Bachelor's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Imre Angelo

Implementation of an optimal control solver for linear problems

Bachelor's thesis in Mathematical Sciences
Supervisor: Markus Arthur Köbis
June 2023

NTNU
Norwegian University of
Science and Technology

Imre Angelo

Implementation of an optimal control solver for linear problems

Bachelor's thesis in Mathematical Sciences
Supervisor: Markus Arthur Köbis
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Abstract

Optimal control theory is pivotal to many fields of science, engineering, and economics, as it provides a mathematical framework to determine the control that optimizes a given measure over time. This thesis details the development and analysis of a linear optimal control solver written in C++/C. The solver was implemented using complete parameterization, a direct discretization method for optimal control.

Abbreviations

ODE	Ordinary Differential Equation
DAE	Differential-Algebraic Equation
CVP	Control Vector Parameterization
LP	Linear Programming
NLP	Non-Linear Programming
CPLEX	The IBM ILOG CPLEX Optimizer
SIMD	Single Instruction, Multiple Data

Table of Contents

Chapter 1

Introduction

Optimal control theory is the mathematical framework that allows us to control systems in the best possible manner. It has seen widespread use in various disciplines including engineering, economics, and computer science. This theory, developed in the mid-20th century, is now an essential tool for determining the best possible performance of a control system, be it in guiding a spacecraft, optimizing production in a factory, or predicting economic trends.

Most optimal control problems are not analytically solvable, or are at least very hard problems to solve analytically, necessitating the use of numerical methods. While this thesis does explore the basic theory of optimal control, there is a significant focus on the practical application of this theory. Its epicenter is the development and analysis of a numerical solver for linear optimal control problems, written in C++/C.

1.1 Outline of Thesis

As someone who started this project with little-to-no experience in applied mathematics, I wrote this thesis with the assumption that potential readers might also lack a solid foundation in applied mathematics. Note, however, that some parts, especially the implementation details of Chapter 5, is intended for someone who wishes to extend or use the solver as a library, and might require some degree of knowledge of low-level programming to fully comprehend. These parts can be skipped if the reader is simply interested in the mathematics at play.

The thesis is therefore divided into two parts; the first part provides a high-level introduction to the mathematical theory employed by the solver, while the sequel details the solver itself and presents some numerical tests performed with and on the program.

Chapter 2 begins with a brief review of one-step methods, particularly the Runge-Kutta family of one-step methods. Chapter 3 offers a simple introduction to optimization, and to the third-party optimizer, CPLEX, that is used as the final step of the optimal control solver. Chapter 4 looks at direct discretization methods of optimal control problems, and explains in detail one such method called complete parameterization.

The second part of the thesis begins in Chapter 5 with a detailed description of the program developed as part of this thesis. Chapter 6 introduces some example problems that we will use for a series of numerical tests in Chapter 7. Finally, we summarize the thesis in Chapter 8 and discuss some of the possible future directions for the solver.

Chapter 2

One Step Methods for Ordinary Differential Equations

The dynamics of an optimal control problem is modelled by a set of equations, some of them differential. Our ability to solve optimal control problems thus relies on our ability to solve ODE's. In this chapter we cover the standard approach to solving ODE's of the form (2.1) numerically.

$$\dot{y} = f(t, y(t)), \quad y(t_0) = y_0, \quad t \in [t_0, t_1] \tag{2.1}$$

2.1 Euler's Method

Perhaps the simplest numerical technique used to approximate the solution to (2.1) is Euler's method. The method itself is given by the simple iteration (2.2) which results in a set of N discrete approximations of the function $y(t)$ at equidistant points t_n separated by a distance of $h = (t_{end} - t_0)/N$, called the *step size*, where t_0 and t_{end} denote the start and end of the interval.

$$y_{n+1} = y_n + hf(t_n, y_n) \tag{2.2}$$

Euler's method can be derived directly from the first order Taylor polynomial. This derivation also gives the error bound on each iteration of the method, called the *local truncation error*, since we know from analysis that the remainder of the n-th order Taylor expansion around t is $\mathcal{O}((t-a)^{n+1}) \leq C \cdot (t-a)^{n+1}$ for some constant C and fixed point a. If we use the notation that $t_n = t_0 + n \cdot h$ and $y_n = y(t_n)$, then choosing $a = t_n$ for the first order Taylor expansion around $t_n + h$ gives us:

$$\begin{aligned} y(t_n + h) &= y(t_n) + h\,\dot{y}(t_n) + \mathcal{O}(h^2) \\ y(t_{n+1}) &= y(t_n) + hf(t_n, y(t_n)) + \mathcal{O}(h^2) \\ y_{n+1} &= y_n + hf(t_n, y_n) + \mathcal{O}(h^2) \end{aligned} \tag{2.3}$$

While Euler's method is straightforward and computationally cheap, it only takes into account the slope at the starting point of each interval and might therefore require a very small step size to achieve an acceptable accuracy. This is where Runge-Kutta methods come in.

2.2 Runge-Kutta Methods

One-step methods (2.4) extend the basic idea introduced by Euler's method, where each step is approximated only by the previous step. They provide more accurate approximations by incorporating additional information within each step, while still being computationally cheap.

$$y_{n+1} = y_n + \Phi(t_n, y_n, h) \tag{2.4}$$

Runge-Kutta methods have long been established as the standard family of one-step methods. This family sets the increment function $\Phi(\cdot)$ to be the weighted sum of s intermediate steps k_i, each of the form (2.5).

$$k_i = f(x_n + c_i h,\ y_n + h \sum_{j=1}^{s} a_{i,j} k_j) \tag{2.5}$$

$$y_{n+1} = y_n + h \sum_{i=1}^{s} b_i k_i \tag{2.6}$$

The coefficients are often presented in a concise table called a *Buthcer tableau* or table. The number of intermediate steps k_i determines the order s of the table. This is not to be confused with the order of the method represented by the table. If the local truncation error of a Runge-Kutta method is $\mathcal{O}(h^{n+1})$ for sufficiently smooth problems, the method is of order n.

c_1	$a_{1,1}$	$a_{1,2}$	$\cdots$	$a_{1,s}$
c_2	$a_{2,1}$	$a_{2,2}$	$\cdots$	$a_{2,s}$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
c_s	$a_{s,1}$	$a_{s,2}$	$\cdots$	$a_{s,s}$
	b_1	b_2	$\cdots$	b_s

Table 2.1: Butcher Table

Different methods use very different strategies for determining the coefficients. One approach might be to find the coefficients that minimize the local truncation error of the method for a very specific class of ODEs, while another approach could be to design methods that conserve certain properties of the system being modeled. There are books and research papers devoted solely to the derivation of these coefficients, and for a more rigorous and in-depth introduction to this topic the reader is referred to [3].

Runge-Kutta methods can be explicit or implicit. Explicit methods are computationally efficient, as each stage k_i depends only on the previous stages. This also means explicit methods correspond to Butcher tables with all non-zero entries $a_{i,j}$ below the diagonal. In implicit methods the stages are interdependent, leading to a system of equations to be solved at each step. These methods offer more stability and are better suited for *stiff* differential equations, which are loosely defined as equations that lead to rapid variation in the solution and thus numerical instability with the use of explicit methods.

Chapter 3

Optimization with CPLEX

Optimal control is an extension of optimization, the process of maximizing or minimizing a given function. The purpose of this chapter is not to provide a thorough overview of optimization, for that the reader is referred to [4], but rather to introduce some terminology that will be useful when we begin looking at optimal control theory. We also introduce the CPLEX optimizer, which is used by the linear optimal control solver.

3.1 Definition

We want to choose a point x, called a *decision variable*, that minimizes some *objective* function $J(x)$. Constrained optimization deals with problems where x must satisfy a set of equality and inequality constraints. The typical constrained optimization problem therefore looks like:

$$\begin{aligned} \min_x \ & J(x) \\ \text{subject to } & g(x) \leq 0 \\ & h(x) = 0 \end{aligned} \tag{3.1}$$

The exact algorithm or optimizer used to solve a specific optimization problem depends on the characteristics of the problem; whether it is constrained or unconstrained, linear or non-linear and so on.

3.2 Linear Programming

Linear programming (LP) is a method for solving optimization problems where the objective function and constraints are all linear. Problems of this form are called linear programs, and are generally formulated as:

$$\begin{aligned} \min_x \quad & \mathbf{c}^\top \mathbf{x} \\ \text{s.t.} \quad & \mathbf{A}\mathbf{x} \leq \mathbf{b}, \\ & \mathbf{x} \geq \mathbf{0}, \end{aligned} \tag{3.2}$$

LP has some very desirable qualities. If a feasible and bounded solution exists, LP guarantees that an optimal solution can be found. Additionally, existing algorithms are well-developed, highly efficient and very capable of handling large-scale problems. In fact, LP is so desirable that a common approach to non-linear optimization is *linearization*, where non-linear problems are approximated by linear problems.

