Dynamic Optimisation using the Julia Programming Language

Nathan Davey

June 2, 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)

ICLOCS is the MATLAB predecessor to the package we are attempting to lay the foundations for here. It implements a number of collocation, mesh refinement and differentiation methods, and provides interfaces to a number solvers. Three areas for improvement stand out with the current implementation of ICLOCS; the user interface, the ability to maintain an ever growing codebase, and the internal representation of states and controls. With regards to the user interface, the user is often required to construct the problem over a number of files, and the user must often formulate the problem in a particular manner to be compatible with the ICLOCS functions. With regards to the state of the codebase, ICLOCS suffers from feature creep, and the ability to implement new features often requires substantial wrangling with the code. With regards to the internal representation of states and controls, from the outset it was assumed that the dimensions of the state and control vectors would always be consistent. This assumes a fixed number of collocation points for all states and controls for a given problem. One of the goals of this text is to outline the foundations for a method which allows for a varying number of collocation points for each state and control using interpolating functions.

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. By types we refer to the data type of variables such as `Int`, `Float64` etc., but julia also provides the ability to define types which contain other types (like structs in C), and abstract types from which other types can be subtyped. We can execute different subroutines depending on the combination of the types of the arguments of a function, and this ability is referred to as multiple dispatch. For example, the function `f(x::Int, y::Float64)` is a different function to `f(x::Int,y::Char)` or even `f(x::Float64, y::Int)`, but all three use the same `f` identifier i.e. we overload `f` for each type combination.

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)

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

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

$$\boldsymbol{x}_L \leq \boldsymbol{x} \leq \boldsymbol{x}_U \tag{3}$$

Where

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

is the objective function to be minimised,

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

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

$$\boldsymbol{g}(\boldsymbol{x}) \leq 0 \tag{4}$$

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

$$\boldsymbol{g}(\boldsymbol{x}) = 0 \tag{5}$$

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 dynmaics, 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

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

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 discritse 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

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

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 a vector of interpolating data points. As illustrated in section 2.3, function $\tilde{x}$ can be initialised without knowledge of the specifics of t. The number of decision variables in 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 $\bar{x}_{ij}$ will consist of a mesh location and state value as below.

$$x_i = \Big[\bar{x}_{ij} = [t_{ij}, x_{ij}]\Big], \quad x(t_{ij}) = x_{ij}, \quad j = 1, 2, \dots m_i. \tag{8}$$

Crucially x_{ij} are our decision variables. These discretised states will go onto 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 $\boldsymbol{x} \in \mathbb{R}^{n \times m}$. By the introduction of and 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 $\boldsymbol{x}$ represents a vector of state approximating functions such that $\boldsymbol{x}_i(t_k) = \tilde{x}_i(x_i, t_k)$, 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.

$$\boldsymbol{u} = [\tilde{u}_i(t, u_i)], \quad u_i \in \mathbb{R}^{m_i}, \quad \tilde{u} : \mathbb{R}^{m_i+1} \to \mathbb{R}, \quad i = 1, 2 \dots c, \tag{9}$$

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 $\boldsymbol{\tau} = [\tau_i] = [t_1, t_2 \dots t_{m_i}], i = 1, 2 \dots n$

$$\boldsymbol{f} = f_i(\tau_i, \boldsymbol{x}(t), \boldsymbol{u}(t)) : \mathbb{R}^{(n+c) \times m_i} \to \mathbb{R}^{m_i}, \quad i = 1, 2 \dots n \tag{10}$$

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 τ_i 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 $\tau_i = t_k, k = 1, 2 \dots m_i$.

Putting this all together we have

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

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

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

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

$$\boldsymbol{f}\begin{cases}\dot{x}_{,1} = & f_1(\tau_1, \boldsymbol{x}(t), \boldsymbol{u}(t)) \\ \dots & \\ \dot{x}_{,n} = & f_n(\tau_n, \boldsymbol{x}(t), \boldsymbol{u}(t))\end{cases}$$

$$n = \text{number of states}, \tau_i = \text{state collocation points}, i = 1, 2 \dots n$$

We use $\dot{x}_{,i}$ 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 $f_1(\tau_1, \boldsymbol{x}(t), \boldsymbol{u}(t))$, τ_1 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. Additionally, inside the function f_i, $\boldsymbol{x}$ and $\boldsymbol{u}$ may actually be evaluated at any time as they are interpolating functions, not necessarily at just the collocation points.

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 11. Note here h_k is the timestep $h_k = t_{k+1} - t_k$ and t_k is the collocation point being evaluated.

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

$$\boldsymbol{x}_i(t_{k+1}) - \boldsymbol{x}_i(t_k) - h_k \boldsymbol{f}_i(t_k) = 0 \tag{11}$$

For the sake of consistency, $\boldsymbol{x}_i(t) = \tilde{x}_i(t, x_i)$, $\boldsymbol{f}_i(\tau) = f_i(\tau, \boldsymbol{x}(t), \boldsymbol{u}(t))$. 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

$$\boldsymbol{x}_i(t_{k+1}) - \boldsymbol{x}_i(t_k) - \frac{h_k}{2}\Big(\boldsymbol{f}_i(t_k) + \boldsymbol{f}_i(t_{k+1})\Big) = 0 \tag{12}$$

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, hense trapizoidal method).

Equations 11 and 12 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 11. The equations of interest are

