University of Texas Rio Grande Valley

ScholarWorks @ UTRGV

Theses and Dissertations

12-2023

An Automatic Solver for Optimal Control Problems

Marcel Efren Benitez
The University of Texas Rio Grande Valley

Follow this and additional works at: https://scholarworks.utrgv.edu/etd

Part of the Mathematics Commons

Recommended Citation

Benitez, Marcel Efren, "An Automatic Solver for Optimal Control Problems" (2023). *Theses and Dissertations*. 1471.
https://scholarworks.utrgv.edu/etd/1471

This Thesis is brought to you for free and open access by ScholarWorks @ UTRGV. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of ScholarWorks @ UTRGV. For more information, please contact justin.white@utrgv.edu, william.flores01@utrgv.edu.

AN AUTOMATIC SOLVER FOR OPTIMAL CONTROL PROBLEMS

A Thesis

by

MARCEL E. BENITEZ

Submitted in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: Mathematics

The University of Texas Rio Grande Valley

December 2023

AN AUTOMATIC SOLVER FOR OPTIMAL CONTROL PROBLEMS

A Thesis
by
MARCEL E. BENITEZ

COMMITTEE MEMBERS

Dr. Josef Sifuentes
Chair of Committee

Dr. Cristina Villabobos
Committee Member

Dr. Andras Balogh
Committee Member

Dr. Mrinal Roychowdhury
Committee Member

December 2023

Copyright 2023 Marcel E. Benitez

All Rights Reserved

ABSTRACT

Benitez, Marcel E., An Automatic Solver for Optimal Control Problems. Master of Science (MS), December, 2023, 82 pp., 18 figures, 7 references.

Optimal control theory is a study that is used to find a control for a dynamical system over a period of time such that a objection function is optimized. In this study we will be looking at optimal control problems for ordinary differential equations or ODEs and see that we can use an automatic solver using the forward-backward sweep using Matlab to solve for them from an 1 dimension to bounded cases and to nth dimension cases.

DEDICATION

I would like to dedicate this to my parents, Miguel A. Benitez Jr., Leonilla Benitez, and to my two siblings, my late brother Andree C. Benitez and my sister Aimee E. Benitez. All my success is thanks to the constant support of my family. My motivation was thanks to them as well for pushing me to be the first in my family to receive a master's degree. They have blessed me for giving me an continuous amount of support, and for that I am forever grateful.

ACKNOWLEDGMENTS

I will like to acknowledge my family for their constant support throughout my entire academia career. To both of my parents for pushing me to pursue my masters in the beginning of the 2020 when the peak of the pandemic was at its highest. As well as accommodating to my needs throughout my graduate career.

I would like to send my deepest and most sinecure gratitude to my chairman committee and advisor to Dr. Josef Sifuentes. He introduced me to Matlab coding as an undergraduate student which got me interested on it possibilities of coding that can be done. He would later on introduce me in a numerical analysis course as graduate student the theory of optimal control, with the idea of an automatic solver for optimal control problems. As well being supportive and empathetic during some personal struggles and providing excellent guidance in the research field and as well in the academic field.

TABLE OF CONTENTS

Page

ABSTRACT . . . iii
DEDICATION . . . iv
ACKNOWLEDGMENTS . . . v
TABLE OF CONTENTS . . . vi
LIST OF FIGURES . . . ix
CHAPTER I. INTRODUCTION . . . 1
1.1 Basic problem and necessary conditions . . . 1
1.2 The Hamiltonian . . . 3
1.3 Optimization . . . 7
1.4 Payoff Terms . . . 8
CHAPTER II. FORWARD-BACKWARD SWEEP METHOD: ONE DIMENSION . . . 10
2.1 Forward-Backward Sweep . . . 10
2.2 One Dimensional Maximum Optimal Control Problem . . . 13
2.2.1 Book version . . . 14
2.2.2 What We Have Constructed . . . 17
2.3 Bounded Case Optimal Control Problem . . . 23
2.3.1 Necessary Conditions . . . 24
2.3.2 What We Have Constructed . . . 27
CHAPTER III. FORWARD-BACKWARD SWEEP METHOD: MULTI DIMENSIONAL . . . 30
3.1 Optimal Control with several variables . . . 30
3.2 Multi-Dimensional Theory . . . 31
3.3 Multi-dimensional optimal control problem one . . . 31
3.4 Multi-dimensional optimal control problem two . . . 32
CHAPTER IV. MULTI DIMENSIONAL SWEEP . . . 34
4.1 Multi Dimensional . . . 34
4.2 Multi Dimensinal Greedy . . . 41
4.3 Subcodes . . . 45

4.3.1 Geval 45
4.3.2 Xeval 45
4.3.3 Fwrap 46
4.3.4 Leval 46
4.3.5 lwrap 47
4.3.6 Ueval 48
4.3.7 Trapmeathod 48
CHAPTER V. RESULTS 50
5.1 Max 1-dimensional case 50
5.2 Bounded case 51
5.3 Multi-Dimensional case 52
5.3.1 Regular 52
5.3.2 Greedy 53
5.4 Multi-Dimensional case HIV Treatment 55
5.4.1 Regular 55
5.4.2 Greedy 56
CHAPTER VI. CONCLUSION 58
REFERENCES 60
APPENDIX A 61
1.1 The Lebesgue Dominated Convergence Theorem 62
1.2 One Dimensional 63
1.2.1 Maximum One Dimensional Code 63
1.2.2 Bounded Case Code 65
1.3 Multi-Dimensional 68
1.3.1 Multi-Dimensional Code 68
1.3.2 Multi-Dimensional Greedy Code 71
1.4 Sub-Codes 74
1.4.1 Forward-Backward Sweep Code 74
1.4.2 Transverse Condition Codes 75
1.4.3 G Evaluation Code 77
1.4.4 X Evaluation Code 77
1.4.5 F Wrapper Code 77

