collocation points, which in many 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. Additionally, in the data point formulation (as opposed to the coefficient formulation) an individual data point x_{ij} will consist of a mesh location and state value as below.

$$x_i = [x_{ij} = [t_{ij}, x(t_{ij})]], \quad j = 1, 2, \dots m_i.$$

These discretised states will go on to become decision variables in our NLP formulation. Traditional formulations such as that in ICLOCS require states to be discretised at set mesh interval points such that $\mathbf{x} \in \mathbb{R}^{n \times m}$. By the introduction of an interpolating function \tilde{x} , we are no longer constrained to requiring the number of decision variables in each state to be the same. In our formulation, \tilde{x} is an interpolating polynomial, hence the discussion in section 2.3. This interpolating polynomial may be a function of polynomial coefficients, or interpolating data points, and for the rest of this discussion we will assume they are the latter. As such, x_i represents a set of interpolating data points, and \mathbf{x} represents a vector of state approximating functions such that $\mathbf{x}_i(t_k) = \tilde{x}_i(x_i, t_k)$, where \tilde{x}_i is the i^{th} state approximation evaluated at time t_k .

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

$$\mathbf{u} = [\tilde{u}_i(u_i, t)], \quad u_i \in \mathbb{R}^{m_i}, \quad \tilde{u} : \mathbb{R}^{m_i+1} \rightarrow \mathbb{R}, \quad i = 1, 2 \dots c, \quad (8)$$

where c is the number of controls.

Our system dynamics function is how we specify the dynamics (in many cases physics) of the system being optimised, and is represented as an array of functions, one 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 $t = t_1, t_2 \dots t_{m_i}$

$$\mathbf{f} = f_i(\mathbf{x}(t), \mathbf{u}(t), t) : \mathbb{R}^{(n+c) \times m_i} \rightarrow \mathbb{R}^{m_i}, \quad i = 1, 2 \dots n \quad (9)$$

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 . Note that for each individual function evaluation, the input array is assumed to have dimensions $(n + c) \times m_i$, and as such we use our interpolating functions to find the values of states and controls at locations of $t = t_k, k = 1, 2 \dots m_i$. Also note that, though we can map multiple time inputs to outputs, these mappings must be direct (as per equation 6) i.e. $f(\mathbf{x}(t_k), \mathbf{u}(t_k), t_k) \rightarrow \dot{\mathbf{x}}(t_k)$.

Putting this all together we have

$$\mathbf{x} \begin{cases} \mathbf{x}_1 = & \tilde{x}_1([x_{11}, x_{12} \dots x_{1m_1}], t) \\ \dots & \\ \mathbf{x}_n = & \tilde{x}_n([x_{n1}, x_{n2} \dots x_{nm_n}], t) \end{cases}$$

n = number of states, m_i = states mesh sizes, $i = 1, 2 \dots n$

$$\mathbf{u} \begin{cases} \mathbf{u}_1 = & \tilde{u}_1([u_{11}, u_{12} \dots u_{1m_1}], t) \\ \dots & \\ \mathbf{u}_c = & \tilde{u}_c([u_{c1}, u_{c2} \dots u_{cm_c}], t) \end{cases}$$

c = number of controls, m_i = controls mesh sizes, $i = 1, 2 \dots c$

$$\mathbf{f} \begin{cases} \dot{\mathbf{x}}_{,1} = & f_1(\mathbf{x}(t_1), \mathbf{u}(t_1) \dots \mathbf{x}(t_{m_1}), \mathbf{u}(t_{m_1}), t) \\ \dots & \\ \dot{\mathbf{x}}_{,n} = & f_n(\mathbf{x}(t_1), \mathbf{u}(t_1) \dots \mathbf{x}(t_{m_n}), \mathbf{u}(t_{m_n}), t) \end{cases}$$

n = number of states, m_i = number of state collocation points, $i = 1, 2 \dots n$

We use $\dot{\mathbf{x}}_i$ to represent the continuous dynamics of the state at the collocation points. In our formulation the time vector t does not necessarily have to have the same length or values for any one such function call, only that it must correspond with the number of output variables being assigned. For example, if we are evaluating m_i collocation points, then when calling $f_1(\mathbf{x}(t), \mathbf{u}(t))$, t here must be of length m_1 , but the mesh points of x_1 to x_n do not have to correspond with the collocation points, and if they do not they are interpolated.

Now that we have our system dynamics functions, we need to transform them into a format that can be solved with an NLP solver. This is the process of transcription and we do this by creating a set of equality constraints that enforce system dynamics at our chosen collocation points. At these points 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.