3.3 CPLEX

This thesis is primarily focused on the implementation of an optimal control solver, not an optimization solver. Yet, as we shall see in the next chapter, we will be required to solve an optimization problem as a necessary subroutine in the optimal control solving routine, or program. For this purpose we will be using a third party optimizer, namely the IBM ILOG CPLEX Optimizer, or CPLEX for short.

CPLEX provides a suite of optimization algorithms for linear, mixed-integer and quadratic programming. By default, CPLEX will automatically choose an appropriate algorithm from this suite. We are not particularly interested in which algorithm is used, instead we treat CPLEX like a black box and concern ourselves only with the input and the output of the optimizer. As long as we are able to feed a valid optimization problem into CPLEX, we will assume the output is correct.

Chapter 4

Optimal Control Theory

Where optimization was concerned with finding a *point* where some objective function is minimized, or maximized, optimal control theory is concerned with the same for functions of time. In this chapter we will look at direct discretization methods for optimal control, and one such method in particular called complete parameterization.

4.1 Definition

We wish to minimize some objective function which represents a certain cost over time. The objective is determined by control variables, which can be manipulated to influence the system's behavior, and state variables, which represent the system's current condition. The goal is then to find the control $u(t)$ with the lowest cost. The dynamics of the system, how it evolves over time, is described by a set of differential and algebraic constraints.

$$\begin{aligned}
\min_{u(t)} \int_{t_0}^{t_{end}} & I(t,y,u)\ \mathrm{dt} + K(t,y,u) \\
\text{s.t. }\ 0 &= F(t, y(t), \dot{y}(t), u(t), f(t)) \\
0 &\leq H(t, y(t), u(t), h(t)) \\
lb_u(t) \leq u(t) &\leq ub_u(t) \\
lb_y(t) \leq y(t) &\leq ub_y(t)
\end{aligned} \tag{4.1}$$

The objective usually consist of the definite integral of some function $I(\cdot)$ over the *time horizon* $[t_0,\, t_{end}]$, as well as some non-integral term $K(\cdot)$ to penalize the final state of the system. Perhaps we are modelling the flight path of a commercial airliner, and we wish to minimize transit times while using as little fuel as possible. The control $u(t)$ might represent the engine throttle, while the state represents the remaining distance $y_1(t)$ and fuel $y_2(t)$. We could then let $I(\cdot) = y_1(t)$ and $K(\cdot) = -y_2(t_{end})$ to incentivize high speeds and lower fuel consumption.

4.2 Direct Discretization Methods

There are two approaches to optimal control: direct methods and indirect methods. The latter transforms the problem into a different problem type, often a multi-point boundary value problem, which is then discretized and solved numerically. Direct methods discretize the optimal control problem itself, hence the name. The result of this discretization is a finite-dimensional optimization problem which can be solved numerically by standard optimization algorithms.

All direct methods work on the grid $\mathbb{G}_N$, where N denotes the number of intervals in the grid. Although not a requirement, the grid is often an equidistant partition of the time horizon $[t_0, t_{end}]$ with step size $h = (t_{end} - t_0)/N$.

$$\mathbb{G}_N = \{t_0 < t_1 < \cdots < t_{N-1} = t_{end}\} \tag{4.2}$$

4.2.1 Algorithm

Direct methods can generally be characterized by the following steps, which can be carried out in any order. The optimization step, however, typically depends on some of the other steps.

- **Control Discretization**
  The control is approximated by a function of M parameters, where M is finite and is typically a function of N.
- **State Discretization**
  The ODE is discretized by a suitable scheme, such as a one-step method, on the grid $\mathbb{G}_N$.
- **Constraint Discretization**
  The objective function and the algebraic constraints are only evaluated on the grid $\mathbb{G}_N$.
- **Calculation of Gradients**
  This step involves the computing of any derivatives required by the optimizer, if there are any. Not all optimization methods require any gradients or Jacobian matrices, so this step is not always present. For instance, when the optimal control problem is linear, the resulting optimization problem becomes a linear program and no additional values are required.
- **Optimization**
  Finally, we have a NLP problem that must be solved numerically by some optimizer. In the case that the optimal control problem has a linear objective function and linear constraints, the resulting problem is formulated as a linear program and can be solved with LP instead.

4.2.2 Complete Parameterization

In complete parameterization, also called full discretization, both the control and state variables are parameterized, meaning approximated by a function of a finite number of parameters. For comparison, another common direct method called *control vector parameterization* (CVP), also known as reduced discretization, only parameterizes the control. By using more parameters, complete parameterization is computationally more expensive but also more robust than CVP.

We begin the complete parameterization approach by creating two vectors y and u to hold the parameters, where y holds N parameters and u holds M parameters. Each parameter y_n and u_n is a vector of the same size as $y(t)$ and $u(t)$, respectively. The exact relationship between N and M depends on the discretization schemes being used. For instance, when using a one-step method for the state discretization it is natural to let $M = N - 1$, since each control input u_n is used to drive the system from a state y_n to the next state y_{n+1}.

$$y = (y_1, y_2, \ldots, y_N), \quad u = (u_1, u_2, \ldots, u_M) \tag{4.3}$$

When discretizing the state ODE, the key principle is that each discrete point of the solution $y(t)$ is a function of the parameters u and y. Then the discrete points cannot be evaluated, but instead act as constraints on the parameters. Using Euler's method as an example, each iteration yields a constraint on y_{n+1}, y_n and u_n:

$$y_{n+1} = y_n + h \cdot f(t_n, y_n, u_n) \tag{4.4}$$

The evaluations of the algebraic constraints and objective on the grid also places similar constraints on the parameters. The result is a discrete optimization problem where the decision variables are the parameters u and y. Given that a solution exists, the optimization step will assign values to the previously unknown parameters according to (4.5). In other words, the parameters have become discrete approximations for $u(t)$ and $y(t)$ on the grid $\mathbb{G}_N$.

$$y_n \approx y(t_n), \quad u_m \approx u(t_m) \tag{4.5}$$

Still, the parameters are only discrete approximations, and we might wish to return a continuous function. All that remains, then, is to construct the actual approximations of $u(t)$ and $y(t)$ from the parameters u and y. The two obvious approaches is to either interpolate between the parameters, or approximate $u(t)$ as a piece-wise constant function. In practice, we often simply return the parameter values, and leave the construction of the approximations to the user.

Chapter 5

Implementation of Complete Parameterization in C++/C

A major part of this project was developing a linear optimal control solver in C++/C. This section will detail some of the specifics of the program and the choices made during development. The program, whose code is open-source on GitHub [1], uses complete parameterization, and IBM's CPLEX as the optimizer, to solve problems of the following form:

$$\min_{u(t),y(t)} \int_{t_0}^{t_{end}} e^{-\varphi \cdot t} \cdot \mathbf{\Phi}_1^\top \cdot \mathbf{y}(t) \, \mathrm{d}t + \mathbf{\Phi}_2^\top \mathbf{y}(t_0) + \mathbf{\Phi}_3^\top \mathbf{y}(t_{end}) \tag{5.1}$$

$$\text{s.t.} \quad \dot{\mathbf{y}}(t) = \mathbf{F_u} \cdot \mathbf{u}(t) + \mathbf{F_y} \cdot \mathbf{y}(t) + \mathbf{F_c} \tag{5.2}$$

$$\mathbf{g}(t) = \mathbf{G_u} \cdot \mathbf{u}(t) + \mathbf{G_y} \cdot \mathbf{y}(t) + \mathbf{G_c} \tag{5.3}$$

$$\mathbf{lb} \leq \mathbf{u}(t) \leq \mathbf{ub} \tag{5.4}$$

$$\mathbf{0} \leq \mathbf{y}(t) \tag{5.5}$$

$$\mathbf{h}(t) \leq \mathbf{H_u} \cdot \mathbf{u}(t) + \mathbf{H_y} \cdot \mathbf{y}(t) \tag{5.6}$$

$$\mathbf{b} = \mathbf{B_0} \cdot \mathbf{y}(t_0) + \mathbf{B_{end}} \cdot \mathbf{y}(t_{end}) \tag{5.7}$$

Let m be the number of dimensions of the state $\mathbf{y}(t)$ and control $\mathbf{u}(t)$. Then $\mathbf{F}_{(\cdot)}$, $\mathbf{G}_{(\cdot)}$ and $\mathbf{H}_{(\cdot)}$ are $m \times m$ matrices whose elements are functions of time. The control is bounded element-wise by two constant vectors **lb** and **ub**, while the state is only required to be non-negative. We have split the DAE of (4.1) into a differential equation (5.2) and a set of algebraic equations, one equality equation (5.3) and one inequality equation (5.6). Finally, there are some boundary conditions for the start and end states.