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

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

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
    ω_n = sqrt(k/m)
    ζ = c / (2*m*ω_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 * ζ * ω_n * x2 - ω_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()
```

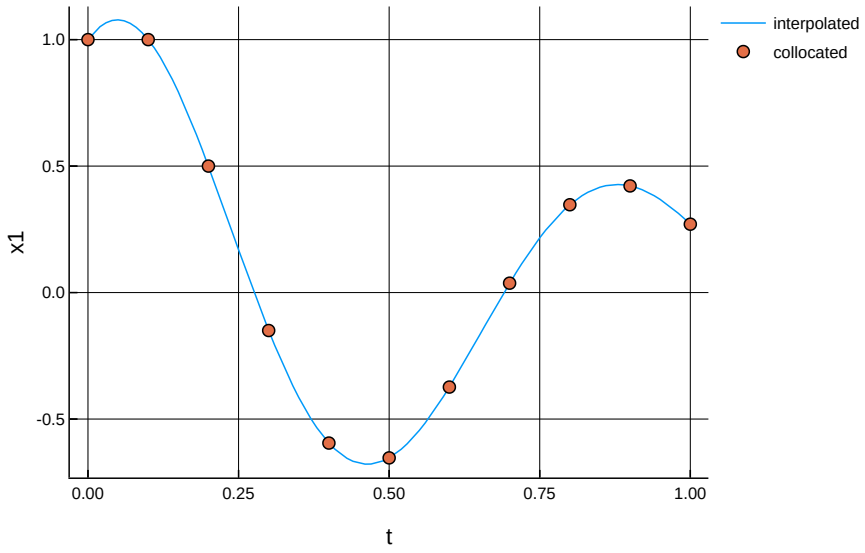

Figure 1: Collocated system dynamics for a mass spring damper

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

$$t_{1,L} \leq t_1 \leq t_{1,U}$$
$$t_{F,L} \leq t_F \leq t_{F,U}$$
$$\boldsymbol{x}_{i,L}(t_1) \leq \boldsymbol{x}_i(t_1) \leq \boldsymbol{x}_{i,U}(t_1)$$
$$\boldsymbol{x}_{i,L}(t_F) \leq \boldsymbol{x}_i(t_F) \leq \boldsymbol{x}_{i,U}(t_F)$$

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

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

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

$$\boldsymbol{x}_{i,L} \leq \boldsymbol{x}_i(\tau_i) \leq \boldsymbol{x}_{i,U} \tag{13}$$

$$\boldsymbol{u}_{i,L} \leq \boldsymbol{u}_i(\tau_i) \leq \boldsymbol{u}_{i,U} \tag{14}$$

$$\boldsymbol{h}(\boldsymbol{\tau}, \boldsymbol{x}(t), \boldsymbol{u}(t)) \leq 0 \tag{15}$$

For the sake of completeness equations 13 and 14 correspond to equation 3 and equation 15 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.

$$-1 \leq \boldsymbol{x}_1(t_k) \leq 1 \tag{16}$$

$$h_1(t_k, \boldsymbol{x}(t_k), \boldsymbol{u}(t_k)) = \boldsymbol{x}_1(t_k) + t_k - 1.5 \leq 0 \tag{17}$$

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.

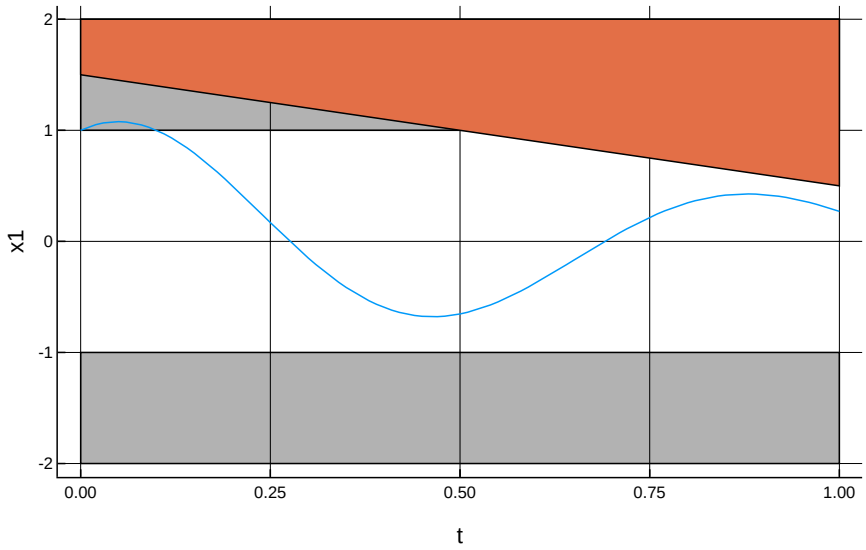

Figure 2: Example inequality constraints from equations 16 and 17

The constraint in equation 16 is violated in the grey area in figure 2, the the equation 17 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

$$J(t, \boldsymbol{x}, \boldsymbol{u}) = \underbrace{\Phi(t_1, t_F, \boldsymbol{x}(t_1), \boldsymbol{x}(t_F))}_{\text{Mayer term}} + \underbrace{\int_{t_1}^{t_F} \rho(\tau_\rho, \boldsymbol{x}(t), \boldsymbol{u}(t))dt}_{\text{Lagrange term}} \tag{18}$$

The nonlinear function in equation 18 maps directy to equation 1 in our NLP problem. With regards to the Lagrange term, there are a few options for calculating the integral. If no interpolating function is used, we could a basic approximation

$$\int_{t_1}^{t_F} \rho(\tau_\rho, \boldsymbol{x}(t), \boldsymbol{u}(t))dt \approx \sum_{k=1}^{m-1} \frac{1}{2}(h_k)\rho(\tau_{\rho k}, \boldsymbol{x}(\tau_{\rho k}), \boldsymbol{u}(\tau_{\rho k}))$$

Alternatively, if the objective function is modelled with an interplating function we can calculate the exact intergral of the interplation as per equation 24.

This completes our formulation of the dynamic optimisation problem.

2.2 Example implementation in ICLOCS

2.3 Lagrange polynomials

Now we will discuss the background for the interpolating functions used to approximate state and control functions (and potentially the objective function), which will lead us to a discussion of the pseudospectral dynamic optimisation method. 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

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

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

We take n to be the number of mesh points (as before in seciton 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 denominaor of equation 19 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 out polynomial

$$w_j = \frac{1}{\prod_{k=1,k\neq j}^{n}(t_j - t_k)} \tag{20}$$

$$L(t) = \frac{\sum_{j=1}^{n} \frac{w_j}{t - t_j} y_j}{\sum_{j=1}^{n} \frac{w_j}{t - t_j}} \tag{21}$$

Where w are 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

$$\dot{L}(t) = \sum_{j=1}^{n} \bar{y}_j l_j(t), \quad \bar{y} = Dy, \tag{22}$$

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

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

2.4 The pseudospectral method

The pseudospectral method is of particular interest as its formulation allows us to exploit the use of the interpolating polynomial describle in section \ref{dynamicsmath} Using interpolating polynomials for our $\tilde{x}$ function has the benefit of having derivative information encoded in the polynomial itself (see seciton 2.3.1). As such, we can use this information, along with a particular distribution of mesh points, to forumulate 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

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

where P_N are Legendre polynomials.

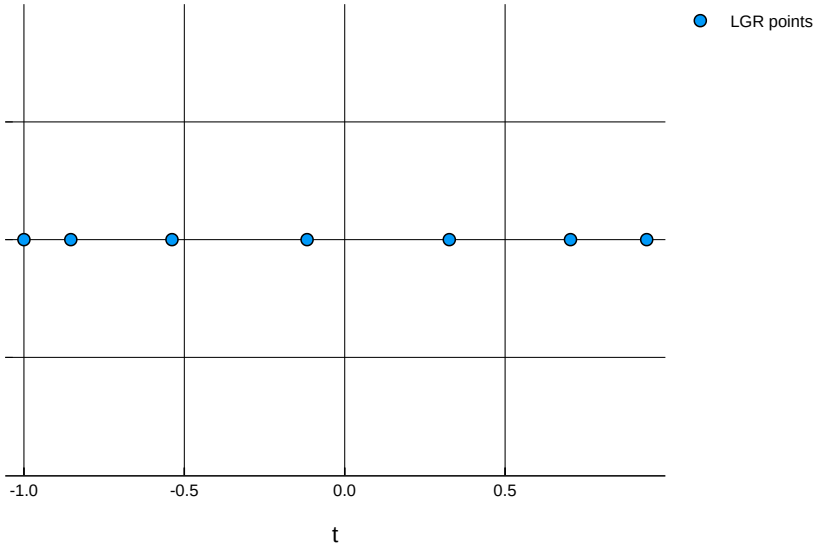

Figure 3: Seven LGR Points

We use the LGR points as a base for the interpolating polynomial mesh, such that t_{ij} in equation 8 are the LGR points. As the LGR points are defined on the interval $[-1, 1)$, we can use the following transformation to transform from $[-1, 1]$ to $[-t_1, t_F]$.