This process generally involves us approximating the derivatives of the state vector at our collocation points, and in some way equating these with our dynamics function. There

are many of these approximations, but the simplest of these (and the one that probably illustrates the point best) is the euler method in equation 10. Note here h_k is the timestep $h_k = t_{k+1} - t_k$ and t_k is the collocation point being evaluated.

$$\frac{\partial \mathbf{x}_i(t_k)}{\partial t} = \mathbf{f}_i(t_k) \approx \frac{\mathbf{x}_i(t_{k+1}) - \mathbf{x}_i(t_k)}{h_k}$$

$$\mathbf{x}_i(t_{k+1}) - \mathbf{x}_i(t_k) - h_k \mathbf{f}_i(t_k) = 0 \quad (10)$$

For the sake of consistency, $\mathbf{x}_i(t) = \tilde{x}_i(x_i, t)$, $\mathbf{f}_i(t) = f_i(\mathbf{x}(t), \mathbf{u}(t), t)$. All other methods are extensions of this principle, but aim to have more accurate estimations of state derivatives at collocation points. Take for example the trapizoidal method

$$\mathbf{x}_i(t_{k+1}) - \mathbf{x}_i(t_k) - \frac{h_k}{2} (\mathbf{f}_i(t_k) + \mathbf{f}_i(t_{k+1})) = 0 \quad (11)$$

Here we take an average of the system dynamics at the k and the $k+1$ collocation point (the derivation comes from the trapizoidal integration approximation, hence trapizoidal method).

Equations 10 and 11 are formulations of the system dynamics as equality constraints which are in the form of equation 5 and thus solvable by an NLP solver. Let's use the simple example of a mass spring damper system to show how system dynamics can be collocated. We assume a fixed starting location, and collocate the dynamics using equation 10. The equations of interest are

$$\mathbf{x} = \begin{bmatrix} x \\ \dot{x} \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$f(x(t_k)) = \begin{bmatrix} \dot{x} \\ \ddot{x} \end{bmatrix} = \begin{bmatrix} x_2 \\ -2\zeta\omega_n x_2 - \omega_n^2 x_1 \end{bmatrix}$$

where x_1 is mass position, $\omega_n = \sqrt{\frac{k}{m}}$ and $\zeta = \frac{c}{2m\omega_n}$ with the constants as defined in the code.


```

function spring()
    k = 5
    m = 0.1
    c = 0.7
     $\omega_n$  = sqrt(k/m)
     $\zeta$  = c / (2*m* $\omega_n$ )
    h_k = 0.1
    t = collect(0:h_k:1)
    x1 = LagrangePoly(t, ones(Float64, length(t))) # x1 state mesh points
    x2 = LagrangePoly(t, zeros(Float64, length(t))) # x2 state mesh points
    function f(x1, x2) # system dynamics
        x1dot = x2
        x2dot = -2 *  $\zeta$  *  $\omega_n$  * x2 -  $\omega_n^2$  * x1
        return x1dot, x2dot
    end
    for i = 1:length(t) - 1
        f1,f2 = f(x1(t[i]), x2(t[i]))
        x1[i+1] = x1(t[i]) + h_k * f1
        x2[i+1] = x2(t[i]) + h_k * f2
    end
    return x1, x2, t
end

x1, x2, t = spring()

```

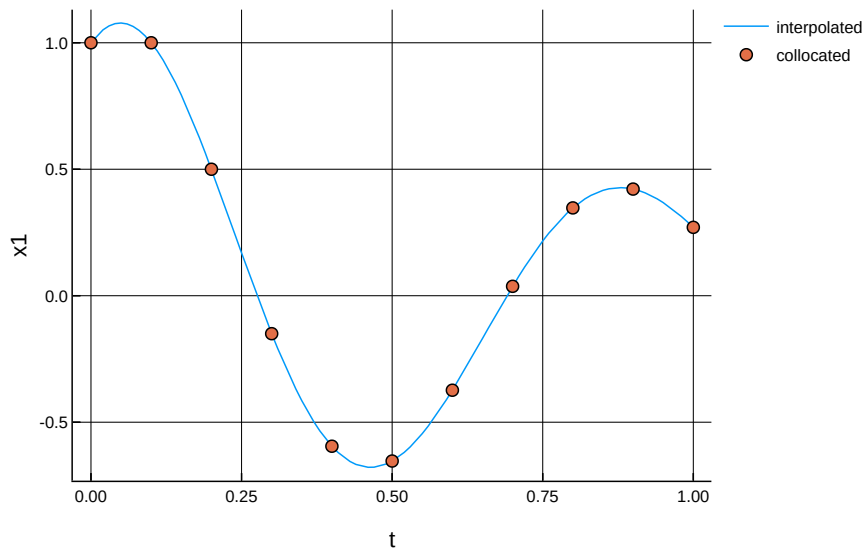


Figure 1: Collocated system dynamics for a mass spring damper

Figure 1 shows the results. The collocation points are marked with a circle, while the interpolated value (in this case using a high order polynomial) are marked by the line. Note the example code here is more akin to a shooting method rather than direct collocation but the results in this case are similar.