We chose to focus on linear problems as a first step. As mention earlier, it is not uncommon to approximate non-linear systems by linear models, and as we move forward the idea is to solve non-linear problems by linearization. The interface (5.1) was inspired by real models used in the field of biology. One such model will be investigated closer as example 3 in the next chapter.

With the hope of creating a tool with real world applications, the program was developed as a standalone library to allow easy integration with other applications. Initially, I wanted a python binding to leverage the ease-of-use and immense popularity of python, but since the focus of this thesis was the mathematics, not a programming exercise, little time was allocated for designing the application for portability. For the time being, the solver is integrated into the visualization software made specifically for Chapter 7.

I chose to write the solver in C++/C due in part to the sheer speed of the language. Even simple optimal control problems can reach thousands of constraints. In fact, its not uncommon for real world applications to be modelled using millions of variables. Even the smallest inefficiency in determining a single variable could propagate to all other variables, massively amplifying the inefficiency.

The actual program uses the algorithm for direct methods described in Chapter 4, with no gradient step since we only care about linear problems. The integral of the objective is evaluated on the grid by using the trapezium rule. The state discretization takes a Butcher table as an argument and uses the corresponding Runge-Kutta method. The solver returns only the parameters u and y, and the visualization software approximates $u(t)$ and $y(t)$ by linear interpolation.

5.1 CPLEX Concert Technology API for C++

The library developed as part of this thesis uses CPLEX, the optimization solver introduced in Chapter 3, to solve the linear program created as part of the complete parameterization method. This section will briefly explain how CPLEX is used, as well as a particular quirk of the C++ API.

In C++/C we have two choices for interfacing with CPLEX. The first is the 'C callable library' which exposes a procedural API written in C. Like many C libraries, this API offers very fine control of the program at the cost of manual error handling and memory management. The other option is Concert Technology, or Concert for short, which wraps the C API in object oriented C++ objects. Concert handles errors and memory automatically, so long as we remember to call a special IloEnv::end() function when we are done. For these reasons I chose Concert for this project.

The basic usage of Concert looks like this: We create an environment (IloEnv) which is responsible for managing our memory. All other Concert objects, identified by their "Ilo" prefix, are created inside of this environment. We then create a model (IloModel) to hold our constraints. When we create our decision variables (IloNumVar), we must give them an upper and lower bound. If we want an unbounded decision variable, the bounds can be set to the maximum value supported by the computer. We also need to add an objective to our model. When we are ready to solve the model, we create an IloCplex object from the model and use the function IloCplex::solve() before finally extracting the results.

```cpp
IloEnv env;
IloModel model(env);

// Make decision variables (with arbitrary bounds lb, ub)
IloNumVar a(env, a_lb, a_ub);
IloNumVar b(env, b_lb, b_ub);

// Add a constraint
model.add(a == 2*b);

// Set objective to min(a + b)
model.add(env, IloMinimize(a + b));

// Solve the model with CPLEX
IloCplex cplex(model);
cplex.solve();

// Extract results
float aVal = cplex.getValue(a);
float bVal = cplex.getValue(b);

// Free memory
env.end();
```

Using Ilo-prefixed objects in any mathematical expression will yield an IloNumExpr object. This class allows for the construction of more complex constraints, and is therefore heavily utilized by the program. All classes from the Concert API act like reference types, meaning they hold references to other IloNumExpr objects.

Exactly how this is implemented is unclear, as Concert is not open-source, it is clear. What is clear, however, is that when an expression is updated it must be resolved to some canonical form. When numerical expressions contain other numerical expressions, this resolution must propagate to every IloNumExpr that indirectly references or is referenced by any of the affected IloNumExpr.

This reference structure allows for the easy construction intermediate steps, especially when using implicit Runge-Kutta methods, as described in more detail in the next section. However, any developer who wishes to extend the program should note that the resolution described above is slow and some care should be taken to avoid unnecessary references.[1] The important takeaway is that Concert is easy to use from a mathematical perspective, at the cost of being somewhat slow and difficult to predict the behaviour of in certain edge cases.

Furthermore, IloNumExpr objects must be initialized at declaration, and they must be initialized with at least one reference to another Concert object. Once an IloNumExpr is linked to another object, this reference cannot be reassigned or removed. This is very inconvenient when constructing complex expressions, so I develop the following workaround which uses an empty decision variable to create an expression that evaluates to 0. The decision variable is unused and therefore ignored by CPLEX when solving the model.

```
IloNumVar z(env, 0, 0);
IloNumExpr zero = z - z;
```

5.2 Matrix Operations with Eigen3

I wanted to use the Eigen3 math library for vector and matrix operations, because Eigen3 is an extremely fast linear algebra library. Most of the speed is achieved by taking advantage of *single instruction, multiple data* (SIMD) instructions. Using these instruction is very cumbersome, as every computer architecture has its own specific implementation, and implementing SIMD is therefore generally left to specialized math libraries.

Most processors implement instruction sets that allow performing basic arithmetic on multiple variables simultaneously, with no performance loss. In a process called *vectorization*, several scalars are packed into a single, large register. Modern home computers typically allow packing up to eight or even sixteen 32-bit scalars into a single 256-bit or 512-bit register. This allows the same operation to be performed on all the packed scalars at the same time, facilitating a theoretical 8-16 times improvement for supported operations, though in practice there is some overhead associated with the packing and unpacking of the values.

SIMD requires the use of primitive data types, such as integers or floating-point numbers. We perform arithmetic directly on classes from Concert, and are therefore not taking advantage of SIMD. This could be implemented in the future, but as Concert enforces an object-oriented design, this would likely require a change to the procedural C callable API. It is safe to assume this is possible with the C API, since, by the nature of being written in C, this library can only use primitives.

[1]It falls outside the scope of this thesis, but the Runge-Kutta method was originally written to be *branchless*, a coding style that should, in this case, allow for a specific compiler optimization to improve the speed of the code. However, the penalty of propagating 0-valued expressions (zero) to all other IloNumExpr objects far outweighed the penalty of branch-prediction misses. Using an if statement to skip such expressions more than doubled the speed of the code.

5.3 Parameterization of the Ordinary Differential Equation

Except for the state discretization, the solver is a fairly straight-forward implementation of the algorithm described in Chapter 4. The code is documented by comments and a ”readme” file, but the state discretization might benefit from a deeper explanation. The state is discretized by a one-step method; specifically, the interface is designed to take a Butcher table as an argument. Note that the code uses 0-index notation, so we will do the same in this section.

With respect to the equidistant grid $\mathbb{G}_{\mathbb{N}}$, we use N parameters for $y(t)$ and $M = N - 1$ parameters for $u(t)$. Let s be order of the given butcher table, and m the number of dimensions of $u(t)$ and $y(t)$. First we initialize an ”empty” $s \times m$ matrix k of numerical expressions, where the i-th row of k represents the vector k_i.

```
Matrix<IloNumExpr> k(s, m);
for (auto i = 0; i < s; i++)
    for(auto j = 0; j < m; j++)
        k(i,j) = zero;
```

We construct each k_i sequentially, of which the most complicated part is the creation of the sum $\sum_{j=0}^{s-1} a_{i,j}k_j$. The reference structure of IloNumExpr discussed in the previous section means that we can simply use k_j in the construction of k_i, even when $i < j$. In this case, k_j is zero at the time of construction, but any changes to the value of k_j will propagate to k_i.

