Dynamic Optimisation using the Julia Programming Language

Nathan Davey

May 29, 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)

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

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

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

Where

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

is the objective function to be minimised, and

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

are functions that represent nonlinear constraints. In this text we use bold font to denote the existance of an array of either variables or functions. In this particular case, $\boldsymbol{g}$ comprises of multiple constraint functions with multiple inputs and outputs. The 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 $\boldsymbol{x}$ is our state, $\boldsymbol{u}$ is our control input and t represents time we have

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

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 $\boldsymbol{x}$, which in reality represents a vector of states such that

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

Where m here is the number state discritzation points, otherwise known as mesh points. This is not to be confused with the number of collocation points, which is most cases is the same but in some is different. Collocation points are the points where system dynamics are enforced, but not all points where we approximate state have to comply with system dynamics. Here x represents a vector of mesh points such that $\boldsymbol{x}_{ik} = x_i(t_k)$, where $x_i(t_k)$ is the i^{th} state evaluated at time t_k. In the name of consistency, we will define some syntax. When referring to state and control, the first subscript value always refers to the index of the state or control being referenced. The second subscript value refers to the index of the mesh point being referenced. For example x_{ij} refers to the j^{th} mesh point for the i^{th} state.

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

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

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

$$\boldsymbol{x} = x_i \in \mathbb{R}^{m_i}, \quad i = 1, 2 \ldots n \tag{7}$$

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

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

$$\boldsymbol{u} = u_i \in \mathbb{R}^{m_i}, \quad i = 1, 2 \ldots c \tag{8}$$

Where c is the number of controls. The state can control points as formulated here are decision variables in our NLP problem, that will be varied by the solver in order to find the optimal solution.

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 $t = t_1, t_2 \ldots t_{m_i}$

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

Where n here is the number of both states and controls, and m_i is the number of collocation points for each state index i. The keen eye will note that for each individual function evaluation, the input array is assumed to have dimensions $n \times m_i$, while in equation 7 and 8 we observe that these arrays do not necessarily have a rectangular representation (rather they are more arrays of varying length vectors). We resolve this by interpolating the values of x and u at some time t.

Putting this all together we have

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

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

$$\boldsymbol{u}\begin{cases} u_1 = & [u_{11}, u_{12} \dots u_{1m_1}] \\ \dots & \\ u_c = & [u_{c1}, u_{c2} \dots u_{cm_c}] \end{cases}$$

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

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

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

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 the state x_1 has m_1 mesh points, then when calling $f_1(\boldsymbol{x}(t), \boldsymbol{u}(t))$, t here must be of length m_1, but the mesh points of x_2 to x_n do not have to correspond with those of x_1, 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 9. Note here h_k is the timestep $h_k = t_{k+1} - t_k$

$$\frac{\partial \boldsymbol{x}_{ik}}{\partial t} = \boldsymbol{f}_{ik} \approx \frac{\boldsymbol{x}_{ik+1} - \boldsymbol{x}_{ik}}{h_k}$$

$$\boldsymbol{x}_{ik+1} - \boldsymbol{x}_{ik} - h_k \boldsymbol{f}_{ik} = 0 \tag{9}$$

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

$$\boldsymbol{x}_{ik+1} - \boldsymbol{x}_{ik} - \frac{h_k}{2}(\boldsymbol{f}_{ik} + \boldsymbol{f}_{ik+1}) = 0 \tag{10}$$