1.4.6 L Evaluation Code 78
1.4.7 L Wrapper Code 79
1.4.8 U Evaluation Code 79
1.4.9 U Wrapper Code 80
1.4.10 Trapezoidal Method Code 81
BIOGRAPHICAL SKETCH 82

LIST OF FIGURES

Page

Figure 5.1: Time Vs X 51
Figure 5.2: Time vs Lambda 51
Figure 5.3: Time vs U 51
Figure 5.4: Time Vs X 52
Figure 5.5: Time vs Lambda 52
Figure 5.6: Time vs U 52
Figure 5.7: Time Vs X 53
Figure 5.8: Time vs Lambda 53
Figure 5.9: Time vs U 53
Figure 5.10: Time Vs Lambda 54
Figure 5.11: Time vs X 54
Figure 5.12: Time vs U 54
Figure 5.13: Time Vs X 56
Figure 5.14: Time vs Lambda 56
Figure 5.15: Time vs U 56
Figure 5.16: Time Vs X 57
Figure 5.17: Time vs Lambda 57
Figure 5.18: Time vs U 57

CHAPTER I

INTRODUCTION

Optimal control theory is a branch in mathematics that computes optimal ways to control a dynamic system. It is used in science, engineering and operations research. An example of an application is sending a rocket to the moon with minimal fuel consumption; An ordinary differential equation describes the state of the scenario being optimized and is depended on a time-dependent control parameter An example will be calculating the movement or flow of electricity or the motion of an object subject to external and internal forces.

In this paper we will be looking at Optimal control in ordinary differential equations, and using Matlab to create an automatic solver to solve for one-dimensional, and multi-dimensional cases, we will also consider cases where the control is bounded. Using the forward-backward sweep method from a textbook called Optimal Control Applied to Biological Models by Suzanne Lenhart [6] as a reference, we will create automatic solvers using the symbolic toolbox of Matlab. We will test our algorithm with examples from the textbook and see how well the solvers do. But before we go over the forward-backward sweep method, let us go over the basic problem, its necessary conditions, the Hamiltonian and a few theories about optimization.

1.1 Basic problem and necessary conditions

In an optimal control problem in ordinary differential equations, $u(t)$ is the control and $x(t)$ is the state. In this case the state variable satisfies the differential equation that depends on the control variable:

$$x' = g(t, x(t,), u(t)). \tag{1.1}$$

Now as the control function changes, the solution to the differential equation will change as well. We then can see the control-to-state relationship as a map as $u(t) \mapsto x = x(u)$, as well x is a function to the independent variable t, but we write $x(u)$ as a reminder that x depends on u. Now our basic optimal control problem will be consisting on finding a piecwise continuous control $u(t)$ and the associated state variable $x(t)$ in order to maximize the given objective function, to which it will be the following:

$$\max_u \int_{t_0}^{t_f} f(t, x(t), u(t))\, dt$$

which is subject to

$$\begin{aligned} x'(t) &= g(t, x(t), u(t)) \qquad (1.2) \\ x(t_0) &= x_0 \end{aligned}$$

By maximizing the control this is called an optimal control. Note that $x(t_f)$ is free it means that the value of $x(t_f)$ is unrestricted. In this study f and g will always be continuously differentiable functions in all three arguments. Hence the control(s) will always be piecewise continuous, and the associated states will always be piecewise differentiable as noted by Suzanne Lenhart.[6]

In this study we found that a principle technique for such an optimal control problem is to solve a set of "necessary conditions" that the optimal control and the corresponding state need to satisfy. We must clarify that it important to understand the logical difference of necessary conditions and sufficient conditions of solution sets.

Necessary Conditions: If $u^*(t)$, $x^*(t)$ are optimal, then the following conditions... hold

Sufficient Conditions: if $u^*(t)$, $x^*(t)$ satisfy the following conditions..., then $u^*(t)$,$x^*(t)$ are optimal. In the terms of our control

$$J(u) = \int_{t_0}^{t_f} f(t, x(t), u(t))dt$$

1.2 The Hamiltonian

Now lets say a piecewise continuous optimal control exists, and is given by u^* with x^* the corresponding state solution. In particular $J(u) \leq J(u^*) < \infty$ for all controls u^*. Letting $h(t)$ be a continuous variation and $\varepsilon \in \mathbb{R}$. Then

$$u^{\varepsilon}(t) = u^*(t) + \varepsilon h(t)$$

is another piecewiese continuous control.

Then let x^{ε} be the state corresponding to control u^{ε}, namely x^* satisfies the following equation

$$\frac{d}{dx} x^{\varepsilon}(t) = g(t, x^{\varepsilon}(t), u^{\varepsilon}(t)) \tag{1.3}$$

for when u^* is continuous; And all trajectories will start in the same position, so take $x^{\varepsilon}(t_0) = x_0$.