Next we will look into formulating our path and boundary constraints for the NLP problem.

2.1.3 Boundary and path constraints

Almost all dynamic optimisation problems will require some form of constraint on our system. Constraints can be categorised as either boundary constraints or path constraints. Boundary constraints occur at the start and endpoints of the problem i.e. at $t = t_1$ and $t = t_F$ (as is consistent with Julia syntax we assume 1 is our initial value rather than 0). Boundary constraints may either be fixed limits or a function of the boundary values. In the case of fixed limits, we have

$$\begin{aligned} t_{1,L} &\leq t_1 \leq t_{1,U} \\ t_{F,L} &\leq t_F \leq t_{F,U} \\ \mathbf{x}_{i,L}(t_1) &\leq \mathbf{x}_i(t_1) \leq \mathbf{x}_{i,U}(t_1) \\ \mathbf{x}_{i,L}(t_F) &\leq \mathbf{x}_i(t_F) \leq \mathbf{x}_{i,U}(t_F) \end{aligned}$$

which are all perturbations of equation 3. Examples of fixed limits would be wanting to finish at a particular location, or with no fuel left etc.

Additionally we may formulate constraints as a function of boundary values

$$\mathbf{b}(t_1, t_F, \mathbf{x}(t_1), \mathbf{x}(t_F)) \leq 0$$

which again can be represented in our NLP solver by equation 4. An example of a boundary constraint function would be a function ensuring that the final speed of a spacecraft matches that of a spacestation in orbit.

Path constraints follow an almost identical formulation but apply to the whole of the path through time and consider control input as well.

$$\mathbf{x}_{i,L} \leq \mathbf{x}_i(t) \leq \mathbf{x}_{i,U} \tag{12}$$

$$\mathbf{u}_{i,L} \leq \mathbf{u}_i(t) \leq \mathbf{u}_{i,U} \tag{13}$$

$$\mathbf{h}(t, \mathbf{x}(t), \mathbf{u}(t)) \leq 0 \tag{14}$$

A direct time mapping is once again assumed for h as described below equation 9. For the sake of completeness equations 12 and 13 correspond to equation 3 and equation 14 to equation 4 in our NLP formulation. Again let's look at a graphical representation to illustrate this using the example from figure 1. Lets add the following constraints to our problem.

$$-1 \leq \mathbf{x}_1(t_k) \leq 1 \tag{15}$$

$$h_1(t_k, \mathbf{x}(t_k), \mathbf{u}(t_k)) = \mathbf{x}_1(t_k) + t_k - 1.5 \leq 0 \tag{16}$$

For the example h_1 could represent a desire for the oscillations to grow smaller with time, while the path constraint could represent physical stops on the system which stop the spring

moving beyond this point. We can Taylor the constraints to fit both the problem dynamics and our specific goals.