$$t = \frac{t_1 - t_F}{2} t_{ij} + \frac{t_1 + t_F}{2}. \tag{23}$$

If we know our interpolating points correspond with LGR points, we can also use the LGR weights w_{LGR} to calculate the exact integral.

$$\int_{-1}^{1} x(t)dt = \sum_{k}^{m} w_{LGR,k} x(t_k) \tag{24}$$

The is useful for determining the object function, or for approximating endpoints as below.

Now we can use the derivate information of our polynomial to enforce our system dynamics

$$\dot{\tilde{x}}_i(\tau_i, x_i) - f_i(\tau_i, \boldsymbol{x}(t), \boldsymbol{u}(t)) = 0, \quad i = 1, 2 \dots n, \tag{25}$$

where $t = t_1, t_2, \dots t_{m_i}$ is a vector of collocation points for this state and $\dot{\tilde{x}}$ is aquired from equation 22. As in this formulation our LGR points do not include the endpoint, we must construct an interpolating constraint for this too.

$$\boldsymbol{x}_i(t_F) = \boldsymbol{x}_i(t_1) + \int_{-1}^{1} \dot{x}_{,i}(t)dt \approx \boldsymbol{x}_i(t_1) + w_{LGR} \boldsymbol{f}_i(\tau_i)$$

Here $t_1 = -1, t_F = 1$. Use equation 23 to transform to the time domain of the optimisation problem. Additionally, w_{LGR} is a row vector, and $\boldsymbol{f}_i$ is a column vector, such that $w_{LGR}\boldsymbol{f}_i \rightarrow$

$\mathbb{R}^{m_i}$. Note we can avoid this additional constraint by using a set of flipped LGR points. It is useful to know however as some pseudospectral methods are defined on the interval (-1,1).

Finally, the full pseudospectral LGR formulation of the dynamic optimisation problem is

$$
\begin{aligned}
\min \quad & J(\tau_\rho, \boldsymbol{x}(t), \boldsymbol{u}(t)) = \Phi(t_1, t_F, \boldsymbol{x}(t_1), \boldsymbol{x}(t_F)) + w_{LGR}\tilde{\rho}(\tau_\rho, \boldsymbol{x}(t), \boldsymbol{u}(t)) \\
\text{s.t.} \quad & \dot{\boldsymbol{x}}(\boldsymbol{\tau}) - \boldsymbol{f}(\boldsymbol{\tau}, \boldsymbol{x}(t), \boldsymbol{u}(t)) = 0 \\
& \boldsymbol{x}(t_F) - \boldsymbol{x}(t_1) - w_{LGR}\boldsymbol{f}(\boldsymbol{\tau}) = 0 \\
& t_{1,L} \leq t_1 \leq t_{1,U} \\
& t_{F,L} \leq t_F \leq t_{F,U} \\
& \boldsymbol{x}_{,L}(t_1) \leq \boldsymbol{x}(t_1) \leq \boldsymbol{x}_{,U}(t_1) \\
& \boldsymbol{x}_{,L}(t_F) \leq \boldsymbol{x}(t_F) \leq \boldsymbol{x}_{,U}(t_F) \\
& \boldsymbol{b}(t_1, t_F, \boldsymbol{x}(t_1), \boldsymbol{x}(t_F)) \leq 0 \\
& \boldsymbol{x}_{,L} \leq \boldsymbol{x}(\boldsymbol{\tau}) \leq \boldsymbol{x}_{,U} \\
& \boldsymbol{u}_{,L} \leq \boldsymbol{u}(\boldsymbol{\tau}) \leq \boldsymbol{u}_{,U} \\
& \boldsymbol{h}(\boldsymbol{\tau}, \boldsymbol{x}(t), \boldsymbol{u}(t)) \leq 0
\end{aligned}
$$

2.5 Julia Packages

There are 2 packages which are referred to at length in this text which may be worth introducing. These are the Polynomials.jl and JuMP.jl packages.

2.5.1 Polynomials.jl

Polynomials.jl is a package by JuliaMath, and adds the ability to create, differentiate and integrate coefficient based polynomials. The interface is simple but intuitive, which is one of the key reasons for choosing it as a package to extend, the other being that it is well maintained. The goal of this project is to extend Polynomials.jl to cover a verity of Lagrange interpolating polynomials which can be used in dynamic optimisation, as well as add a few new features along the way. To summarise, Polynomials.jl is good, but we can make it better.

2.5.2 JuMP.jl

JuMP.jl is a package by Julia Computing which aims to provide a standard interface to a number of linear, quadratic and nonlinear programming solvers. Like many julia packages in their infancy, JuMP.jl is under heavy development and the interface is somewhat restrictive in its current implementation. Nonetheless, it abstracts well much of the complexities associated with MathOptInterface.jl, the package which acts as a wrapper for a number of solvers. To this end, JuMP is an NLP solver wrapper wrapper, and they're here to lay down some fire beats.

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 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

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 afformentioned Polynomials.jl package (see section 2.5.2).

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 funtions 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 mathcing 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 and 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 inconsistant 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. Seconds, 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 execution 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. We'll use 6 points for a 5th order polynomial, which will give us an exact representation. Note the fact that we have an exact representation should not affect the benchmarking results as the same number of operations are taking places. First, let's start with polynomial fitting