It can easily be seen that $u^{\varepsilon}(t) \to u^*(t)$ for all t as $\varepsilon \to 0$. For all t

$$\frac{\partial u^{\varepsilon}(t)}{\partial \varepsilon}|_{\varepsilon=0} = h(t).$$

This is also similar to x^{ε}. and based on the assumptions on g we get the following

$$x^{\varepsilon}(t) \to x^*(t)$$

for each fixed t. Furthermore its derivative

$$\frac{\partial}{\partial x} x^{\varepsilon}(t)|_{\varepsilon=0},$$

for each t, exists; In this case we will only need to know that it exists not the actual value of quantity.

So the objective function at u^{ε} will be

$$J(u^{\varepsilon}) = \int_{t_0}^{t_f} f(t, x^{\varepsilon}(t), u^{\varepsilon}(t)) dt.$$

Let us now begin to see the adjoint function or variable λ. If we let $\lambda(t)$ be a piecewise differentiable function on $[t_0,\ t_f]$ be determined. Then by the Fundamental Theorem of Calculus gives the following

$$\int_{t_0}^{t_f} \frac{d}{dt}[\lambda(t)x^{\varepsilon}(t)]dt = \lambda(t_1)x^{\varepsilon}(t_1) - \lambda(t_0)x^{\varepsilon}(t_0),$$

implying

$$\int_{t_0}^{t_f} \frac{d}{dt}[\lambda(t)x^{\varepsilon}(t)]dt + \lambda(t_0)x_0 - \lambda(t_1)x^{\varepsilon}(t_1) = 0.$$

Then adding 0 to the expression to out $J(u^{\varepsilon})$ gives

$$\begin{aligned}
J(u^{\varepsilon}) &= \int_{t_0}^{t_f} [f(t, x^{\varepsilon}(t), u^{\varepsilon}(t)) + \frac{d}{dt}(\lambda(t)x^{\varepsilon}(t)]dt \\
&\quad + \lambda(t_0)x_0 - \lambda(t_1)x^{\varepsilon}(t_1) \\
&= \int_{t_0}^{t_f} [f(t, x^{\varepsilon}(t), u^{\varepsilon}(t)) + \lambda'(t)x^{\varepsilon}(t) + \lambda(t)g(t, x^{\varepsilon}(t), u^{\varepsilon}(t))]dt \\
&\quad + \lambda(t_0)x_0 - \lambda(t_1)x^{\varepsilon}(t_1),
\end{aligned}$$

then after use of the product rule and knowing that $g(t, x^{\varepsilon}, u^{\varepsilon}) = \frac{d}{dt}x^{\varepsilon}$ for all but finitely many points. The maximum of J_1 with respect to the control u_1 occurs at u^* then the derivative of $J(u^{\varepsilon})_1$ with respect to ε, in the direction of h, is zero. i.e

$$0 = \frac{d}{d\varepsilon}J(u^{\varepsilon})|_{\varepsilon=0} = \lim_{\varepsilon\to\infty} \frac{J(u^{\varepsilon}) - J(u^*)}{\varepsilon}.$$

Which gives the limit of an integral expression. With another version of the Lebesgue Dominated Convergence Theorem Given by Halsey Royden [4],this will be discussed in the appendix. This theorem will let us move the limit and its derivative inside the integral. This is because of the compact interval of integration and the piecewise differentiable of the intergrand. Thus we have the following

$$
\begin{aligned}
0 &= \frac{d}{d\varepsilon} J(u^{\varepsilon})|_{\varepsilon=0} \\
&= \int_{t_0}^{t_f} \frac{\partial}{\partial \varepsilon}[f(t,x^{\varepsilon}(t),u^{\varepsilon}(t)) + \lambda'(t)x^{\varepsilon}(t) + \lambda(t)g(t,x^{\varepsilon}(t),u^{\varepsilon}(t)dt]|_{\varepsilon=0} \\
& -\frac{\partial}{\partial \varepsilon}\lambda(t_1)x^{\varepsilon}(t_1)|_{\varepsilon=0}.
\end{aligned}
$$

Then after applying the chain rule to f and g , we then get

$$
\begin{aligned}
0 &= \int_{t_0}^{t_f}[f_x \frac{\partial x^{\varepsilon}}{\partial \varepsilon} + f_u \frac{\partial u^{\varepsilon}}{\partial \varepsilon} + \lambda'(t)\frac{\partial x^{\varepsilon}}{\partial \varepsilon} + \lambda(t)(g_x \frac{\partial x^{\varepsilon}}{\partial \varepsilon} + g_u \frac{\partial u^{\varepsilon}}{\partial \varepsilon})]|_{\varepsilon=0}\, dt \qquad (1.4) \\
& -\lambda(t_1)\frac{\partial x^{\varepsilon}}{\partial \varepsilon}(t_1)|_{\varepsilon=0}
\end{aligned}
$$

where the argument of the $f_x,\ f_u,\ g_x$ and g_u terms are dependent on $t, x^*(t), u^*(t)$. Then rearranging terms in 1.4, we have the following

$$
\begin{aligned}
0 &= \int_{t_0}^{t_f}[(f_x + \lambda(t)g_x + \lambda'(t))\frac{\partial x^{\varepsilon}}{\partial \varepsilon}(t)|_{\varepsilon=0} + (f_u + \lambda(t)g_u)h(t)]\, dt \qquad (1.5) \\
& -\lambda(t_1)\frac{\partial x^{\varepsilon}}{\partial \varepsilon}(t_1)|_{\varepsilon=0}
\end{aligned}
$$