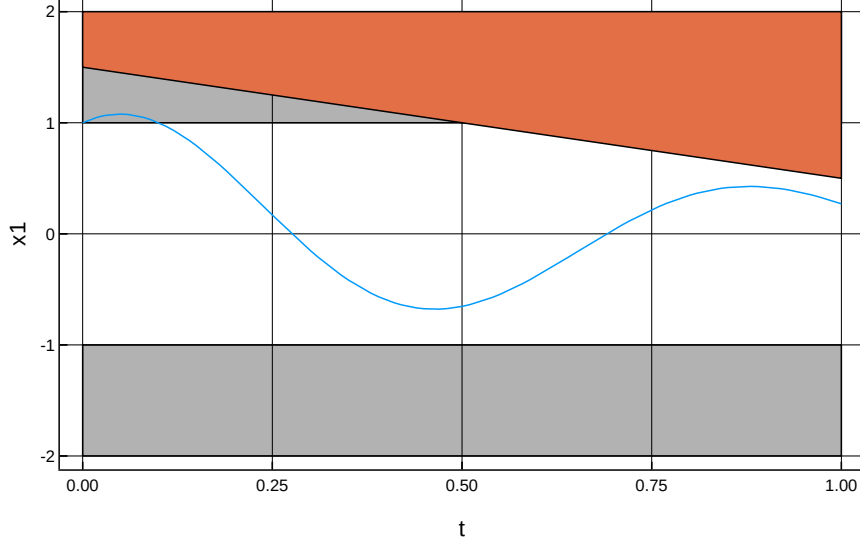


Figure 2: Example inequality constraints from equations 15 and 16

The constraint in equation 15 is violated in the grey area in figure 2, the the equation 16 is violated in the orange region. We observe that a small portion of the state violates the path constraint above $x_1 = 1$, and in practice the optimiser would change the problem variables such that this were not the case (with the physics still being satisfied). Nonlinear constraint functions such as h and b need not merely constrain one variable, and in many cases will be a combination of many, but this would be more difficult to represent graphically. If boundary constraints were enforced, we would then consider the restrictions on the state at time $t = 0$ and $t = 1$.

Finally we will look at the objective function formulation.

2.1.4 Objective function

The objective function to be minimised is divided into a Mayer term and a Lagrange term. The Mayer term is a function of the boundary conditions, while the Lagrange term is a function of the whole state trajectory through time. Together they form the Bolza problem

$$J(t, \mathbf{x}, \mathbf{u}) = \underbrace{\Phi(t_1, t_F, \mathbf{x}(t_1), \mathbf{x}(t_F))}_{\text{Mayer term}} + \underbrace{\int_{t_1}^{t_F} \rho(t, \mathbf{x}(t), \mathbf{u}(t)) dt}_{\text{Lagrange term}} \quad (17)$$

The nonlinear function in equation 17 maps directly to equation 1 in our NLP problem.

This completes our formulation of the dynamic optimisation problem. Now we will discuss the background for a pseudospectral representation of this method.

2.2 The pseudospectral method

Using interpolating polynomials for our \tilde{x} function has the benefit of having derivative information encoded in the polynomial itself (see section 2.3.1). As such, we can use this information, along with a particular distribution of mesh points, to formulate the pseudospectral methods for dynamic optimisation. There are a few different formulations of the pseudospectral method, but to keep things simple we will assume a single polynomial approximates each state and control. More complex formulations include multiple piecewise polynomials approximating each state and control, but this complicates derivative formulation. Additionally, we will be focusing on using Legendre-Gauss-Radau (LGR) points to approximate our polynomial, which are defined on the interval $[-1, 1)$. LGR points are the solution to the polynomial

$$P_N(\tau) + P_{N-1}(\tau) = 0,$$

where P_N are Legendre polynomials.

```
Error: MethodError: no method matching !(::Symbol)
Closest candidates are:
  !(::Matched::Missing) at missing.jl:100
  !(::Matched::Bool) at bool.jl:35
  !(::Matched::Function) at operators.jl:880
  ...
```

Figure 3: Seven LGR Points

2.3 Lagrange polynomials

Lagrange interpolating polynomials in the barycentric form are useful for fast polynomial interpolation and are forward stable for problems with mesh points clustered towards the ends of intervals [?]. Using t_k and t_j to represent our mesh points, and y to represent a function evaluations at points of t such that $y = f(t)$, the unmodified lagrange formula is

$$l_j(t) = \prod_{k=1, k \neq j}^n \frac{t - t_k}{t_j - t_k} \quad (18)$$

$$L(t) = \sum_{j=1}^n y_j l_j(t)$$

We take n to be the number of mesh points (as before in section 2.1), t to be the evaluation point, L is the output of our interpolation at t and l_j are intermediary lagrange weights. The denominator of equation 18 is a function of the mesh points and not of the evaluation point t . As such, we can precompute these beforehand and use them when evaluating our polynomial interpolation. This gives us the barycentric form of our polynomial

$$w_j = \frac{1}{\prod_{k=1, k \neq j}^n (t_j - t_k)} \quad (19)$$

$$L(t) = \frac{\sum_{j=1}^n \frac{w_j}{t - t_j} y_j}{\sum_{j=1}^n \frac{w_j}{t - t_j}} \quad (20)$$

Where w are barycentric weights. Note that the weights are purely a function of the mesh points (the domain of the polynomial) and not of the reference evaluations y at the mesh points. This means we can calculate our polynomial weights without knowledge of the function outputs at mesh points (in this case y), and thus if we want to update our polynomial approximation (providing our mesh spacing is the same) all we need do is update our output values which we use for interpolation. No further computation of the weights is required, and the cost of adding additional weights is much lower than recalculating our approximation.