```
x = [-1,-0.6,-0.2,0.2,0.6,1]
y = @. x^5 + 3x^2 - x + 1
# 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:             15.596 μs
  gctime:           0.000 ns (0.00%)
  memory:           8.44 KiB
  allocs:           40

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

BenchmarkTools.TrialEstimate:
  time:             1.662 μs
  gctime:           0.000 ns (0.00%)
  memory:           768 bytes
  allocs:           29
```

This is perhapse 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 creatation and evaluation of the polynomial at point `x=0.25`.

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

BenchmarkTools.TrialEstimate:
  time:             15.981 μs
  gctime:           0.000 ns (0.00%)
  memory:           8.45 KiB
  allocs:           41

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

BenchmarkTools.TrialEstimate:
  time:             1.816 μs
  gctime:           0.000 ns (0.00%)
  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 1:j-1
            w[j] *= xj - x[k]
        end
    end
    return 1 ./ w
end
```

This achieves the same result as equation 20. 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 interpolation points (y_j) without having to recompute 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 using `lPoly[:] = ynew`

```
lPoly = fit(LagrangePoly, x, y)
ynew = @. 5x^5 + 7x^4 - 3x - 1
# update our y values in lPloly and evaluate at x
updateAndEval(lPoly, ynew, x) = lPoly[:] = ynew; lPoly(x)
mean(@benchmark updateAndEval(lPoly, ynew, 0.25))

BenchmarkTools.TrialEstimate:
  time:             146.259 ns
  gctime:           33.332 ns (22.79%)
  memory:           224 bytes
  allocs:           5
```

Differentiation as per equation 22 has also been implemented in the barycentric form, and works identically to the derivative function in Polynomials.jl. When we differentiate a polynomial, we receive a new polynomial representing the differential of the input polynomial.

```
lPolyDer = derivative(lPoly)
```

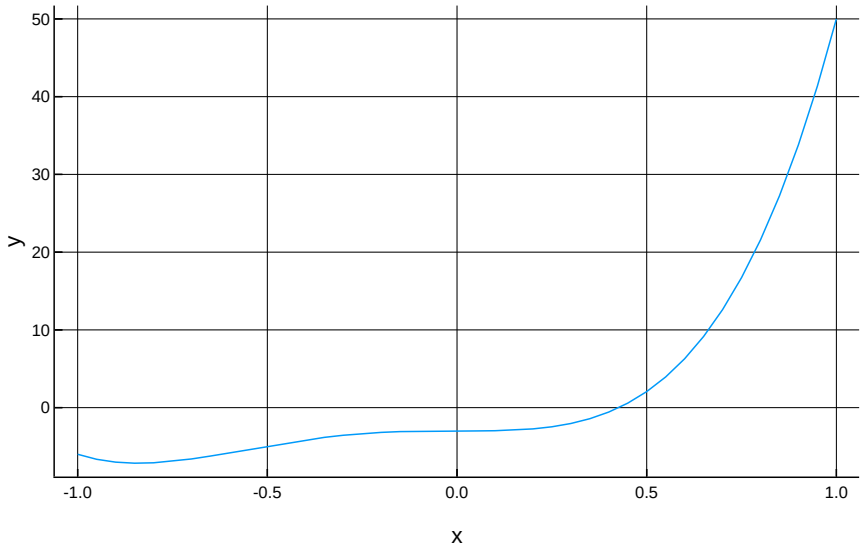

Figure 4: lPolyDer plot i.e. Proof that I wrote code that actually works and it's not all smoke and mirrors.

Once again due to the barycentric implementation the fitting to derivative time exceeds that of the Polynomials.jl package.

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

BenchmarkTools.TrialEstimate:
  time:             16.165 μs
  gctime:           0.000 ns (0.00%)
  memory:           8.86 KiB
  allocs:           46

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

BenchmarkTools.TrialEstimate:
  time:             3.722 μs
  gctime:           176.209 ns (4.73%)
  memory:           2.08 KiB
  allocs:           39
```

What's more, if we intend to recalculate derivatives using the same mesh spacing, MorePolynomials provides a function for the derivative matrix, which when stored saves us having to recalculate it every time we update our function interpolation points. We demonstrate this below using `lPolyDer[:] = derivmatrix*ynew`

```
D = derivmatrix(lPoly)
ynew = @. 7x^5 + 9x^4 - x - 2
# update our y values and recalculate derivative
updateDerivAndEval(lPolyDer, ynew, D, x) = lPolyDer[:] = D*ynew; lPolyDer(x)
mean(@benchmark updateDerivAndEval(lPoly, ynew, D, 0.25))

BenchmarkTools.TrialEstimate:
  time:             284.996 ns
  gctime:           53.331 ns (18.71%)
  memory:           352 bytes
  allocs:           6
```

An extension of the Lagrange polynomial is the Legendre-Gauss-Radau Lagrange polynomial [1]. 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.

[1] Try 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:             16.147 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:             6.607 μs
  gctime:           594.072 ns (8.99%)
  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

< 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 =

    0.0011
```

4.1.2 Low order Vandermonde methods

One method for analytically determining coefficients from data points is by inversion of the Vandermonde matrix.

BENCHMARK

4.1.3 Piecewise polynomials

4.1.4 User interface

The challenge of any user interface is to balance the exposure provided to the user of complex functionality with the ability to remain intuitive and simple to use. This is more apparent when designing a graphical user interface, but elements of this are apparent in any interface. By far, one of the key elements to maintaining an intuitive interface is consistency. As previously stated one of the objectives of the MorePolynomials.jl package is to maintain close compatibility with Polynomials.jl. One aspect of maintaining this compatibility it being consistent with maintaining the intent of user functions, and Julia's multiple dispatch system is well suited for this role.

Take the `fit()` command for example. In Polynomials.jl, if we want to fit a polynomial to a set of datapoints, we have two arguments, the `x` values and the `y` values

```
fittedpoly = fit(x, y)
```

In MorePolynomials.jl, we extend this function to cover our custom polynomials using the method provided by Polynomials.jl

```
fittedpoly = fit(LagrangePoly, x, y)
```

Crucially however, we also make use of multiple dispatch to dispatch on a function arguments, allowing the user to define a function to fit, rather than merely data points.

```
fittedpoly = fit( (x) -> x^2 - 5, 7)
```

As no data point spacing has been specified (only the number of points), we can also guess that the user might wish to use an LGR spacing, as this provides a good approximation when using Lagrange polynomials. This is not consistent with the arguments that Polynomials.jl fit command takes, but it is consistent with the intent of the user. The user knows what it means to fit a polynomial to a function, and so this formulation is intuitive from a human perspective. We employ similar patterns in our speech everyday. One can say "I am going to work", or "I am going to Spain" or "I am going to walk the dog". All these actions are conducted in different manners, but the intent of the statement is the same, that one will travel from one place to another through some means. Here, we take $fit()$ to mean fit some input to a polynomial representation, the internals are dependent on the context. As the core role of the programming language is to provide an interface between the human and the computer is makes sense (when we have the ability) to tailor the experience in a way similar to how a human would behave.

4.1.5 Error handling

4.2 JuDO.jl

The development of the JuDO.jl package has been somewhat more staggered than the MorePolynomials.jl package. While MorePolynomials.jl inherists 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, and a proposed architecture to be taken forward based on the project aims.

4.2.1 Generic transcription implementation

A transcription method which employs the use of user defined functions has been implemented in julia using the JuDO package with the Ipopt NLP solver. The key innovation here is the formatting of the problem such that the user can define any generic functions for the objective function, constraints and dynamics and, providing the functions accept a standard input and output, the solver behave as expected.