By choosing the adjoint function to simplify 1.5. The coefficients of

$$
\frac{\partial x^{\varepsilon}}{\partial \varepsilon}(t)|_{\varepsilon=0}
$$

will then cancel out, followed by choosing the adjoint function $\lambda(t)$. We then obtain the adjoint equation

$$
\lambda'(t) = -[f_x(t,x^*(t),u^*(t)) + \lambda(t)g_x(t,x^*(t),u^*(t))]
$$

and the boundary condition (or transversality condition)

$$
\lambda(t_1) = 0.
$$

Now 1.5 reduces to

$$0 = \int_{t_0}^{t_f} (f_u(t,x^*(t),u^*(t)) + \lambda(t)g_u(t,x^*(t),u^*(t)))\ h(t)\ dt.$$

And since this holds for any piecewise continuous variation function $h(t)$, then it holds for

$$h(t) = f_u(t,x^*(t),u^*(t)) + \lambda(t)g_u(t,x^*(t),u^*(t)).$$

But for this case

$$0 = \int_{t_0}^{t_f} (f_u(t,x^*(t),u^*(t)) + \lambda(t)g_u(t,x^*(t),u^*(t)))^2\ dt$$

it implies the optimality condition

$$f_u(t,x^*(t),u^*(t)) + \lambda(t)g_u(t,x^*(t),u^*(t)) = 0 \text{ for all } t_0 \leq t \leq t_f.$$

With all these equations they form a set of necessary conditions that a optimal control and state must satisfy; But in application, it is not necessary to rederive the above equations in this way for a certain problem. In fact, all these conditions from the Hamiltonian H which is defined as follows

$$H(t,x,u,\lambda) = f(t,x,u) + \lambda g(t,x,u)$$

With this we are then maximizing H with respect to u at u^*; And all the above conditions can be written as terms for the Hamiltonian:

$$\begin{aligned}
\frac{\partial H}{\partial u} &= 0 \text{ at } \Rightarrow f_u + \lambda g_u = 0 \text{ (Optimality condition)}, \\
\lambda' &= -\frac{\partial H}{\partial x} \Rightarrow \lambda' = -(f_x + \lambda g_x) \text{ (Adjoint equation)}, \\
\lambda(t_1) &= 0 \text{ (Transversality condition)}.
\end{aligned}$$

Now we obtained the dynamics of the state equation as it :

$$x' = g(t,x,u) = \frac{\partial H}{\partial \lambda},\ x(t_0) = x_0$$

1.3 Optimization

In optimal control theory, the most important result would be the the Principle of Optimality. This theory shows how the optimal control problem over sub-interval time notations of its original time span relates to the optimal control in full time. The following theory is the Principle of Optimality;

Let u^* be an optimal control, and let x^* be the resulting state, for the following problem

$$\begin{aligned} \max_u \quad & J(u) = \max_u \int_{t_0}^{t_f} f(t,x(t),u(t))dt \\ \text{subject to} \quad & x'(t) = g(t,x(t),u(t)),\ x(t_0) = x_0. \end{aligned} \tag{1.6}$$

Let $\hat{t}$ be a fixed point in time such that $t_0 < \hat{t} < t_f$.Then functions $\hat{u}^* = u^*|_{[\hat{t},t_f]}$, and $\hat{x}^* = x^*|_{[\hat{t},t_f]}$ will then form an optimal pair for the restricted problem

$$\begin{aligned} \max_u \quad & \hat{J}(u) = \max_u \int_{\hat{t}}^{t_f} f(t,x(t),u(t))dt \\ \text{subject to} \quad & x'(t) = g(t,x(t),u(t)),\ x(\hat{t}) = x^*(t). \end{aligned} \tag{1.7}$$

Lastly if u^* is a unique optimal control for 1.6, then $\hat{u}^*$ is the unique optimal control for 1.7. The following is the proof of the theorem.

This is done by contradiction. Now suppose, the contrary, that $\hat{u}^*$ is not optimal, meaning there exists a control $\hat{u}_1$ on the interval $[t_0,t_f]$ such that $\hat{J}(\hat{u}_1) > \hat{J}(\hat{u}^*)$. Which constructs a new

control u_1 on the interval $[t_0, t_f]$ as the following

$$u_1(t) = \begin{cases} u^* & \text{for } t_0 \leq t \leq \hat{t} \\ \hat{u}_1 & \text{for } \hat{t} < t < t_f. \end{cases}$$

Now if we let x_1 be the state associated with control u_1. Notice that u_1 and U^* both agree on $[t_0, \hat{t}]$, meaning that both x_1 and x^* will also agree there. Thus

$$\begin{aligned} J(u_1) - J(u^*) &= \left(\int_{t_0}^{\hat{t}} f(t, x_1, u_1)dt + \hat{J}(\hat{u}_1)\right) - \left(\int_{t_0}^{\hat{t}} f(t, x^*, u^*)dt + \hat{J}(\hat{u}^*)\right) \\ &= \hat{J}(\hat{u}_1) - \hat{J}(\hat{u}^*) \\ &> 0. \end{aligned}$$

Notice that this contradicts out initial u^* that was optimal for 1.6. Thus no control $\hat{u}_1$ exists, and $\hat{u}^*$ is optimal for 1.7