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 9 and 10 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 9. 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()
interpolated_t = collect(0:0.01:1)
plot(interpolated_t, x1.(interpolated_t), labels = ["interpolated" ""])
scatter!(t, x1.(t), xlabel="t", ylabel="x1", labels = ["collocated" ""])
```

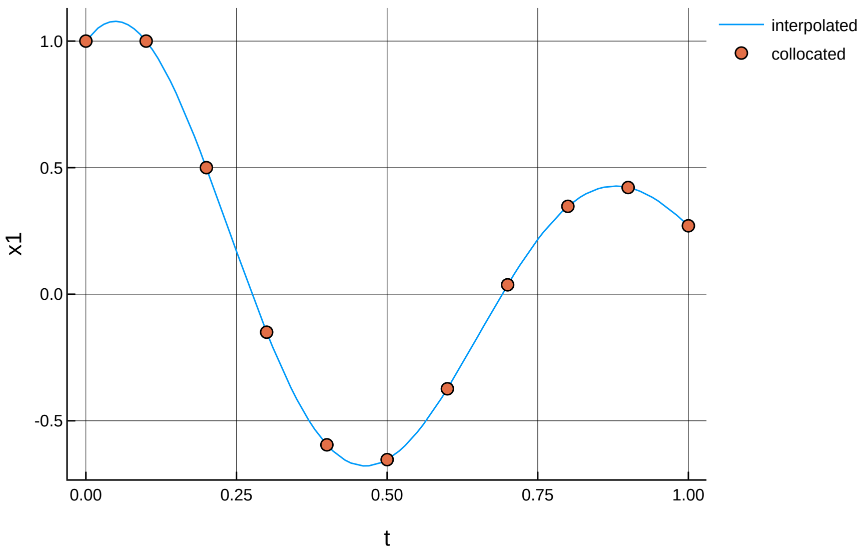

Figure 1: Collocated system dynamics for a mass spring damper

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

$$\begin{gathered} t_{1,L} \leq t_1 \leq t_{1,U} \\ t_{F,L} \leq t_F \leq t_{F,U} \\ \boldsymbol{x}_{i1,L} \leq \boldsymbol{x}_i(t_1) \leq \boldsymbol{x}_{i1,U} \\ \boldsymbol{x}_{iF,L} \leq \boldsymbol{x}_i(t_F) \leq \boldsymbol{x}_{iF,U} \end{gathered}$$

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.

$$\boldsymbol{x}_{i,L} \leq \boldsymbol{x}_i(t_k) \leq \boldsymbol{x}_{i,U} \tag{11}$$

$$\boldsymbol{u}_{i,L} \leq \boldsymbol{u}_i(t_k) \leq \boldsymbol{u}_{i,U} \tag{12}$$

$$\boldsymbol{h}(t_k, \boldsymbol{x}(t_k), \boldsymbol{u}(t_k)) \leq 0 \tag{13}$$

Once again for the sake of completeness equations 11 and 12 correspond to equation 3 and equation 13 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.

$$-1 \leq x_1(t_k) \leq 1 \tag{14}$$

$$h_1(t_k, \boldsymbol{x}(t_k), \boldsymbol{u}(t_k)) = x_1(t_k) + t_k - 1.5 \leq 0 \tag{15}$$

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.

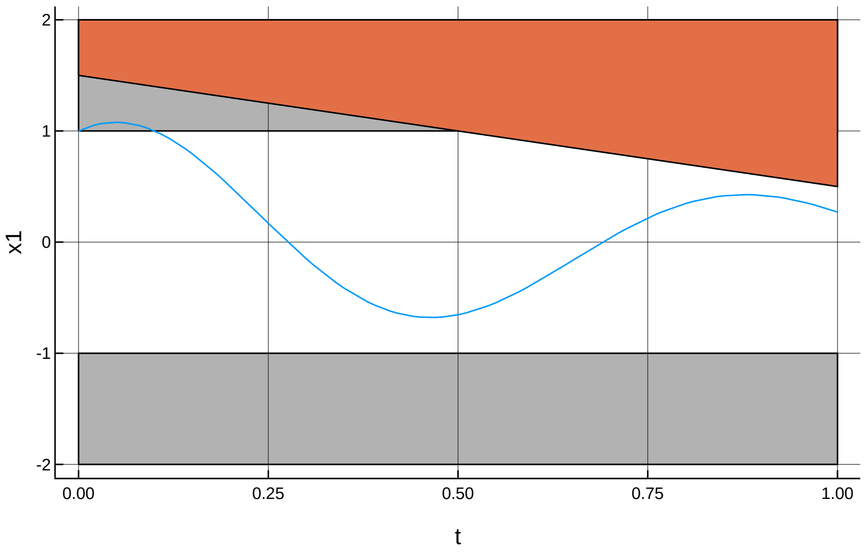

Figure 2: Example inequality constraints from equations 14 and 15

The constraint in equation 14 is violated in the grey area in figure 2, the the equation 15 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

$$J(t, \boldsymbol{x}, \boldsymbol{u}) = \underbrace{\Phi(t_1, t_F, \boldsymbol{x}(t_1), \boldsymbol{x}(t_F))}_{\text{Mayer term}} + \underbrace{\int_{t_1}^{t_F} w(t, \boldsymbol{x}(t), \boldsymbol{u}(t))dt}_{\text{Lagrange term}} \tag{16}$$

The nonlinear function in equation 16 maps directy 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

2.3 Julia Packages

2.3.1 JuMP.jl

2.3.2 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 **??**, 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 2.3.2).

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.