The user defined problem, which also acts as an internal interface between JuMP and JuDO methods, is defined as follows.

```
mutable struct TrajProblem
    objectiveFunc::Function
    dynamicsFunc::Function
    pathConstraintFunc::Function
    controlVector::Array
    stateVector::Array # each row represents a state, maybe use an add state
guess function which adds each state guess
    timeStep::Union{Array, Float64}
    boundaryConstraints::BoundaryConstraint
    numStates::Int
    numControls::Int
    numCollocationPoints::Int
end
```

JuMP provides its own interface for defining functions with derivatives, the trick is formatting the interface in such a way as to work with dynamic optimisation problems. This is accomplished using three components, a global `CurrentProblem` type, a translating layer and a derivative function.

First, the global `CurrentProblem` type allows us to access the full problem definition from any location within our program. This is useful as it provides a standard interface which we can access within functions, which we can used to access crucial data such as they problem functions and state and control interpolation functions. Moreover, the user defined functions in JuMP only allow the parsing of JuMP variables internally, so to have access to any additional data defined externally one must define such an interfaces as this. Within the JuDO code this `CurrentProblem` type is defined as a global constant, which acts like a pointer to the location of the user defined `TrajProblem`. The interface is defined below.

```
mutable struct CurrentProblem
    nullableProblem::Union{TrajProblem,Nothing}
end
const CURRENT_PROBLEM = CurrentProblem(nothing)
setCurrentProblem(problem::TrajProblem) = (CURRENT_PROBLEM.nullableProblem =
problem)
getCurrentProblem() = CURRENT_PROBLEM.nullableProblem
```

Now the translational layer provides a set of functions for translating between the JuMP user defined functions, and our own custom functions and collocation methods. Their implementation primarily consists of translating 1D decision variable vectors from JuMP into 2D state and control vectors, and then back to 1D vectors for JuMP. Additionally, JuMP do not provide a robust method for determining what decision variable we're focusing on within a user defined function, so additional interfaces for state and control indexing are provided by this layer.

Finally, for each user defined function we provide for JuMP, we must additionally provide the derivative information. This is done using the ForwardDiff.jl package which provides forward mode automatic differentiation for Julia functions, which can be used to calculate the Jacobian and Hessian arrays. ForwardDiff.jl achieves this by injecting the function with a custom type which has union with the `Real` type, which has the limitation of requiring all internal operations to be compatible with the real type. If this is not the case however a forward difference method may be implemented in its place. The differentiation functions work by calculating the fully Jacobian and then returning the row associated with the state or control in question. For example, in the case of the cart-pole example in section **??** below, we can calculate the Jacobian using command.

```
jacobian = ForwardDiff.jacobian(collocateConstraint, [stateVector; controlVector])

116×150 Array{Float64,2}:
 -1.0   0.0       -0.0344827  …   0.0         0.0         0.0
  0.0  -1.0        0.0            0.0         0.0         0.0
  0.0  -0.101483  -1.0            0.0         0.0         0.0
  0.0   0.879517   0.0            0.0         0.0         0.0
  0.0   0.0        0.0            0.0         0.0         0.0
  0.0   0.0        0.0        …   0.0         0.0         0.0
  0.0   0.0        0.0            0.0         0.0         0.0
  0.0   0.0        0.0            0.0         0.0         0.0
  0.0   0.0        0.0            0.0         0.0         0.0
  0.0   0.0        0.0            0.0         0.0         0.0
  ⋮                             ⋱
  0.0   0.0        0.0            0.0         0.0         0.0
  0.0   0.0        0.0            0.0         0.0         0.0
  0.0   0.0        0.0            0.0         0.0         0.0
  0.0   0.0        0.0        …   0.0         0.0         0.0
  0.0   0.0        0.0            0.0         0.0         0.0
  0.0   0.0        0.0           -0.0344827   0.0         0.0
  0.0   0.0        0.0            0.0        -0.0344827   0.0
  0.0   0.0        0.0            1.0         0.0        -0.0344827
  0.0   0.0        0.0        …   0.0         1.0        -0.0689655
```

Take note of the sparsity pattern here. Julia has methods for handling sparse matrices, and one of the future goals of the project is to support sparse matrix represents for derivatives. If we are looking for the derivatives of the first state (or more correctly in this case the first collocation function) relative to perturbations in the other states, we can take the first row of this Jacobian.

```
df1dt = jacobian[1,:]'

1×150 LinearAlgebra.Adjoint{Float64,Array{Float64,1}}:
 -1.0  0.0  -0.0344827  0.0  0.0  1.0  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

Now that we have the core components to interface with JuMP, the rest is a matter of formulating the JuMP NLP problem and solving. Links to the full implementation can be found in the appendix.

4.2.2 Example cart-pole swing-up

An example implementation is the cart-pole swing-up problem, where the objective is to swing a mass above a cart using the minimum effort squared. The example is taken from Matthew Kelly's paper [**?**] and has the following formulation

$$\boldsymbol{x}(t) = \begin{bmatrix} q \\ \theta \\ \dot{q} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}_1(t) \\ \boldsymbol{x}_2(t) \\ \boldsymbol{x}_3(t) \\ \boldsymbol{x}_4(t) \end{bmatrix},$$

$$\boldsymbol{f}(\tau, \boldsymbol{x}, \boldsymbol{u}) = \begin{bmatrix} \dot{q} \\ \dot{\theta} \\ \ddot{q} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}_3 \\ \boldsymbol{x}_4 \\ \frac{lm_2 \sin(\boldsymbol{x}_2)\boldsymbol{x}_4^2 + \boldsymbol{u}_1 + m_2 g \cos(\boldsymbol{x}_2)\sin(\boldsymbol{x}_2)}{m_1 + m_2(1-\cos^2(\boldsymbol{x}_2))} \\ -\frac{lm_2 \cos(\boldsymbol{x}_2)\sin(\boldsymbol{x}_2)\boldsymbol{x}_4^2 + \boldsymbol{u}_1 \cos(\boldsymbol{x}_2) + g(m_1+m_2)\sin(\boldsymbol{x}_2)}{lm_1 + lm_2(1-\cos^2(\boldsymbol{x}_2))} \end{bmatrix}_\tau,$$

$$J(\boldsymbol{u}) = \int_0^{t_F} \boldsymbol{u}^2(t) dt,$$

$$\begin{aligned} \boldsymbol{x}_1(t_1) &= 0, \quad \boldsymbol{x}_1(t_F) = 1, \\ \boldsymbol{x}_2(t_1) &= 0, \quad \boldsymbol{x}_2(t_F) = \pi, \\ \boldsymbol{x}_3(t_1) &= 0, \quad \boldsymbol{x}_3(t_F) = 0, \\ \boldsymbol{x}_4(t_1) &= 0, \quad \boldsymbol{x}_4(t_F) = 0, \end{aligned}$$

$$\begin{aligned} -2 \leq \boldsymbol{x}_1(t) \leq 2, \quad -20 \leq \boldsymbol{u}_1(t) \leq 20, \\ \boldsymbol{x} \in \mathbb{R}^4, \quad \boldsymbol{u}_1 \in \mathbb{R} \end{aligned}$$

where q is the cart position, θ is the pole angle, $\boldsymbol{u}_1$ is the force applied horizontally to the cart, m_1 is the mass of the cart, m_2 is the mass at the end of the pole, g is standard acceleration due to gravity and l is the length of the pole.

Within our code, we begin by constructing the problem definition which will be parsed to the solver.