1.4 Payoff Terms

Most of the time, in addition to maximizing (or minimizing) terms over the time interval, We would like to maximize a function value at one particular point of time, most important, at the end of the time interval. For example lets say you want to minimize the tumor cells at a final time in a cancer model, or the number of infected individuals at a final time in an epidemic model. These conditions must be altered appropriately. Let us consider the following set-up

$$\begin{aligned} \max_u \quad & [\phi(x(t_f)) + \int_{t_0}^{t_f} f(t, x(t), u(t))dt] \\ \text{subject to} \quad & x'(t) = g(t, x(t), u(t)),\ x(t_0) = x_0, \end{aligned}$$

where $\phi(x(t_f))$ is the goal with respect to the final position or population level, $x(t_f)$. The function $\phi(x(t_f))$ is called the *payoff term*. This can also be referred as the salvage term. Considering the

resulting change on the derivation of the necessary conditions. The function then becomes

$$J(u) = \int_{t_0}^{t_f} f(t, x(t), u(t))dt + \phi(x(t_f)).$$

In the calculation of

$$0 = \lim_{\varepsilon \to 0} \frac{J(u^{\varepsilon}) - J(u^*)}{\varepsilon}$$

the changes only occur in the conditions of the final time

$$\begin{aligned} 0 = \int_{t_0}^{t_f} & [(f_x + \lambda g_x + \lambda') \frac{dx^{\varepsilon}}{d\varepsilon}|_{\varepsilon=0} + (f_u + \lambda g_u)\, h\,]\, dt \\ -(\lambda(t_f) - \phi'(x(t_f))) & \frac{\partial x^{\varepsilon}}{\partial \varepsilon}\, (t_f)\, |_{\varepsilon=0}. \end{aligned} \tag{1.8}$$

So, choosing an adjoint variable λ to satisfy the previous adjoint equation we get the following

$$\begin{aligned} &\lambda'(t) = -f_x(t, x^*, u^*) - \lambda(t) g_x(t, x^*, u^*), \\ &\lambda(t_f) = \phi(x^*(t_f)), \end{aligned}$$

then 1.9 is reduces to

$$0 = \int_{t_0}^{t_f} (f_u + \lambda g_u)\, h\, dt,$$

and its optimally condition is

$$f_u(t, x^*, u^*) + \lambda g_u(t, x^*, u^*) = 0$$

follows as before. the only change in it's necessary conditions is the transversality condition, as it follows

$$\lambda(t_f) = \phi'(x^*(t_f)).$$

CHAPTER II

FORWARD-BACKWARD SWEEP METHOD: ONE DIMENSION

2.1 Forward-Backward Sweep

We first consider the following optimal control problem

$$\begin{aligned}
\max_{u} \quad & \int_{t_0}^{t_f} f(t,x(t),u(t)\ dt\ +\ \phi(x(t_f)) \\
\text{subject to} \quad & x'(t) = g(t,x(t),u(t)), \quad x(t_0) = x_0.
\end{aligned}$$

In order to solve such problems numerically, We must devise an algorithm that can generate a approximation to a optimal piecewise continuous control u^*. First break a time interval $[t_0,t_f]$, where t_0 is the initial point and t_f is the end point, into equally spaced specific points of interest as such $t_0 = b_1, b_2, \cdots, b_N, b_{N+1} = t_f$; And the approximation vector will be $\overrightarrow{u} = (u_1, u_2, \cdots, u_{N+1})$, where $u_i \approx u(b_i)$.

As we discussed previously, any solution to the above optimal control problem, it must satisfy

$$\begin{aligned}
& x'(t) = g(t,x(t),u(t)), \quad x(t_0) = x_0, \\
& \lambda'(t) = -\frac{\partial H}{\partial x} = -(f_x(t,x,u) + \lambda(t) g_x(t,x,u)), \quad \lambda(t_f) = \nabla\phi(x(t_f)), \\
& 0 = \frac{\partial H}{\partial u} = f_u(t,x,u) + \lambda(t) g_u(t,x,u) \quad \text{at} \quad u^*.
\end{aligned}$$

The third equation, which is the optimality condition, can be manipulated in order to find a representation of u^* in terms of $t, x, \text{and}, \lambda$. If these terms are then substituted back into the ODEs for x, λ, then the two equations form its two-point boundary value problem.

We would then like to take certain characteristics of the optimality system; but first we are given a initial condition for state x and a final time condition for the adjoint λ. Second we take a function g of t, x, and, u only. Now values of λ are not needed to solve the differential equation for x using a standard ODE solver, in this case we will be using the Rung-Kutta. This method is known as the Forward-Backward Sweep method, this method is also very intuitive. We will be giving a rough outline of the algorithm below, according to Suzanne Lenhart[6]. With $\overrightarrow{x} = (x_1, \cdots, x_{N+1})and\overrightarrow{\lambda} = (\lambda_1, \cdots, \lambda_{N+1})$ being the vector approximations for the state and adjoint, here are the steps.

Step 1: Make an initial guess $\overrightarrow{u}$ over the interval.

Step 2: Using the initial condition $x(t_0) = x_0$ and the values for $\overrightarrow{u}$, solve $\overrightarrow{x}$ forward in time according to its differential equation in the optimality system.

Step 3: Using the transversality condition $\lambda_{N+1} = \lambda(t_f) = 0$ and the values for $\overrightarrow{u}$ and $\overrightarrow{x}$, solve λ backward in time according to its differential equation in the optimality system.

Step 4: Update $\overrightarrow{u}$ by entering the new $\overrightarrow{x}$ and $\overrightarrow{\lambda}$ values into the characterization of the optimal control.