2.3.1 Differentiation matrix

Having constructed our polynomial in the barycentric form, we can also calculate the differentiation matrix such that

$$\dot{L}(t) = \sum_{j=1}^n \bar{y}_j l_j(t), \quad \bar{y} = Dy,$$

where y is a column vector. The differentiation matrix D is defined as

$$D_{ij} = \begin{cases} \frac{w_j}{w_i(t_i - t_j)}, & i \neq j \\ \sum_{k=0, k \neq i}^n \frac{1}{t_i - t_k}, & i = j \end{cases}$$

2.4 Julia Packages

2.4.1 JuMP.jl

2.4.2 Polynomials.jl

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 4.1, 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 particularly 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 compiler 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 preexisting 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

In order to address the goals in section 3, two packages have been developed which address different but related goals. The first package, Julia dynamic optimisation (JuDO) is a dynamic optimisation toolbox in julia with the aim of supporting multiple solvers (currently only JuMP has been implemented). The second package, MorePolynomials.jl, is an extension of the aforementioned Polynomials.jl package (see section 2.4.2).

We will begin with a discussion of the MorePolynomials.jl package.

4.1 MorePolynomials.jl

MorePolynomials aims to address the need for polynomial representations of state and control vectors in dynmaic optimisation problems, while being generic enough to be used

with other applications. As not to reinvent the wheel (though much wheel reinventing is present in the wild west that is the Julia package library), it is built as an extension to the `Polynomials.jl` package. One advantage of this is by subtyping our polynomials from the `AbstractPolynomial` type, we can make use of many of the generic polynomial functions without having to write additional code. Unfortunately much of this code 'inheritance' (though not to be taken in the traditional manner) relies in this case on having a matching backend data representation, which is not the case. All is not lost however, as we can dispatch on our own custom types and essentially just rewrite the necessary subroutines for our polynomials. Take the example of the `fit()` command from `Polynomials.jl`.

```
function fit(P::Type{<:AbstractPolynomial},
            x::AbstractVector{T},
            y::AbstractVector{T},
            deg::Integer = length(x) - 1;
            weights = nothing,
            var = :x,) where {T}
    x = mapdomain(P, x)
    vand = vander(P, x, deg)
    if weights !== nothing
        coeffs = _wlstsq(vand, y, weights)
    else
        coeffs = pinv(vand) * y
    end
    return P(T.(coeffs), var)
end
```

This function assumes the existence of the `coeffs` data structure, which does not exist in our present formulation. As such, we dispatch on the function with our own implementation. In the case of the `LagrangePoly`, we have

```
function fit(P::Type{<:AbstractLagrangePolynomial}, x::AbstractVector{T},
            y::AbstractVector{T}; var = :x) where {T}
    return LagrangePoly(x,y;var)
end
```

Others can be direct inherited from the package without any code additions. For example in the case of the `roots` function in `Polynomials.jl`

```
function roots(q::AbstractPolynomial{T}; kwargs...) where {T <: Number}
    p = convert(Polynomial{T}, q)
    roots(p; kwargs...)
end
```

Here the `AbstractPolynomial` is converted to type `Polynomial` and then the roots can be found based on the internal `Polynomial` type. As long as we implement the `convert()` function for our polynomial and the `Polynomial` type, we need not dispatch on the `roots()` function. The important not is that to the user it makes no difference thanks to the power of multiple dispatch.

`MorePolynomials.jl` adds the following additional functionality to the `Polynomials.jl` package.

- Fast fitting of low order polynomials using precomputed symbolic expressions
- Lagrange interpolating polynomials in barycentric form
- Lagrange polynomial differentiation and differentiation matrices
- Legendre-Gauss-Radau based Lagrange polynomials with integration
- Combination piecewise polynomials with infinite bound mapping
- Legendre polynomials

We will now take a closer look at the backend implementation of MorePolynomials.jl.