```
numCP = 30
tf = 2
timeStep = ones(1,numCP-1) * tf / (numCP-1)
time = pushfirst!(cumsum(timeStep';dims=1)[:,1],0.0)
controlVectorGuess = zeros(1,numCP)
stateVectorGuess = [1 π 0 0]' * time' / tf
boundaryConstraints = BoundaryConstraint([0;0;0;0],[1;π;0;0])

# create user defined problem
problem = TrajProblem(objectiveFunc, dynamicsFunc, pathConstraintFunc,
controlVectorGuess, stateVectorGuess, timeStep, boundaryConstraints, 4, 1, numCP)
```

We're using 30 collocation points here, and we assume the state and control mesh points correspond exactly with the collocation points. Additionally we also define an initial guess for state and control.

Next, we define custom functions for the dynamics, the objective function and path constraints.

```
function dynamicsFunc(stateVector, controlVector)
    l = 0.5
    m1 = 1
    m2 = 0.3
    g = 9.81
    x2 = stateVector[2,:]
    u = controlVector[1,:]
    x1dot = stateVector[3,:]
    x2dot = stateVector[4,:]
    x3dot = map((x2, x2d,u) -> (l * m2 * sin(x2)*x2d^2 + u + m2 * g *
cos(x2)*sin(x2) )/ (m1 + m2 *(1 - cos(x2)^2)), x2, x2dot, u)
    x4dot = map( (x2,x2d,u) -> -
(l*m2*cos(x2)*sin(x2)*x2d^2+u*cos(x2)+(m1+m2)*g*sin(x2)) /
(l*m1+l*m2*(1-cos(x2)^2)) , x2, x2dot, u)
    return [x1dot';x2dot';x3dot';x4dot']
end

objectiveFunc(stateVector, controlVector) = controlVector.^2

function pathConstraintFunc(stateVector, controlVector) # less than or equal to
output value. Output must by 1 D vector of constraints
    dmax = 2
    umax = 20
    bounds = zeros(typeof(stateVector[1,1]),4,size(stateVector,2))
    bounds[1,:] = -dmax .- stateVector[1,:]
    bounds[2,:] = -dmax .+ stateVector[1,:]
    bounds[3,:] = -umax .- controlVector[1,:]
    bounds[4,:] = -umax .+ controlVector[1,:]
    return [bounds...]
end
```

In this formulation we are not yet considering interpolated functions, but this is of little

consiquene if we're using trapizoidal collocation, as we will be here. The disadvantage of this is that in our function formulation we have to keep in mind the array nature of our states and controls, something present in the ICLOCS package too.

Finally we define the methods for collocation and interpolations.

```
function collocateConstraint(state::Array, control::Array)
    ΔstateVector = 0.5 .* problem.timeStep[1] .*
(problem.dynamicsFunc(stateVector[:,2:end], controlVector[:,2:end]) +
problem.dynamicsFunc(stateVector[:,1:end-1], controlVector[:,1:end-1]))
    ζ = stateVector[:, 2:end] - stateVector[:,1:end-1] - ΔstateVector
    return ζ
end

function objectiveFuncInterp(state::Array, control::Array)  # can we type union
this? so we don't have to define two functions, only one
    # return interpolated integral
    return [sum(0.5 .* problem.timeStep[1] .*
(problem.objectiveFunc(stateVector[2:end,:], controlVector[2:end]) .+
problem.objectiveFunc(stateVector[1:end-1,:], controlVector[1:end-1])))]
end
```

The exact implementation within the code is slightly different due to some additional translation code for JuMP as described above, but this code could nonetheless be implemented in its shown form with some wrapper functions. The results of the solve are found below.

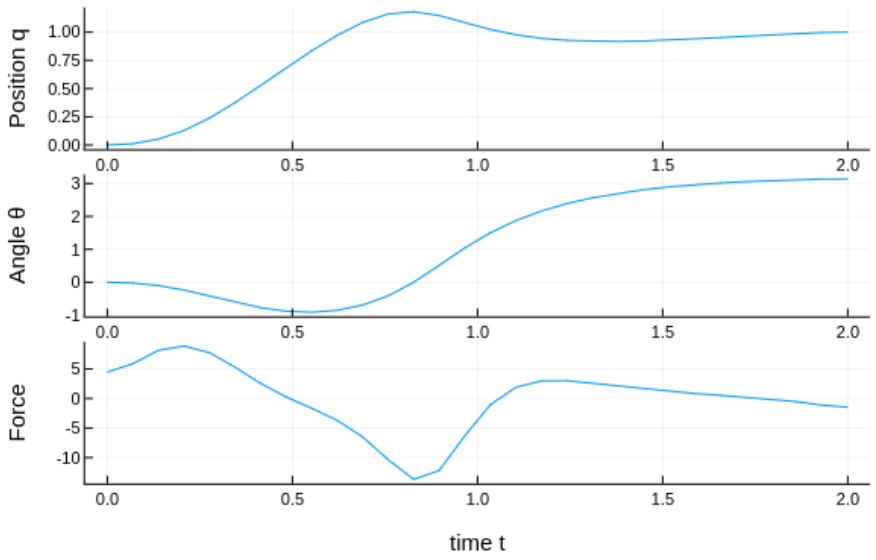

Figure 5: Cart-pole optimisation results using JuMP/JuDO

4.2.3 Example pseudospectral method

The current JuDO code can be altered to accept interpolating functions generated by MorePolynomials.jl. We will use an example problem from a paper by Garg [**?**] to demonstrate this.

$$J(\boldsymbol{x},\boldsymbol{u}) = \frac{1}{2}\int_0^{t_F} (\boldsymbol{x}_1(t) + \boldsymbol{u}_1^2(t))dt, \quad \dot{\boldsymbol{x}}_1(t) = 2\boldsymbol{x}_1(t) + 2\boldsymbol{u}_1(t)\sqrt{\boldsymbol{x}(t)}$$
$$\boldsymbol{x}_1(0) = 2, \quad \boldsymbol{x}_1(t_F) = 1, \quad t_F = 5.$$

The problem formulation is now as follows.