The following code creates the sum $\sum_{j=0}^{s-1} a_{i,j}k_j$. Note that the rest of k_i is created in the same iteration of i, but this construction is very straight forward and thus excluded to make the code more readable and concise.

```
for (auto i = 0; i < s; i++)
{
    Matrix<IloNumExpr> sum(s, 1);

    for (auto j = 0; j < s; j++)
    {
        IloNumExpr expr = zero;

        for (auto ii = 0; ii < s; ii++)
            if(table.a[i][ii])
                expr = expr + table.a[i][ii] * k(ii, j);

        sum(j) = expr;
    }
    // Construct k_i here and set the i-th row of k equal to that
}
```

Once we have constructed all intermediate values k_i, we add the constraints to our model. This whole process is repeated $N - 1$ times, until all the parameters are included in some constraint.

```
for (auto j = 0; j < m; j++)
{
    IloNumExpr biki = zero;

    for (auto i = 0; i < table.order; i++)
        biki = biki + k(i, j) * table.b[i];

    // Add constraint
    model.add(y(j, n + 1) == y(j, n) + biki);
}
```

Chapter 6

Examples

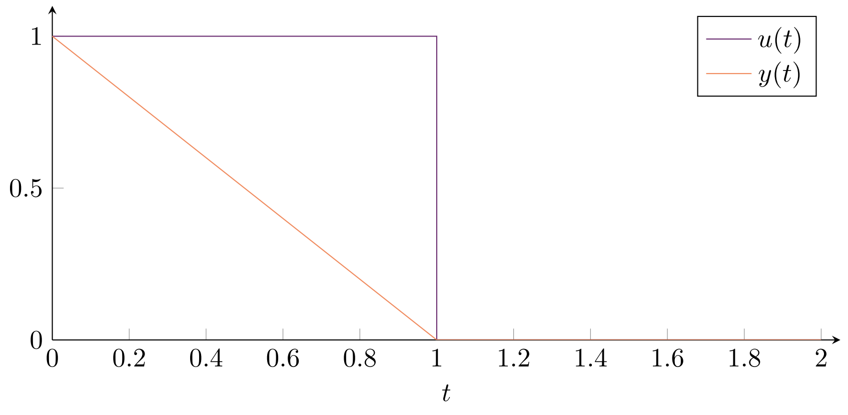
This section will introduce some example optimal control problems and their solutions. These problems will be used in Chapter 7 to evaluate various aspects of the solver.

6.1 Example 1

The first problem was chosen to be as simple as possible, without being entirely trivial. The model is simple enough that we can predict the expected output of the program.

$$\begin{aligned}
&\min_{u(t),y(t)} \int_0^2 y(t)\,\mathrm{dt} \\
&\text{s.t. } \dot{y} = -u(t) \\
&\quad 0 \le u \le 1 \\
&\quad 0 \le y \\
&\quad y(0) = 1
\end{aligned} \tag{6.1}$$

The control is proportional to the rate of change of the current state of the system. Perhaps the control is the thrust of a rocket engine, and the state is the remaining fuel. In that case, we can power the rocket by spending fuel. Our goal would then be to use all the fuel as fast as possible, as we want to minimize the definite integral of the state, so we expect the control to be at full capacity until the state is drained and the control "switches off" instantly.

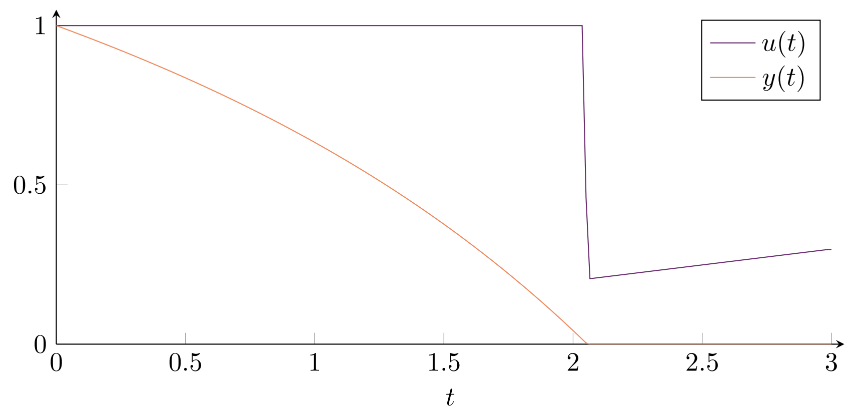


6.2 Example 2

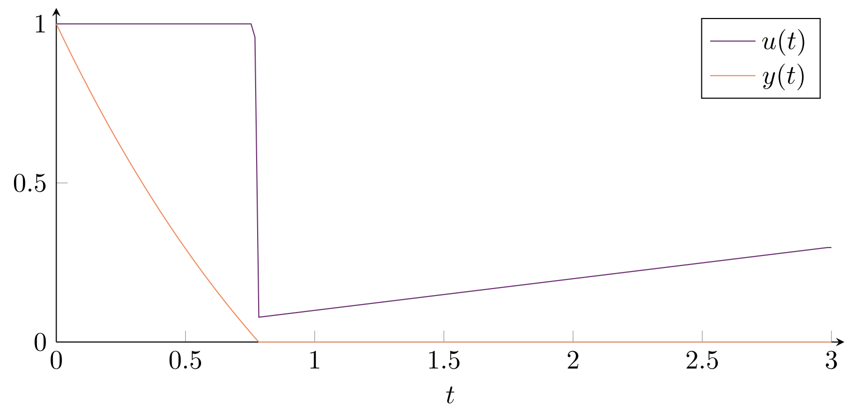
The second example is only a slight complication of the previous example. We add a constant term $\alpha \cdot t$ and a state term $\beta \cdot y(t)$ to the differential equation. Although we do not have an analytic solution to this problem, the model is still simple enough that we can "predict" the solution and verify if a given solution makes sense.

$$
\begin{aligned}
&\min_{u(t),y(t)} \int_0^3 y(t)\,\mathrm{dt} \\
&\text{s.t. } \dot{y} = \alpha \cdot t + \beta \cdot y(t) - u(t) \\
&\quad 0 \le u \le 1 \\
&\quad 0 \le y \\
&y(0) = 1
\end{aligned}
\tag{6.2}
$$

The following solution to the problem with $\alpha = 0.1$ and $\beta = 0.7$ was found by the solver. We expect the control to sustain its maximum output until the state reaches 0, at which point the control must counteract the constant term $0.1 \cdot t$ of the system to maintain the current state.



As expected, the state falls to 0 as fast as possible, before the control acts to sustain the current state. The curvature of the state is a result of the differential term $\beta \cdot y(t)$ that dictates the system will grow with strength proportional to $y(t)$. If β was negative, for example if we let $\beta = -0.7$, the system would resist growth instead, and the graph would curve in the other direction, as well as reach the zero-state sooner:

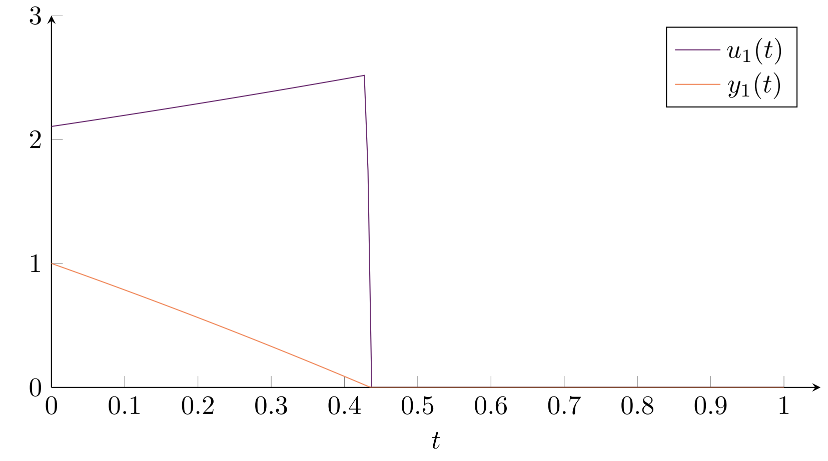


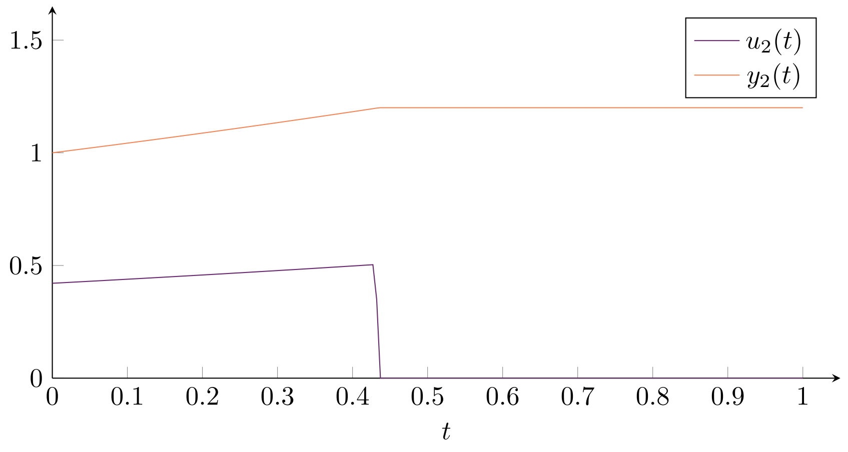
6.3 Example 3

The next example is inspired by the field of biology, and models a microbe. The first state variable $y_1(t)$ represents nutrients available to the microbe. The microbe is able to control the intake of nutrients with $u(t)$. The nutrients consumed are converted into mass $y_2(t)$ that the microbe wants to maximize as fast as possible. Once the available nutrients are consumed, the microbe rests. Since nature has been running its own optimization algorithm for millions of years, we expect the microbe to find the optimal solution.