4.1.1 Lagrange Implementation

The Polynomials.jl package has the following data structure which allows polynomial to be stored using it's coefficients.

```
struct Polynomial{T <: Number} <: StandardBasisPolynomial{T}
    coeffs::Vector{T}
    var::Symbol
end
```

As the standard Polynomial type from Polynomials.jl already provides an interface for storing coefficient based polynomials, an additional data structure and supporting subroutines has been introduced to implement the Lagrange polynomial.

```
mutable struct LagrangePoly{T<:Number} <: AbstractLagrangePolynomial{T}
    x::Vector{T}
    y::Vector{T}
    weights::Vector{T}
    domain::Interval{T}
    var::Symbol
end
```

Take note of a couple of features. First, we have chosen to use `x` and `y` to represent our mesh points and function evaluation points respectively, which is inconsistent with the notation in this text. This is by choice to maintain consistency with the Polynomials.jl package, and is mentioned here as many additional decisions of notation in this package have been chosen for similar reasons. Second, the types of all fields (excluding `var`) are parameterised and forced to be the same. The benefits of this are twofold: we don't have to call `convert` functions when executing any computation that is a function of more than one field, and the compiler is fully aware of the type at runtime, and thus can optimise operations for this type, leading to significant performance gains. This choice of representation is made for the benefit gain in polynomial creation and alteration, at the expense of larger memory presence and evaluation time.

To illustrate the differences between the two representations, let's benchmark some polynomials. First, let's start with polynomial fitting

```

x = [-1,-0.6,-0.2,0.2,0.6,1]
y = map( (x)->x^5 + 3x^2 - x + 1, x)
# y points for a 5th order polynomial x^5 + 3x^2 - x + 1

mean(@benchmark fit(Polynomial, x, y)) # using Polynomials.jl

BenchmarkTools.TrialEstimate:
  time:           16.474 μs
  gctime:         592.633 ns (3.60%)
  memory:         8.44 KiB
  allocs:         40

mean(@benchmark fit(LagrangePoly, x, y)) # using MorePolynomials.jl

BenchmarkTools.TrialEstimate:
  time:           1.829 μs
  gctime:         72.512 ns (3.96%)
  memory:         768 bytes
  allocs:         29

```

This is perhaps an unfair comparison as Polynomials.jl has the additional step of converting to a coefficient representation. A more fair comparison would be from time to polynomial creation to first evaluation. Let's time the creation and evaluation of the polynomial at point $x=0.25$.

```

mean(@benchmark fit(Polynomial, x, y)(0.25)) # using Polynomials.jl

BenchmarkTools.TrialEstimate:
  time:           17.078 μs
  gctime:         708.019 ns (4.15%)
  memory:         8.45 KiB
  allocs:         41

mean(@benchmark fit(LagrangePoly, x, y)(0.25)) # using MorePolynomials.jl

BenchmarkTools.TrialEstimate:
  time:           1.879 μs
  gctime:         74.529 ns (3.97%)
  memory:         784 bytes
  allocs:         30

```

We observe a minor time increase in both cases but MorePolynomials.jl is still almost an order of magnitude faster. This is largely due to the barycentric representation, which uses barycentric weights to speed up the evaluation of the polynomial, allowing an evaluation time in the order of $\mathcal{O}(n)$ flops. This incurs overhead when generating the weights for the polynomial, but this is still significantly faster than the least-squares approximation employed by Polynomials.jl. Internally, the generation of weights takes the following form

```

function lagrange_bary_weights(x::AbstractVector{T}) where {T}
    numPoints = length(x)
    w = ones(T, numPoints)
    for j in 2:numPoints
        xj = x[j]
        for k in 1:j-1
            w[k] *= x[k] - xj
        end
        for k in j+1:numPoints
            w[k] *= xj - x[k]
        end
    end
    return 1 ./ w
end

```

This achieves the same result as equation 19. The exact number of flops required to calculate the weights is $n \times (n - 1)$. We execute the same for loop twice so to take advantage of the cpu cache when indexing the final weight.

As mentioned in section 2.3, the barycentric representation allows us to update our representation without having to recumpute weights if our mesh points are the same. This can be particularly useful for dynamic optimisation problems, where the states are perpetually changing but the mesh remains the same between solves. We demonstrate this below

```

lPoly = fit(LagrangePoly, x, y)
ynew = map( (x)->5x^5 + 7x^4 - 3x - 1, x)
updateAndEval(lPoly, ynew, x) = lPoly[:] = ynew; lPoly(x)
mean(@benchmark updateAndEval(lPoly, ynew, 0.25))

```

```

BenchmarkTools.TrialEstimate:
  time:           139.725 ns
  gctime:         28.488 ns (20.39%)
  memory:         224 bytes
  allocs:         5

```

An extension of the Lagrange polynomial is the Legendre-Gauss-Radau Lagrange polynomial¹. This uses LGR points to generate the mesh. See this paper on the benefits of using LGR points of function approximation in dynamic optimisation [?]. The current implementation finds the LGR points as a function of the eigenvalues of a symmetric Legendre polynomial based matrix, but future implementations will most likely use a combination of this method and the implementation in the package FastGaussQuadrature, dynamically switching based on speed of implementation and number of points. For the moment, our implementation is faster at low orders. Additionally, if the mesh points follow the LGR formulation, we can make use of the LGR weights generated from FastGaussQuadrature for fast integration. Let's now compare some polynomial integration between Polynomials.jl and MorePolynomials.jl.