Step 5: Check convergence. If values of the variables in this iteration and the last iteration are negligibly close, output the current values as solutions, If values are not close, return to step 2.

There are a few notes form the algorithm that we must note. First for the initial guess, $\overrightarrow{u} \equiv 0$, it is almost always sufficient; But in certain problems, where the division by u occurs for example a different initial guess must be used. And occasionally, the initial guess may require adjusting if the algorithm has problems converging. Which it is often in step 4 that it is necessary to use a convex combination between the previous control values and values given by the current characterization. This most often speeds the convergence. In so doing, this is done with the provided code from Suzanne Lenhart [6] and as well by the code that we have structured. As you will see in

step 2 and step 3, that any standard ODE solver can be used, but as we mentioned earlier we will be looking and the Rung-Kutta 4 ODE solver. Especially when a given a step size h and an ODE $x'(t) = f(t, x(t))$, and the approximation $x(t+h)$ given $x(t)$ we receive the following

$$x(t+h) \approx x(t) + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

where

$$\begin{aligned}
k_1 &= f(t,\, x(t)\,) \\
k_2 &= f(t + \frac{h}{2},\, x(t) + \frac{h}{2}\, k_1) \\
k_3 &= f(t + \frac{h}{2},\, x(t) + \frac{h}{2}\, k_2) \\
k_4 &= f(t + h,\, x(t) + h\, k_3).
\end{aligned}$$

There are also different types of convergence tests for step 5. But most often it is sufficient to require $||u - u_{old}|| = \sum_{i=1}^{N+1} |u_i - \ u_{old_i}|$ to be small, seeing that $\overrightarrow{u}$ is a vector of estimated values of the control while in the current iteration, and $\overrightarrow{u_{old}}$ is the vector of estimated values of the previous iteration. In this case , $||\cdot||$ is referred to the ℓ^1 norm vector, which the sum of absolute value of its terms. Both of these vectors are length of $N+1$ and have N time steps. The following equation is used in a slightly stricter convergence test, because we require the relative error to be negligibly small,

$$\frac{||\overrightarrow{u} - \overrightarrow{u_{old}}||}{||u||} \leq \delta,$$

where δ is the accepted tolerance. Though we have the equation, a small adjustment was made; There must allow for zero controls, so we multiply both sides by $||\overrightarrow{u}||$ in order to remove from its

denominator. Thus our requirement is

$$\begin{aligned}\delta||\overrightarrow{u}|| - ||\overrightarrow{u} - \overrightarrow{u_{old}}|| &\geq 0, \\ \delta \sum_{i=1}^{N+1} |u_i| - \sum_{i=1}^{N+1} |u_i - u_{old_i}| &\geq 0.\end{aligned}$$

This is a requirement of all variables, not for just the control, In the following examples we have set $N = 1000$ and $\delta = 0.001$

2.2 One Dimensional Maximum Optimal Control Problem

In this section we will be going over an 1 dimensional optimal control problem; And using Matlab on how it is used to solve it automatically. We will see two versions one from the text book and then introduce on what we have found. The following example is from the textbook by Suzanne Lenhart [6]. Let

$$\begin{aligned}&\max_u \int_0^1 Ax(t) - b(t)^2 \, dt \\ &\text{subject to } x'(t) \ = -\frac{1}{2}\, x(t)^2 \ + \ Cu(t),\ x(0) = x_0 > -2, \\ &A \geq 0,\ B > 0.\end{aligned}$$

In this example we require that $B > 0$ so so that is a maximization problem. Now we must develop the optimality system before we get to the code. First note the Hamiltonian is

$$H = Ax - Bu^2 - \frac{1}{2}\lambda x^2 + C\lambda u,$$

after using optimality condition that we discussed in chapter I,

$$0 = \frac{\partial H}{\partial u} = -2Bu + C\lambda \Rightarrow u^* = \frac{C\lambda}{2B},$$

we then now calculate the adjoint equations, and receive the following

$$\begin{aligned} x'(t) &= -\frac{1}{2}x^2 + Cu, x(0) = x_0 \\ \lambda'(t) &= -A + x\lambda, \lambda(1) = 0. \end{aligned}$$

With these two differential functions and the representation of u^*, we will be able to generate the numerical code. We will first see what the textbook has shown us, and then what we have found.

2.2.1 Book version

The following code is given from the text book Optimal control Applied to Biological Models by Suzanne Lenhart [6], note that the values $A, B, C, x0$ where set to $A = 1, B = 1, C = 4, x0 = 1$.

for $k = 0, \cdots$ **do**

Solve the State Equation

$x'(t) = g(t, x, u_k)$ for $t \in [t_0, t_f]$

$x(t_0) = x_0$

Solve the Transversality Condition

$\lambda'(t) = -\frac{\partial H}{\partial x}(t, \lambda, x, u_k)$

$\lambda(t_f) = \nabla\phi(x(t_f))$

Solve for u

solve $\frac{\partial H}{\partial u}(t, u, x, \lambda) = 0$ for u^*

$u_{k+1} = \frac{1}{2}u_k + \frac{1}{2}u^*$

if Convergence critera is satisfied then

$u = u_{k+1}$

end if

end for

The first line gives a variable y as an output and taking in variables $A,B,C,x0$, other than $x0$, A,B,C are set to be vectors, with inputs as we discussed above. The *while* loop begin with *test* being -1 and it will keep running till *test* is non-negative and then it is convergence and the loop will end. Since we need a variable time vector, we set t to be a vector that equally space nodes is $N+1$ nodes that is between 0 and 1, $t = linespace(0,1,N+1)$ and with that we set it to h to be equal to be the spacing between, $h = 1/N$; We then set u to be our initial guess so as an array of zeros of $N+1$ steps, $u = zeros(1,N+1)$, as for our x and λ will not be guesses because they will later on be replaced with the forward seep and the backward sweep process so we set them to be an array of zeros of $N+1$ steps, $\lambda = zeros(1,N+1)$ and $x = zeros(1,N+1)$, but with the initial value of x equal to $x0$, so $x(1) = x0$.