$$\begin{aligned}
\min_{u(t),y(t)} \int_0^1 & -e^{-t} \cdot y_2(t)\,\mathrm{dt} \\
\text{s.t. } \dot{y}_1 &= -u_1 \\
\dot{y}_2 &= u_2 \\
0 &\le u_1, u_2 \le 10 \\
0 &\le y_1, y_2 \\
0 &= u_1 - \alpha \cdot u_2 \\
0 &\le y_2 - \frac{1}{k_1} \cdot u_1 + \frac{1}{k_1} \cdot u_2 \\
y_0(0), y_1(0) &= 1
\end{aligned} \tag{6.3}$$

The optimal solution would be for the microbe to eat all the available nutrients as fast as possible. This is also what we see in the following graphs produced with $\alpha = 5$, $k_1 = 2$, $k_2 = 8$.





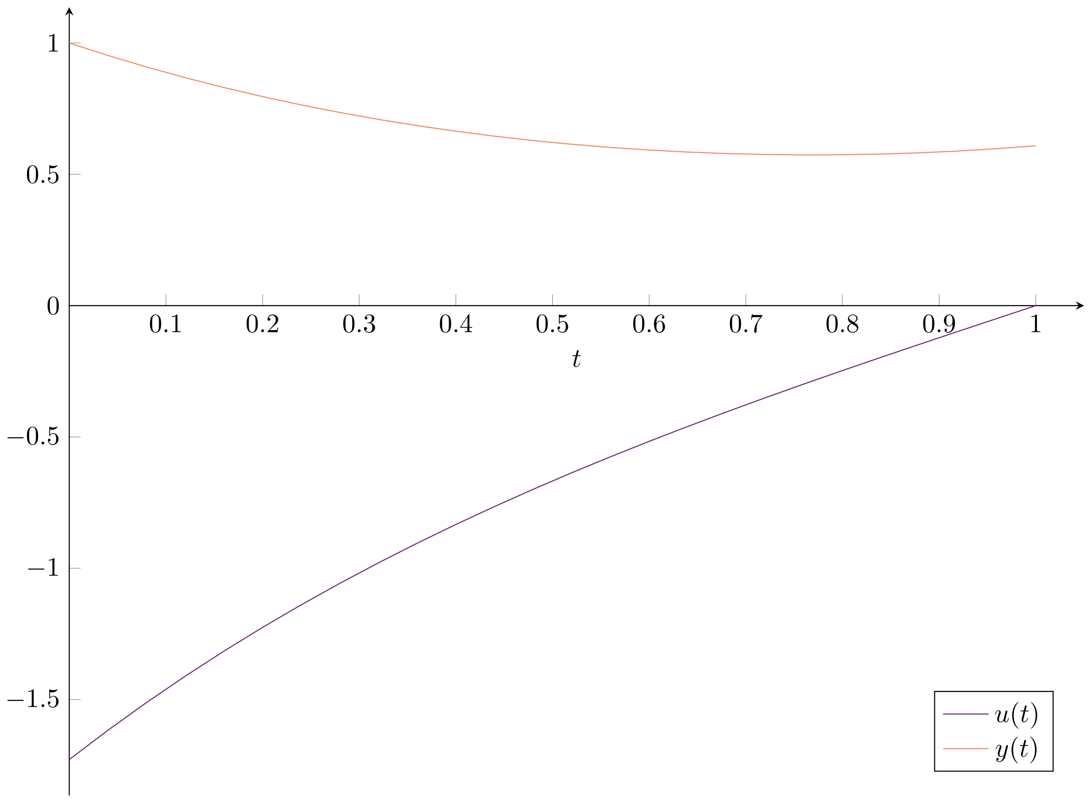
6.4 Example 4

Though we focus on linear problems, CPLEX is also capable of quadratic programming. For this example, the interface was modified to handle problems where the objective is quadratic, more specifically the integral of $\mathbf{\Phi}_1^\top \cdot \mathbf{y}(t)^2 + \mathbf{\Phi}_2^\top \cdot \mathbf{u}(t)^2$. Nothing else was changed. The modified code is available in a separate branch in the git repo [1].

$$\begin{aligned} \min_{u(t),y(t)} \int_0^1 y(t)^2 + \frac{1}{2}u(t)^2 \,\mathrm{dt} \\ \text{s.t. } \dot{y} = \frac{1}{2}y(t) + u(t) \\ y(0) = 1 \end{aligned} \tag{6.4}$$

A problem with a known analytical solution was chosen so that we may compare the program against a verified solution. We know from Gerdts [2], page 147, that the exact solution is (6.5).

$$y(t) = \frac{2\exp(3t) + \exp(3)}{\exp(3t/2)(2 + \exp(3))}, \quad u(t) = \frac{2(\exp(3t) - \exp(3))}{\exp(3t/2)(2 + \exp(3))} \tag{6.5}$$



Chapter 7

Results

Having introduced the solver and some example problems, we are now ready to use the examples in a series of tests. We will consider two main areas: accuracy and efficiency.

A selection of four different Runge-Kutta methods were chosen for all the tests: Euler's method, implicit Euler's method, Heun's second order method, and the classic fourth order method, denoted RK4. These methods were chosen to represent a broad selection of Runge-Kutta methods, as we have both explicit and implicit methods, as well as first, second and fourth order methods.

0	0
	1

Table 7.1: Euler's Method

1	1
	1

Table 7.2: Implicit Euler's Method

0		
1	1	
	1/2	1/2

Table 7.3: Heun's Method

0				
1/2	1/2			
1/2	0	1/2		
1	0	0	1	
	1/6	1/3	1/3	1/6

Table 7.4: RK4

The tests investigate accuracy and performance as a function of the one-step method used, as well as the number of parameters used in the optimization step. Since the number of parameters is determined by the number of steps N in the one-step method, where y has N parameters and u has $N-1$ parameters for a total of $2N-1$ parameters per dimension. Example 3 is the only two-dimensional problem, and as such we will cut the maximum number of steps in half to ensure we are testing with an equal number of parameters. The tests were all performed as close to, without exceeding, 1000 parameters as possible.[1]

Furthermore, the first example is so simple that there are essentially no interesting results to show. In fact, example 1 is just a special case of example 2, in particular when $\alpha = \beta = 0$. We shall therefore refrain from any further discussions of example 1.

[1]This means 500 steps or 999 parameters for the one-dimensional cases, and 250 steps or 998 parameters for the two-dimensional case.

7.1 Accuracy

We know how the choice of one-step method generally impacts the accuracy of an ODE, and want to test how this translates to our solver. Measuring the accuracy of example 4 is trivial, since we know the analytical solutions. For examples 2 and 3, however, we will assume the solver is correct at low enough step sizes and calculate a very high resolution solution to compare against.

Instead of comparing the local truncation error, we will measure the difference in the final objective value, which is more an indication of the global truncation error. I felt this was a more interesting comparison since the goal of optimal control is to minimize the objective, so a small difference in the evaluated objective indicates the found solutions are almost equally good, even if the output is very different.

7.1.1 Example 2

As the number of steps grows, the absolute difference between the one-step methods grows very small. Since we are more concerned with the order of the error, the errors are plotted on a logarithmic scale.