¹Try saying that really quickly many times in a row!

```

x = lgr_points(6)

mean(@benchmark integrate(fit(Polynomial, x, y)),0) # using Polynomials.jl

BenchmarkTools.TrialEstimate:
  time:           16.524 μs
  gctime:         0.000 ns (0.00%)
  memory:         8.91 KiB
  allocs:         46

mean(@benchmark integrate(fit(LGRPoly, y))) # using MorePolynomials.jl

BenchmarkTools.TrialEstimate:
  time:           6.235 μs
  gctime:         513.240 ns (8.23%)
  memory:         3.23 KiB
  allocs:         59

```

And just for fun let's compare this to a similar implementation in matlab (output is in seconds)

```

open(`matlab -nodesktop -nosplash`, "w", stdout) do io
  println(io,"x = $x;")
  println(io,"y = $y;")
  println(io,"int = @() polyint(polyfit(x,y,5));")
  println(io,"time_taken = timeit(int)")
end

MATLAB is selecting SOFTWARE OpenGL rendering.

               < M A T L A B (R) >
      Copyright 1984-2019 The MathWorks, Inc.
R2019a Update 3 (9.6.0.1135713) 64-bit (glnxa64)
                June 5, 2019

To get started, type doc.
For product information, visit www.mathworks.com.

>> >> >>
time_taken =

    0.0010

```

4.1.2 Low order Vendemonde methods

4.1.3 Piecewise Polynomials

4.1.4 User interface

4.1.5 Error handling

4.1.6 Benchmarking and performance gains

4.2 JuDO.jl

The development of the JuDO.jl package has been somewhat more staggered than the MorePolynomials.jl package. While MorePolynomials.jl inherits much of the design ethos from Polynomials.jl, JuDO.jl has no such counterpart. Additionally, one of the key learning points from ICLOCS was that a good base design facilitates easy and intuitive feature addition, and avoids problems associated with feature creep. As such, progress with JuDO.jl has been slow but deliberate. In this text we will outline a basic generic implementation of transcription methods in JuMP,