```
using MorePolynomials

numCP = 39
lgrPoints = lgr_points(numCP)
x = LagrangePoly([lgrPoints...,1],ones(numCP+1)) # state
u = LGRPoly(ones(numCP)) # control
boundaryConstraints = BoundaryConstraint([2],[1])
problem = TrajProblem(x, u, derivative(x),derivmatrix(x),
boundaryConstraints,[lgrPoints...,1],1,1,lgr_weights(u))
```

Note we now include a derivative matrix in our formulation. The advantage of using fixed Lagrange mesh points is that new derivatives can be calculated by a simple matrix multiplication as per equation 22. We also provide a polynomial to represent the state derivative. Internally, we now collocate the state using equation 25

```
function collocateConstraint(k, stateVector...)
    problem.x[:] = stateVector
    return problem.xdot(τ[ζr]) - problem.dynamicsFunc(τ[ζr], problem.x, problem.u)
end
```

where `k` is a reference to the index of the collocation point. Note here we treat `ydot()` as an interpolating function, rather than indexing into an array as per the cart pole example. Thus we fulfil aim 3.5. The results of the optimisation are in figure 6.

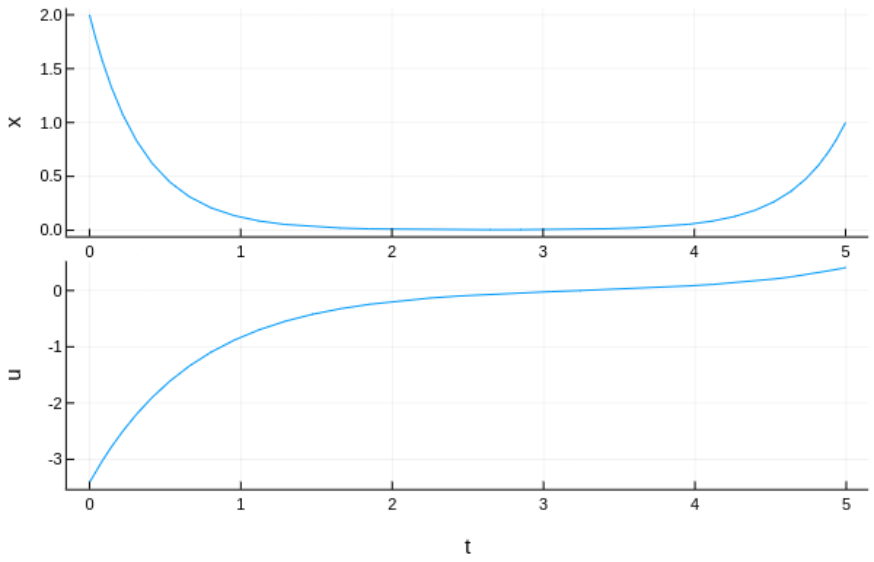

Figure 6: Pseudospectral example solution

5 Next steps

5.1 JuDO architecture

An architectural outline for JuDO is proposed in figure 7. The generic implementation prototype already employs some of the features here, but not to the full extent. The methodology behind the architecture is outlined below, and the methods proposed for how to implement each component based on experience with ICLOCS and Julia.

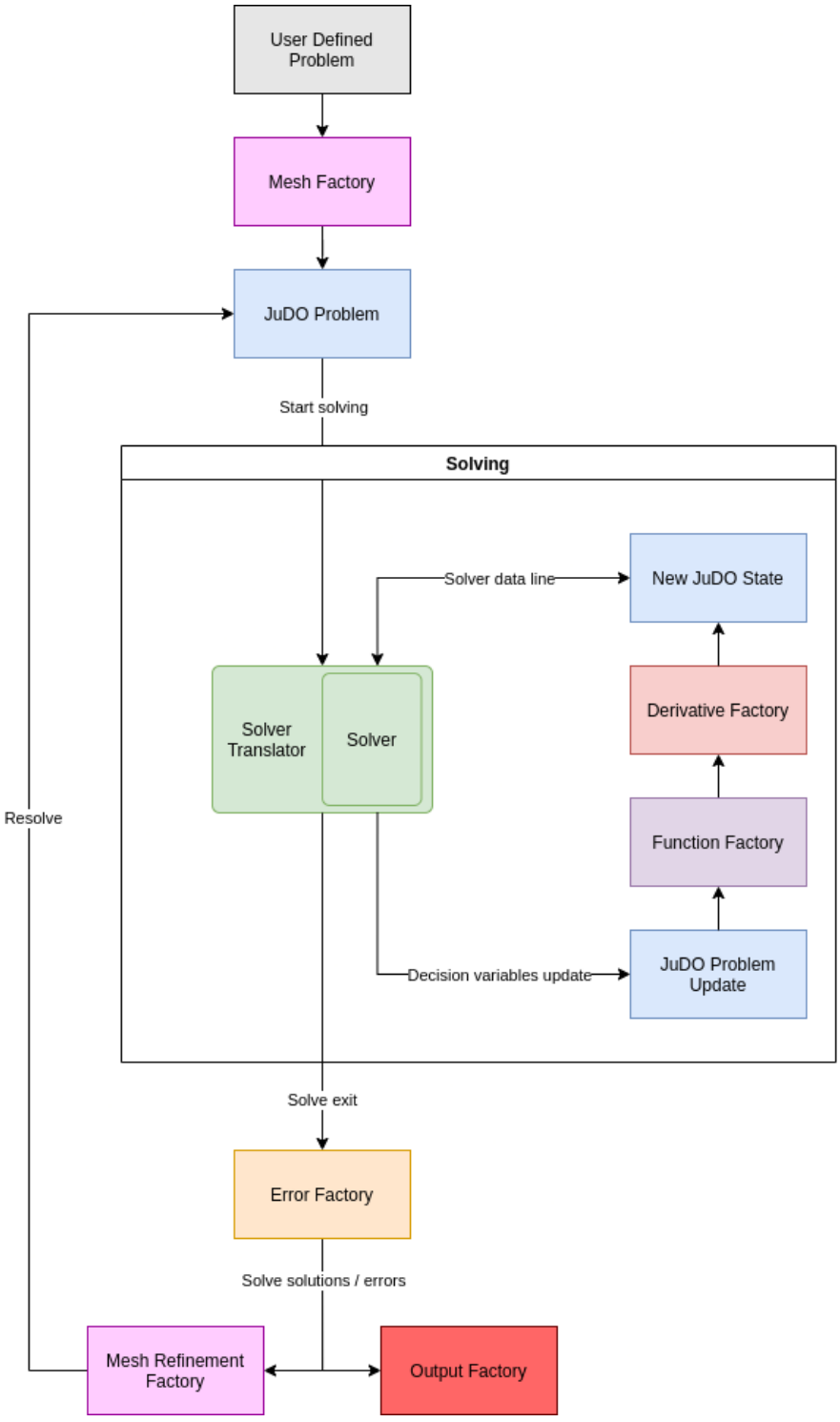

Figure 7: JuDO architecture

The architecture has eight components, five of which are referred to as factories. It is proposed that, to facilitate aim 3.1, the package should be broken up into individual factories, each of which act as an autonomous agent, with a set of standard interfaces and definitions of the problem. This interface is the JuDO problem type itself, which is parsed between factories. The role of the factory is to manage one component of the problem and provide an expected output. For example, the role of the derivate factory is to handle all derivative production within JuDO. Additionally, using multiple dispatch we can dispatch on various problem formulations within each factory and, providing the interface is standardised for a particular problem formulation, there should be minimal developer headache. We may also dispatch on particular solvers if they support a particular feature which may be exploited internally. For example a black box solver does not require derivative information, and so the derivative factory would dispatch and return nothing accordingly. This factory architecture is also useful from a development perspective. Multiple developers can work on different factories within JuDO while being confident that their interface will integrate well with the rest of the solver.

5.1.1 User Defined Problem

The aim here is to provide a standard intuitive interface for problem definitions which requires as little knowledge of the backend implementation as possible. Ideally the user would define their problem as seen in the mathematical formulation. There should be little need to try and reformulate the problem in order to work with matrix operations or specific functions. We can leverage julia macros to reformulate dynamics functions in a format understandable by our solver. For parameters such as solver selection, it is suggested that custom types should be used, such as `struct Ipopt <: AbstractSolver end`, as an example. For states and controls, a julia symbol is used. The following user defined problem syntax is suggested.