```julia
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

```julia
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

```julia
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.

```julia
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.

```julia
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. The types all fields are parameterised 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. 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:             15.184 μs
  gctime:           504.154 ns (3.32%)
  memory:           8.44 KiB
  allocs:           40

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

BenchmarkTools.TrialEstimate:
  time:             1.683 μs
  gctime:           62.977 ns (3.74%)
  memory:           768 bytes
  allocs:           29
```

This is perhapse an undfair comparison as Polynomials.jl has the additional step of converting to a coefficient representation. A more fair comparison would be from time to creation to first evaluation. Let's time this.

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

BenchmarkTools.TrialEstimate:
  time:             15.572 μs
  gctime:           521.111 ns (3.35%)
  memory:           8.45 KiB
  allocs:           41

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

BenchmarkTools.TrialEstimate:
  time:             1.726 μs
  gctime:           61.517 ns (3.56%)
  memory:           784 bytes
  allocs:           30
```

4.1.2 User interface

4.1.3 Error handling

4.1.4 Benchmarking and performance gains

5 Documentation

6 Next steps

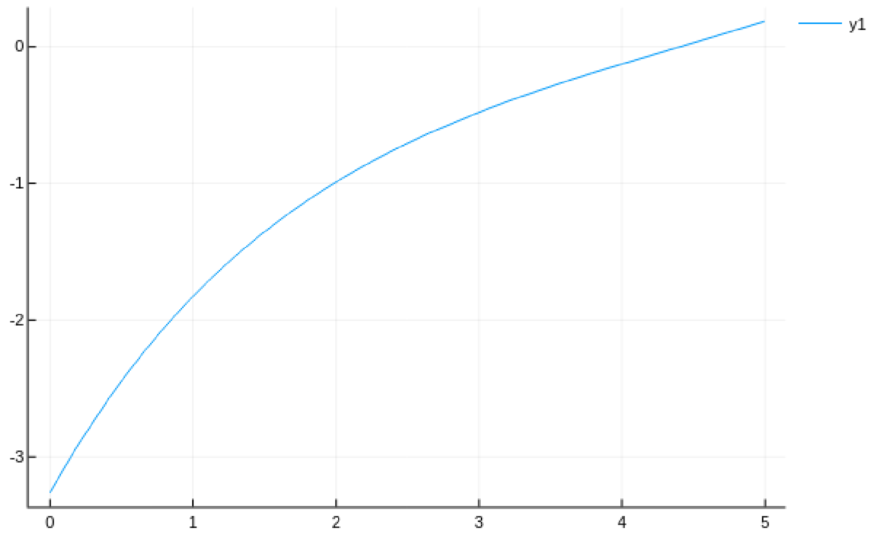

Figure 3: a

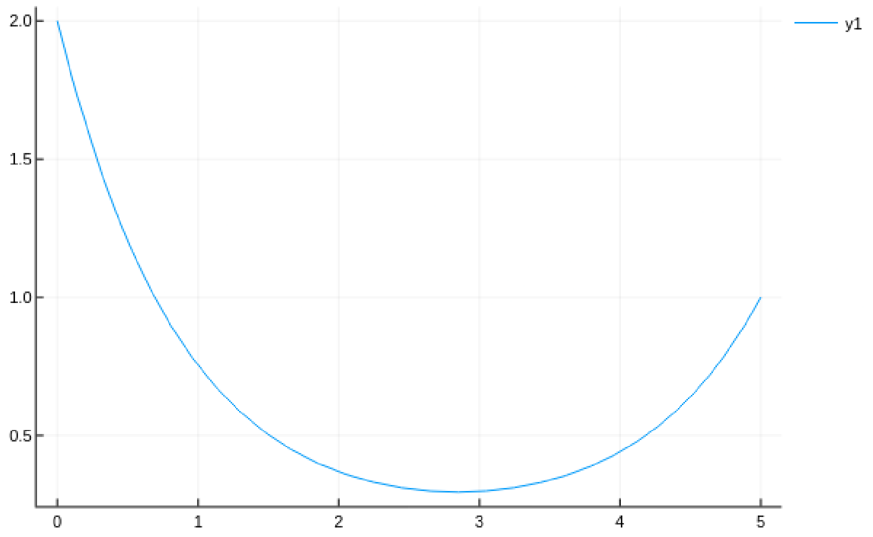

Figure 4: a

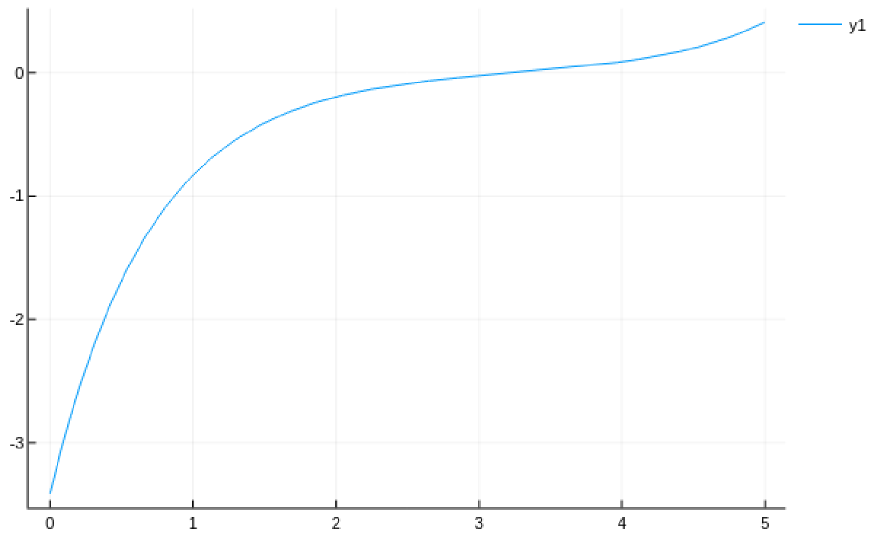

Figure 5: a

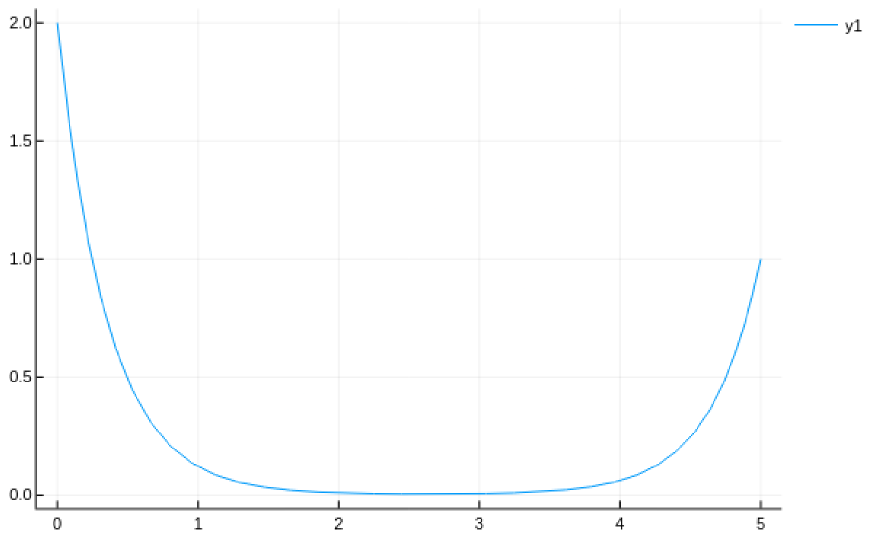

Figure 6: a