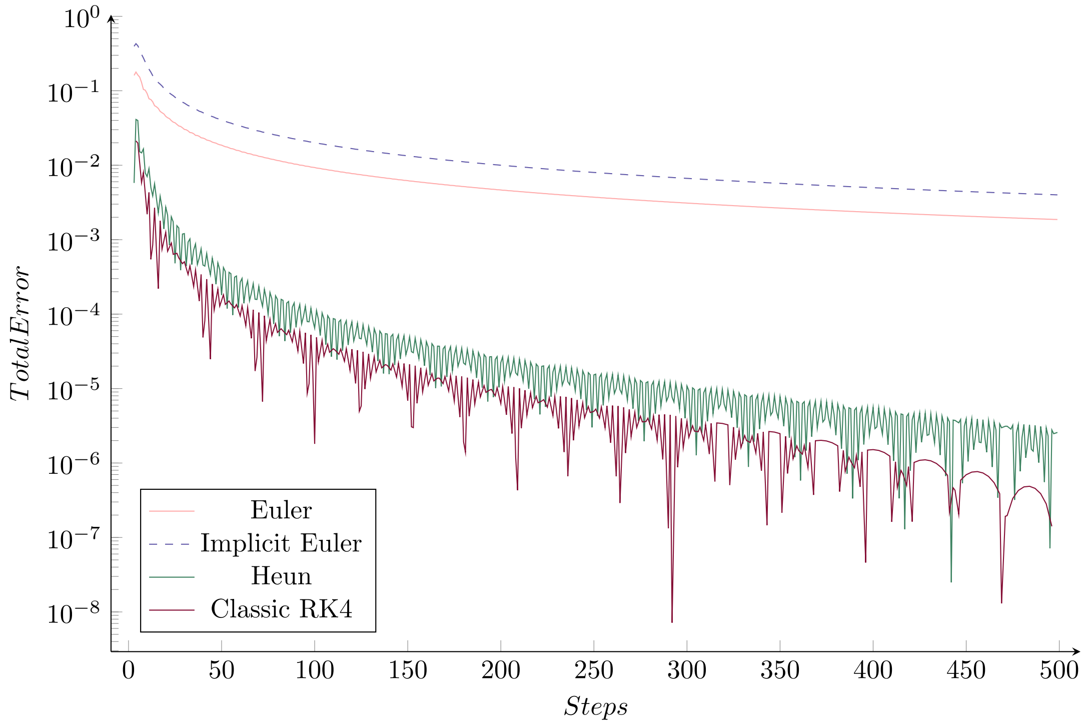


Figure 7.1: Total Error in Example 2

Unsurprisingly, the higher order methods produced better approximations, as is the case with regular ODEs. The variation in the total error fluctuated greatly between number of steps, for instance the error was nearly twice as much when using 300 steps when compared to using only 290 steps, but the graph shows clearly that all of the methods tested have an upper bound on the total error that decreases as the number of steps increase. However, this error probably does not converge to 0.

The difference between the higher order methods was smaller than I expected, with the second order method occasionally, but very rarely, outperforming the fourth order method. Still, its worth noting that the variation is only large in relative terms, and very small in absolute terms. At the same time, the difference between the first order methods and the higher order methods was larger than I anticipated, especially as the number of steps grew. Even at 250 steps the first order methods could not produce results as accurate as only 10 steps of the second order method, as shown in the below table.

Method	**10 steps**	**250 steps**
Euler's	9.1889123e-02	9.3130914e-03
Backward Euler	2.1038588e-01	2.0024691e-02
Heun's 2nd	7.0283198e-03	1.5279249e-05
Classic RK4	2.1967056e-03	4.9357457e-06

Table 7.5: Objective Error at 10 and 250 steps

7.1.2 Example 3

The choice of one-step method had no impact on example 3; the accuracy was only a function of the number of steps used. The accuracy still seems to be proportional to the number of steps, decreasing as the step-size decreases. This is likely a consequence of the state $y(t)$ being linear, meaning that Euler's method already is as accurate as possible at any given step-size.

Steps	**Error (Example 3)**
10	0.045359104
20	0.021163834
30	0.013364563
40	0.0095477565
50	0.0072515475
60	0.0057245166
70	0.0046481000
80	0.0038354421
90	0.0032026838
100	0.0027027758
110	0.0022906928
120	0.0019463652
130	0.0016587084
140	0.0014102166
150	0.0011940376
160	0.0010073758
170	0.00084136088
180	0.00069309382
190	0.00056226215
200	0.00044356512

Figure 7.2: Objective Error for all One-Step Methods in Example 3

7.1.3 Example 4

In the previous example, we assumed the objective approached the smallest possible value as the number of steps used in the state discretization increased. While we can verify visually that the approximations produced by the solver seem correct, we are yet to test the error of the high resolution approximation. This changes with example 4, since we know the analytic solution to this problem.

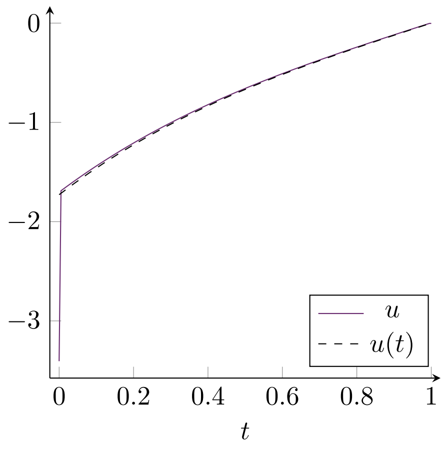


Figure 7.3: Control Approximation (500 steps)

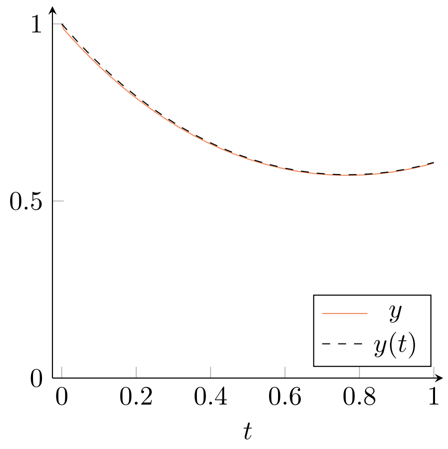


Figure 7.4: State Approximation (500 steps)

The approximations are mostly correct, with the notable exception of u_0 which exhibits a strange behaviour where the error grows with the number of steps. The error seems to converge to 1.728328996, which happens to be the same magnitude as the true value $|u(0)|$. In other words, it seems the limit $\lim_{n\to\infty} u_0 = 2 \cdot u(0)$. This happens regardless of what one-step method is used. The following graphs show this behaviour by measuring the error in max-norm, which for u always was $|u_0 - u(0)|$.

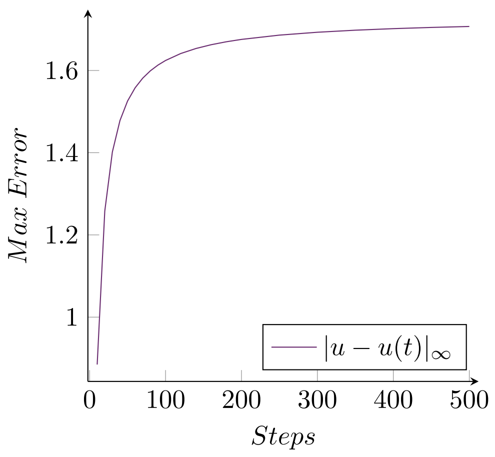


Figure 7.5: Control Error in Max Norm

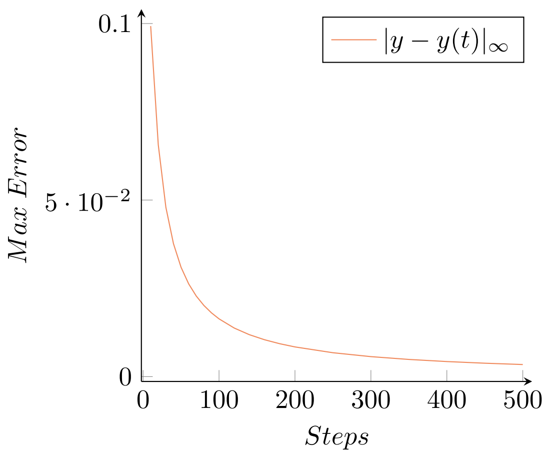


Figure 7.6: State Error in Max Norm

Unsure of the cause of this behaviour, I ran some more tests, starting by simply taking a closer look at approximations produced at high step-sizes. This did, however, not yield any results.