Within the while loop we set the previous values of the vectors $\overrightarrow{u}$, $\overrightarrow{x}$, $\overrightarrow{\lambda}$ of their previous values, we have them labeled as $\overrightarrow{u_{old}}$, $\overrightarrow{x_{old}}$, $\overrightarrow{\lambda_{old}}$, $u_{old} = u$, $x_{old} = x$, $\lambda_{old} = \lambda$. With the storing the current values as the previous.

we can now use the Runge-Kutta method to solve $\overrightarrow{x}$ forward in time; Which begins in the first *for* loop. It first calculates the value k_1, which is the RHS (right hand-side) of the differential equation, $k_1 = -0.5 * x(t)^2 + C * u(i)$, then it calculates k_2 with x being replaced with $x + \frac{h}{2}k_1$, we also replaced the time variable t with $t + h/2$, but there is no dependence on t in the differential equation, but u is a function of t. So when it comes to calculating k_2 and k_3 we replace u_i with $u_{i+h/2}$. This is also not assigned by our vector. In this case they found an interpolating polynomial of u; we have them labeled as $k2 = -0.5 * (x(i) + h2 * k1)^2 + C * 0.5 * (u(i) + u(i+1))$, $k3 = -0.5 * (x(i) + h2 * k2)^2 + C * 0.5 * (u(i) + u(i+1))$. As for k_4 we would need a full time step so call u_{i+1}, we have it as $k4 = -0.5 * (x(i) + h * k3)^2 + C * u(i+1)$. Note that to find x_1, we need x_2 and so on and so forth from $x_1 \cdots x_N$, But we need x_N to find x_{N+1}, hence the *for* loop only runs N times. The following is how it is labeled as,

```
for i = 1:N

k1 = -0.5*x(i)^2 + C*u(i);

k2 = -0.5*(x(i) + h2*k1)^2 + C*0.5*(u(i) + u(i+1));
```

```
    k3 = -0.5*(x(i) + h2*k2)^2 + C*0.5*(u(i) + u(i+1));

    k4 = -0.5*(x(i) + h*k3)^2 + C*u(i+1);

    x(i+1) = x(i) + (h/6)*(k1 + 2*k2 + 2*k3 + k4);

    end
```

With the second *for* loop, it uses the Runge-Kutta method as well to solve λ but this time backwards. It is almost similar to the forward sweep, but we introduce a new index where i counts to N, and j counts backwards from $N+1$ to 2, so we have it labeled as $i = 1 : N, j = N+2-i$. Just like forward sweep we calculate k_1, which comes from the differential equation, but since we are moving backwards it needs a increment of $-\frac{1}{N}$ times. And for $k2$ and k_3 we go a half step so we replaced λ_j with $\lambda_j - \frac{h}{2}k1$ and $\lambda_j - \frac{h}{2}k2$ and k_4 is similar to the forward Runge-Kutta to solve for $\overrightarrow{x}$, where we need a full step. The following is how the backward code is labeled

```
for i = 1:N

    j = N + 2 - i;

    k1 = -A + lambda(j)*x(j);

    k2 = -A + (lambda(j) - h2*k1)*0.5*(x(j)+x(j-1));

    k3 = -A + (lambda(j) - h2*k2)*0.5*(x(j)+x(j-1));

    k4 = -A + (lambda(j) - h*k3)*x(j-1);

    lambda(j-1) = lambda(j) - ...

        (h/6)*(k1 + 2*k2 + 2*k3 + k4);

end
```

Remember that each λ_i is used to find the previous, i.e λ_2 is used to find λ_1. Hence we count to 2.

The representation of $\overrightarrow{u}$ is now using the new values for $\overrightarrow{\lambda}$, But this is not stored as the control, more as a temporary vector $\overrightarrow{u1}$, we have this set up as $u1 = C * lambda/(2*B)$. Now the control $\overrightarrow{u}$ is the set of the last iteration of $\overrightarrow{u}$, which is $\overrightarrow{u_{old}}$ and its new representation. This is known as our convex combination as discussed from earlier. Then it follows to see if the variables converge and labeled as temp1,temp2 and temp3; In which temp1 is for $\overrightarrow{u}$, temp2 is for $\overrightarrow{x}$, and $\overrightarrow{\lambda}$.