5 Documentation

6 Next steps

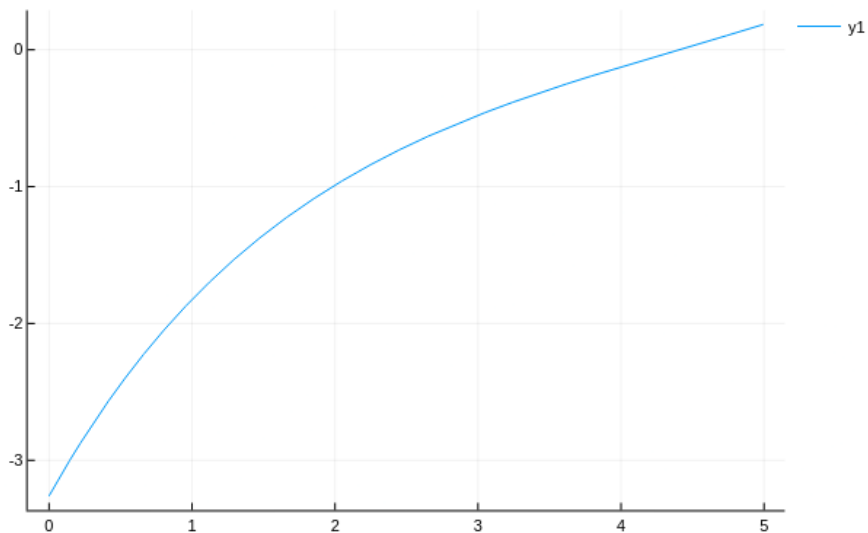


Figure 4: a

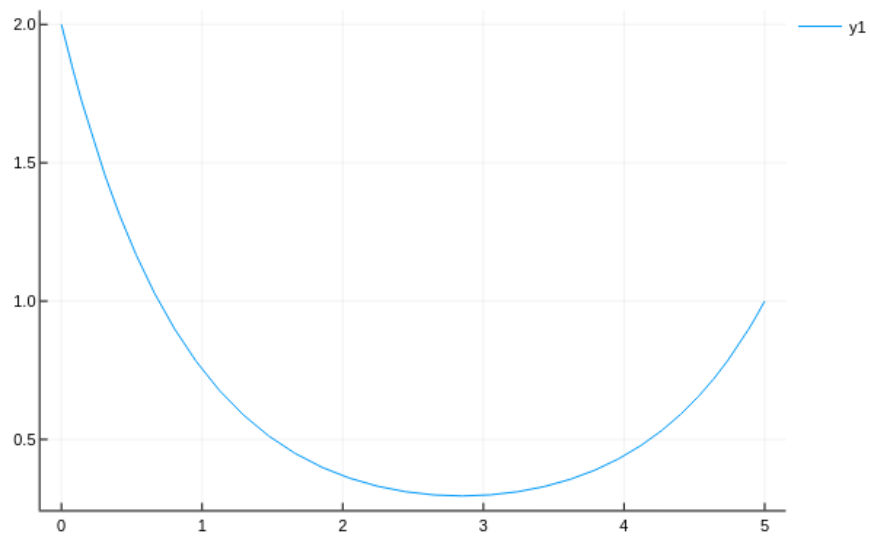


Figure 5: a

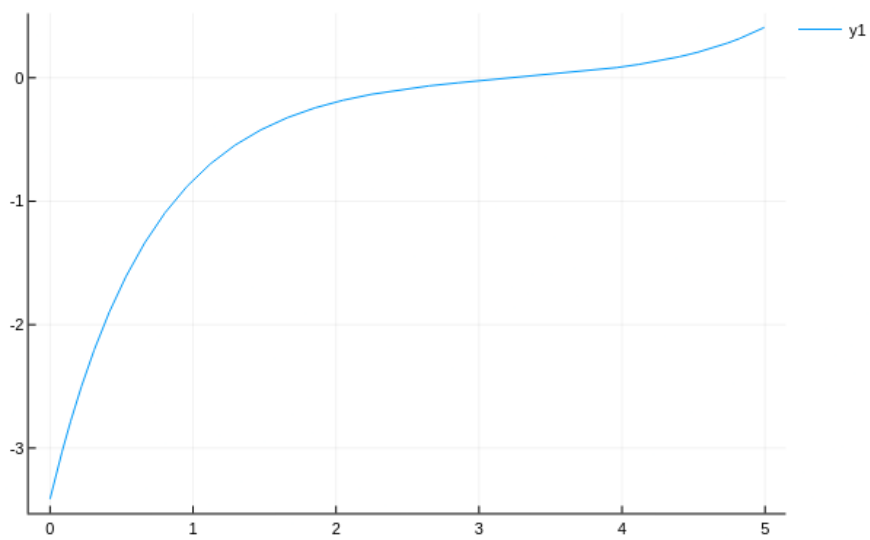


Figure 6: a

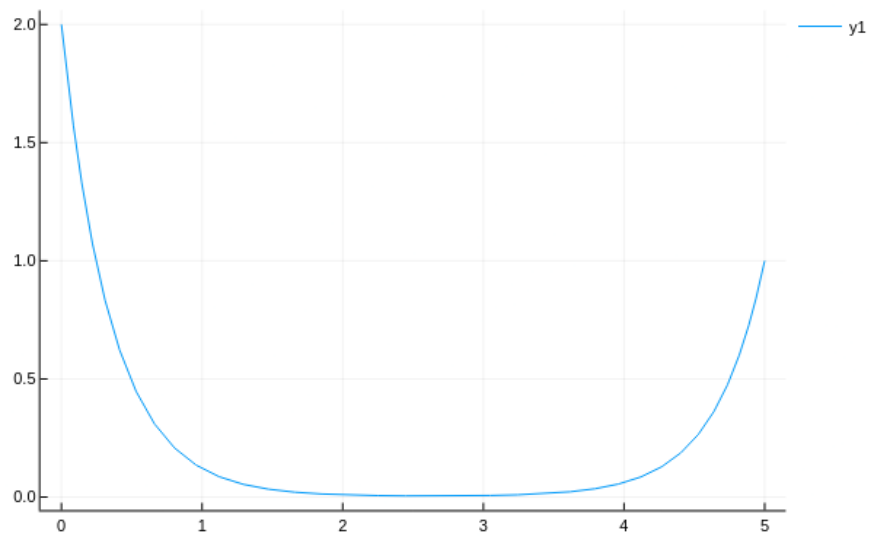


Figure 7: a