```
problem = new_JuDO_problem()
solver(problem,SolverType)
addstate(problem, :x)
addcontrol(problem, :u)
addparameter(problem, :p, val)
xdot = @JuDOFunc function xdotfunc(p,τ)
            xdot = f(x(τ), u(τ), p)
            return xdot
       end
adddynamicsfunction(problem, :x, xdot) # an example function is used here
constrain = @JuDOFunc function constrainfunc(p, τ)
            constrain = h(x(τ), u(τ), p)
            return constrain >= 0
       end
addboundsfunction(problem, :x, xdot) # an example function is used here
```

5.1.2 JuDO Problem

The role of the JuDO problem is to act as an internal standard interface between all other factories. Internally, it is represented by a datastructure containing all relevant components of the problem. Each factory either modifies or uses data from the JuDO problem. Additionally, components within the JuDO problem may have their own structures. An example

is the `AbstractState` which may contain a vector of Lagrange interpolating polynomials, as an example. This abstraction allows us to generate standard interfaces for each data component, ensuring future compatibility is part of the structure changes form (if we used a package other than MorePolynomials for example). The following datastructure is suggested.

```
mutable struct JuDOProblem <: AbstractJuDOProblem
    x::AbstractState
    u::AbstractControl
    dynamics::AbstractDynamics
    constraints::AbstractConstraint
    objective::AbstractObjective
    solver::AbstractSolver
    results::AbstractResults
end
```

5.1.3 Solver Translator

Different solvers will have different interfaces, and while we could dispatch on each solver, this incurs a significant amount of code rewriting for each solver. A more elegant solution would be to standardise the interface between each solver by using a layer of abstraction. This requires only the rewriting of a few translation functions per solver, rather than changing the entire system architecture, and is the preferred implementation method.

5.1.4 Mesh Factory

The role of the meshing factory is to generate meshes based on the user defined problem, and refine meshes after successful solves. The user can either specify what type of mesh they would like, or we can generate one for them based on the problem definition. The following function definition is proposed for the mesh factory.

```
function mesh_factory(solver::AbstractSolver, udp::AbstractUserDefinedProblem) #
initial problem formulation
    #...
    return JuDO_problem
end
function mesh_factory(solver::AbstractSolver, problem::AbstractJuDOProblem) # Mesh
refinement
```

5.1.5 Function Factory

The aim of the function factory is to provide a standard interface for handling functions, such that various transcription methods can be employed. Additionally, a standard interface will ensure compatibility with derivative calculation in the derivative factory. The following function definition is proposed for the function factory.

```
function function_factory(problem::AbstractJuDOProblem,; kwargs...)
    #...
    return problem
end
function function_factory(problem::AbstractJuDOProblem, solver::AbstractSolver;
kwargs...) # custom solver implementation
function dynamics_transcription(problem::AbstractJuDOProblem,
dynamicsFunction::Function, method::AbstractTranscriptionMethod) # system dynamics
transcription
function objective_function(problem::AbstractJuDOProblem)
function constraints_function(problem::AbstractJuDOProblem)
```

It is important to note that although we provide an interface for dispatching on the solver, the intention is to use this sparingly when implementing a feature specific to the solver. The majority of solver specific actions should be taken in the solver translator layer. Note also that each factory is not necessarily an object as in a traditional OOP program, but rather a collection of functions. This has the added benefit that these functions can be called by other functions. For example, it is useful in this case for the derivative factory to call some functions within the function factory to calculate derivatives.

5.1.6 Derivative Factory

The derivative factory calculates the derivatives of NLP functions to parse to the NLP solver. The aim here is to support multiple derivative methods, as well as sparsity patterns. Because of the standard JuDO problem interface, we can deduce from within the problem what the parameters to dispatch on for the function factory function calls should be. The following function definition is proposed for the derivative factory.

```
function derivative_factory(problem::AbstractJuDOProblem; kwargs...)
    #...
    return problem
end
function derivative_factory(problem::AbstractJuDOProblem, solver::AbstractSolver)
# custom solver implementation
function jacobian(problem::AbstractJuDOProblem)
function hessian(problem::AbstractJuDOProblem)
```

5.1.7 Error Factory

The role of the error factory is to handle any errors from the solver elegantly in a standardised manner, and to manage the analysis of errors. The error analysis can then be packaged and sent to the mesh factory for further mesh refinement. The following function definition is proposed for the error factory.

```
function error_factory(problem::AbstractJuDOProblem; kwargs...)
    #...
    return problem
end
function error_factory(problem::AbstractJuDOProblem, solver::AbstractSolver) #
custom solver implementation
function calculate_error(problem::AbstractJuDOProblem)
```

5.1.8 Output Factory

The role of the output factory is to display the results of the optimisation problem, and proved interfaces for the user to save the problem. We can do this by taking the solved problem as well as the error analysis and presenting it to the user. The output factory should ideally handle any interactions with the user, including solver errors produced by the error factory. The following function definition is proposed for the output factory.

```
function output_factory(problem::AbstractJuDOProblem; kwargs...)
    #...
    return problem
end
function print_output(problem::AbstractJuDOProblem)
function plot_output(problem::AbstractJuDOProblem)
function save_output(problem::AbstractJuDOProblem)
```