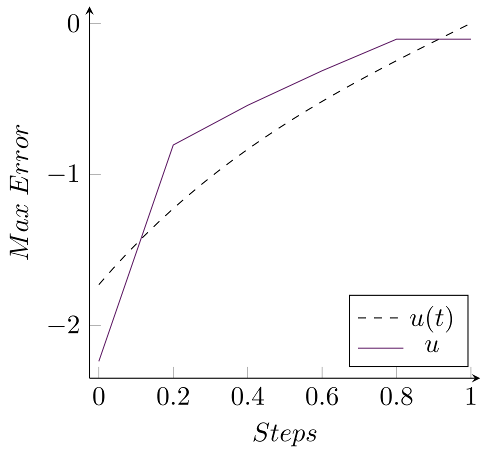


Figure 7.7: Control Approximation (6 steps)

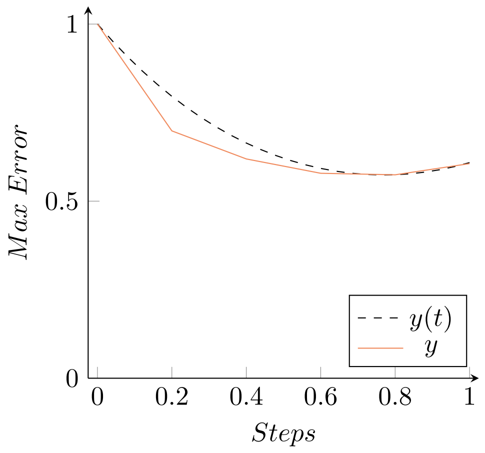


Figure 7.8: State Approximation (6 steps)

All I found was that the unexpected behaviour goes away if we add the boundary condition that $u_0 = u(0) \approx -1.728328996$. This also causes the max error of y to decrease as we get a better approximation of the trajectory at $u(0)$, limiting the undershoot shown in Figure 7.8.

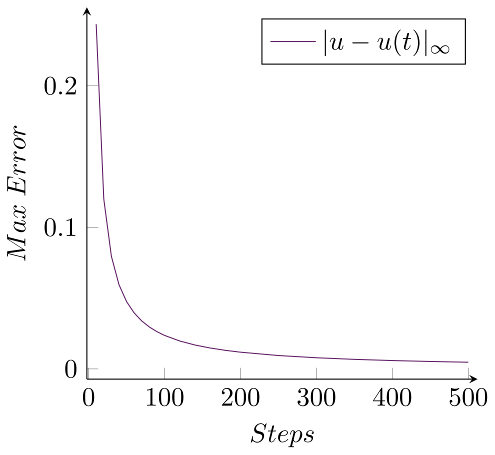


Figure 7.9: Control Error in Max Norm (Adjusted u_0)

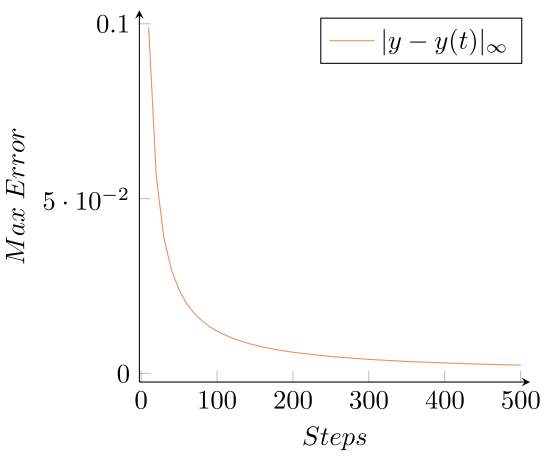


Figure 7.10: State Error in Max Norm (Adjusted u_0)

At this point, the extra tests have proved inconclusive in determining whether the behaviour exhibited by example 4 is caused by a bug in the program, or is inherent in the algorithm used by CPLEX to solve quadratic problems. Still, the rest of the approximations are fairly accurate at all step sizes, and the large error of u_0 seems to have a negligible effect on the rest of the approximation when the step-size is very small, which boosts the confidence in the correctness of the solver in the linear case.

7.2 Performance

As mentioned in Chapter 5, performance is extremely important for this application due to the iterative nature of the discretizations. Any inefficiency introduced at a single step could propagate into potentially millions of steps. This section explores how the chosen one-step methods compared with respect to the time spent discretizing the state ODE. Since the most computationally expensive steps are the state discretization and the optimization, we will also compare the run time of each method to that of CPLEX, as well as the other one-step methods.

The accuracy tests in the previous example are deterministic; they will always return the same result. This is not the case with the following timing tests, since the execution time of any program is dependent on the processing power available at run time. Modern home computers are not controlled environments in the sense that thousands of background tasks cause the available processing power to fluctuate between runs. We are more interested in trends, and will therefore plot only the tests where N is a multiple of 20. This helps smooth out the graphs and makes them easier to read.

7.2.1 Example 2

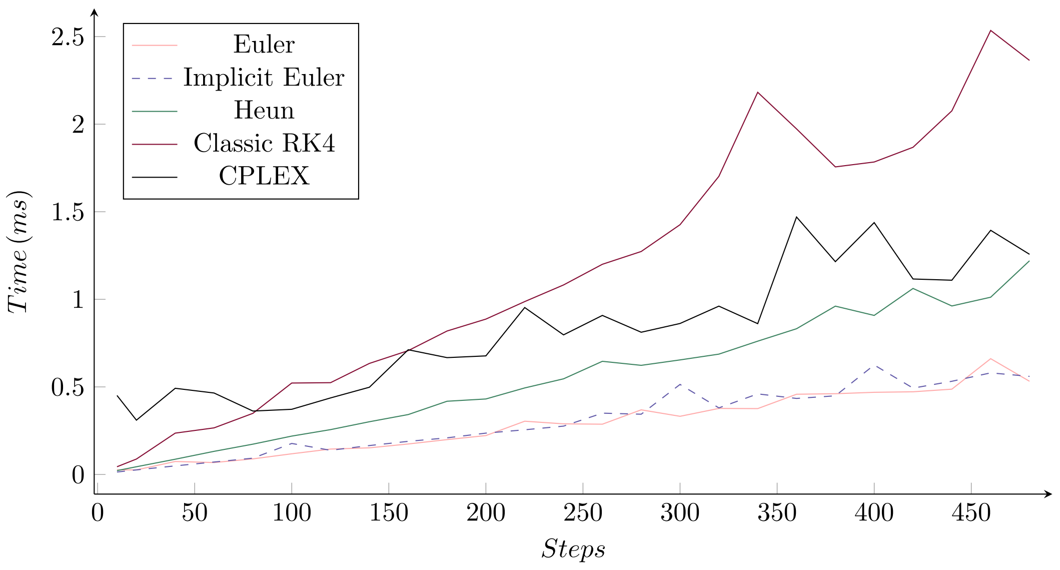


Figure 7.11: Time Spent Discretizing the State ODE and Optimizing for Example 2

My initial observation was that the performance of CPLEX, as well as all the one-step methods seem to be proportional to the number of steps. It makes sense that the Runge-Kutta methods would scale proportionally to the number of intermediate steps, which makes sense as their computational complexity is $\mathcal{O}(n \cdot s^s)$, meaning the fourth order method is about 4 times slower than the first order methods, and about twice as slow as the second order method. What was more interesting to see was that the growth rate of CPLEX is less than that of RK4, with RK4 overtaking CPLEX as the most expensive method in about 100 steps.

I observed that the time spent by any method could be almost twice as much when a test was run multiple times, making the full graphs entirely undecipherable. For that reason I chose to plot fewer tests, however, the graph for a full test suite is quite interesting when comparing a single one-step method to the optimization step used in the same test. The following graph shows all tests of the RK4 method, ran sequentially in order from $N = 3$ to $N = 499$.

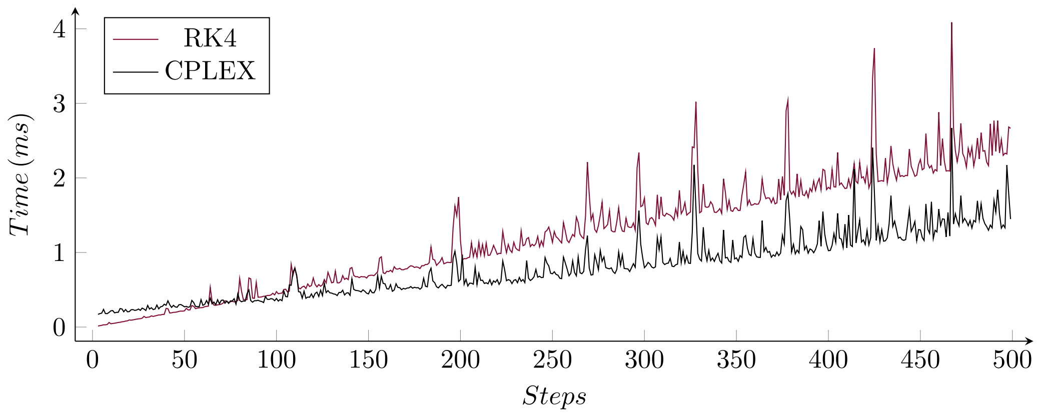
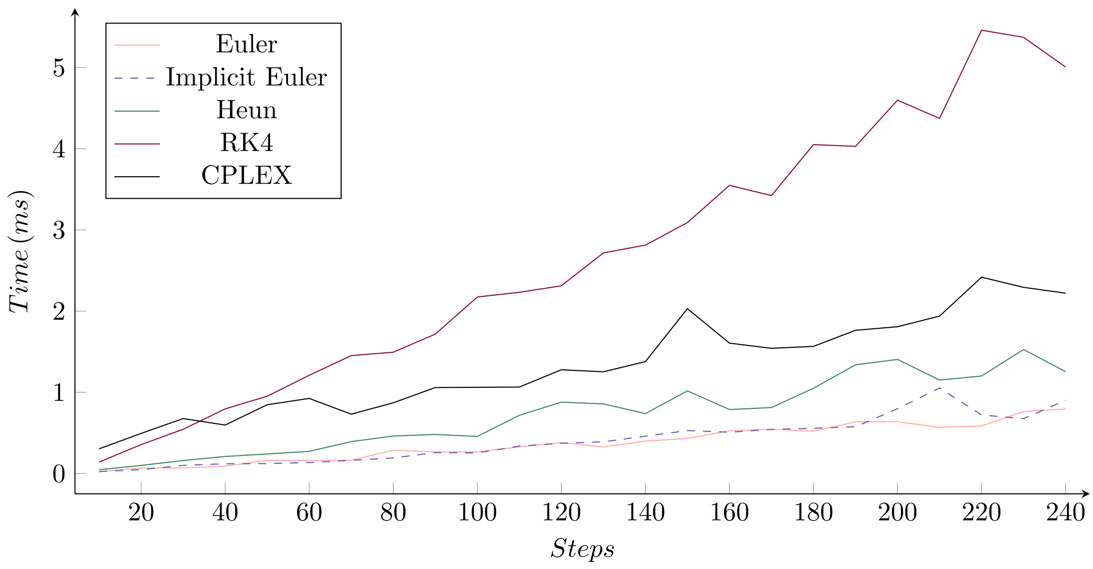


Figure 7.12: CPLEX compared to RK4 (same run)

The spikes in this graph that appear in both the one-step method and CPLEX during the same test, are almost certainly a consequence of random background tasks running on my computer. Perhaps the larger spikes that seem to appear at a regular interval are caused by a single, very intensive task. What is more interesting to note, though, is how clearly we can see that both CPLEX and the one-step method is bounded from below by some linear function.

7.2.2 Example 3



Recall that the number of steps is cut in half for this example, to retain the maximum number of parameters used. With that in mind, we once again see that both the one-step methods and CPLEX grows linearly with the number of parameters, only at a slightly higher rate than before; the higher rate likely caused by the fact that we are now using 2×2 matrices instead of scalars. When combined with the observations from example 2, this starts to paint a picture that the execution time of the full program seems to be linear for linear problems.

7.2.3 Example 4

I have mentioned in Chapter 3 that linear programming is very desirable, in part because of the sheer efficiency of the long-established algorithms used for this class of problems. The following is a great example of this. Since example 4 is quadratic in the objective, and thus cannot be solved with LP, we expect an increase in the time spent by the optimizer.

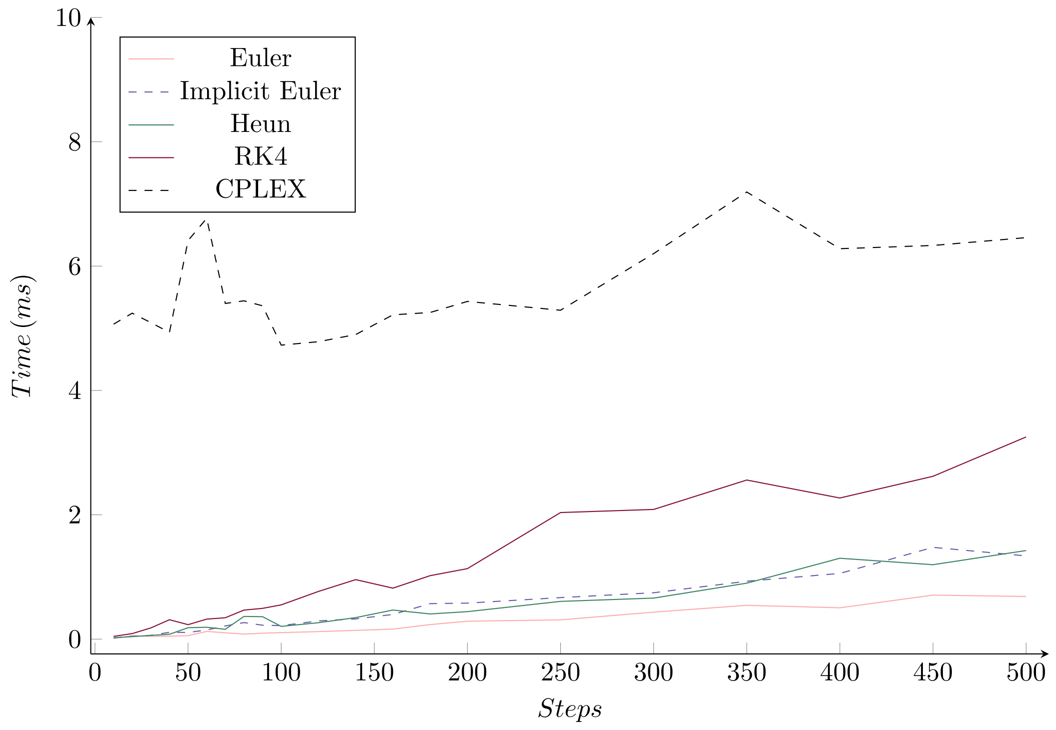


Figure 7.13: Execution Time of One-Step Methods and CPLEX for Example 4

I observe that the one-step methods behave as they did in earlier tests, while CPLEX performs much worse, as expected. Another observation is that CPLEX still seems to scale linearly with the number of parameters.

Chapter 8

Summary

The goal of this thesis was to develop an optimal control solver for linear problems, using complete parameterization. This has largely been a success, with the solver being able to produce accurate solutions in a reasonable time frame, even standing its ground when pushed outside the scope of its design, such as by using a quadratic objective function.

In the process, we've seen that the performance of our linear problems were $\mathcal{O}(n)$, where n is the number of parameters used. Assuming this trend continues as the number of steps increase, we should be able to solve linear problems with a million parameters using RK4 in less than 10 seconds. That being said, there is still much room for improvement when it comes to performance, both from a mathematical perspective as well as a computer programming perspective.

8.1 Future Work

Chapter 5 mentioned some possible code improvements, such as SIMD, that should improve the performance of the program. Another programming-related improvement might be multi-threading the one-step method. While one-step methods are not traditionally associated with parallel computing, since each step is dependent on the value of the previous one, we are not limited in this way with parameterization, since the steps are not evaluated but instead treated as constraints for the discrete optimization problem.

We've also seen that execution time of the program as a whole is proportional to the number of parameters. By not parameterizing the state variables at all, CVP could halve the number of parameters used. This would come at the cost of robustness and accuracy at similar step-sizes, but it remains to be seen if this speed makes up the loss of accuracy, and it would be interesting to test first hand which approach can achieve the greatest accuracy in a given time-frame.

Many commercial Runge-Kutta methods also implement automatic step selection, where the step size is adjusted based on an estimate of the local truncation error. Such methods might significantly decrease the number of points in the grid $\mathbb{G}_N$, as we know is the case for regular ODEs.

While linear problems are powerful in their own right, most real world processes are best modelled by non-linear systems. After seeing the performance impact of changing the objective from linear to quadratic, we might consider linearization as the best approach to NLP. In that case, some linearization method could be embedded into the solver as a translation layer from NLP to LP, and no changes to the current code base would be necessary.

For the time being, the solver is coupled to the visualization software made specifically for this thesis. The solver was designed as a library, meaning it is not dependent on any part of the visualization software. Accordingly, decoupling the solver and turning into a standalone library should be a trivial task for an experienced C++/C developer. The widespread popularity of Python also makes a Python binding an attractive option for this solver.

Bibliography

[1] Imre Angelo. *Linear Optimal Control*. https://github.com/ImredeAngelo/LinearOptimalControl. git repository. 2023. (Visited on 31st May 2023).

[2] Matthias Gerdts. *Optimal Control of Ordinary Differential Equations and Differential-Algebraic Equations*. 2006.

[3] Ernst Hairer, Syvert Paul Nørsett and Gerhard Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. 2nd ed. Springer, 1993. DOI: 10.1007/978-3-540-78862-1.

[4] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. 2nd ed. Springer, 2006. DOI: 10.1007/978-0-387-40065-5.