Then they all are multiplied by $\delta = 0.001$. Then recall we need all three variables to be non-negative values so we have *test* be reassigned to be the minimum of these three values. Therefore if all three variables are non-negative then the *while* loop ends and convergence has been achieved, but if convergence has not been achieved then it runs the *while* till convergence occurs. To which it is labeled as the following,

```
u1 = C*lambda/(2*B);
u = 0.5*(u1 + oldu);


temp1 = delta*sum(abs(u)) - sum(abs(oldu - u));
temp2 = delta*sum(abs(x)) - sum(abs(oldx - x));
temp3 = delta*sum(abs(lambda)) - ...
    sum(abs(oldlambda - lambda));
test = min(temp1, min(temp2, temp3));
```

Finally all values are then stored in the output matrix y

```
y(1,:) = t;
y(2,:) = x;
y(3,:) = lambda;
y(4,:) = u;
```

2.2.2 What We Have Constructed

We will now go over what we have constructed. Using the same problem and using the same values as $A = 1, B = 1, C = 4, x_0 = 1$. We first plug in our optimal control problem in this case our F and G function; Our F function is labeled as max $F = Ax(t) - b(t)^2 \, dt$

, and our G function is labeled as $G = \; x'(t) \; = -\frac{1}{2}\, x(t)^2 \; + \; Cu(t)$. Our optimal control problem is the maximum of the integral of 0 to 1. So t_0 is 0 and our t_f is 1. We will also be needing a maxiter of 40, so it we have a max number of iteration that is passes through our **while** loop. Finally a number of time steps call it N_t. And since we have vectors x and u based of t, then we have a symbolic

vector t,x,u. This function will take in our functions F and G and its variables and compute it. The following is the psudocode,

for $k = 0,\cdots$ **do**

Solve the State Equation

$x'(t) = g(t,x,u_k)$ for $t \in [t_0,t_f]$

$x(t_0) = x_0$

Solve the Transversality Condition

$\lambda'(t) = -\frac{\partial H}{\partial x}(t,\lambda,x,u_k)$

$\lambda(t_f) = \nabla\phi(x(t_f))$

Solve for u

solve $\frac{\partial H}{\partial u}(t,u,x,\lambda) = 0$ for u^*

$u_{k+1} = \frac{1}{2}u_k + \frac{1}{2}u^*$

if Convergence critera is satisfied then

$u = u_{k+1}$

end if

end for

Just like the book we need vector time, but numerically since our t is symbolic for the time being. let call it tn and it is for the number of space nodes from $t0$ to tf with the number of time steps N to which is set to Nt. and since we have a maxiter, we must have a starting iter called *iter* and it is set to 0. Agian since we have a symbolic t,x,u we will name our variables $u0,xn$ and our $u0$ will be our initial guess and must be an array of zeros, and as well our xn. but similar to the book our xn will have the initial guess of $x0$. Since we also need a $\overrightarrow{\lambda}$ vector, but a numerical λ vector since later on we will be using a symbolic λ we shall call it $\lambda 0$. We then have the following labeling,

```
test = - 1
\delta = .001
N = Nt;
tn = linspace(t0,tf,N+1);
iter = 0;
u0 = zeros(1,N+1);
xn = zeros(1,N+1);
xn(1) = x0;
lambda0 = zeros(1,N+1);
```

Now before we go into our *while* loop, we must have make our functions anonymous and set our G to $@(t,x,u)$, and our F function to $@(t,x,u,lambda)$ since they will both form our Hamiltonian, that we discussed in chapter I. When then have a symbolic t,x,u and λ to help form our Hamiltonian H, we then turn it into a anonymous function of t,x,u,λ, as the

```
Ganon = matlabFunction(G);
Ganon2 = @(t,x,u) Ganon(u,x);
Fanon= matlabFunction(F);
Fanon2 = @(t,x,u,lambda) Fanon(lambda,x);
syms t x u
```

```
syms lambda
H = F + lambda.*G;
Hanon = matlabFunction(H);
Hanon2 = @(t,x,u,lambda) Hanon(lambda,u,x).
```

We then take the partial derivative of H with respect to x to find our λ' and our x' is our derivative of H with respect to λ, and finally to solve our u, we find the partial derivative of H with respect to u and set it equal to 0. Once we have our x,u and λ we then set them as anonymous functions. With H_x set to t,x,u, our H_λ set to t,x,u,λ and our H_u set to $@(lambda)$ since we will be using it to find the values in our backward sweep in the *while* loop. thus,

```
Hl = diff(Hanon2,x);
Hu = diff(Hanon2,u);
Hlanon = matlabFunction(Hl);
Hlanon2 = @(t,x,u,lambda) Hlanon(x,lambda);
usol = solve(  Hu == 0 ,u);
uanon = matlabFunction(usol);
uanon2 = @(lambda) uanon(lambda).
oldu = u0;
oldx = xn;
oldlambda = lambda0;
```

Now that we have our anonymous functions, we can now go into the *while* loop; But lets set an $\overrightarrow{oldu}, \overrightarrow{oldx}, \overrightarrow{oldlambda}$ and have it equal to $u0, xn, \lambda 0$.This is because we must start with the *while* loop with a numerical value not a symbolic value.Note unlike the textbook where $oldu, oldx$ and $oldlambda$ is x,u,λ, our algorithm has x,u,λ as symbolic. The following is the *while* loop of the code,

```
while (test < 0 && iter < maxiter )
    iter = iter +1;
```

```
    x = RKXU(Ganon2,x0,t0,tf,N,oldu);
    lambda = transversecond(Hlanon2,x,oldu,tf,N);
    u1 = uanon2(lambda);
    u = 0.5*(u1 + oldu);
    temp1 = delta * sum(abs(u))-sum(abs(oldu-u));
    temp2 = delta * sum(abs(x))-sum(abs(oldx-x));
    temp3 = delta * sum(abs(lambda))- ...
        sum(abs(oldlambda-lambda));
    test = min(temp1,min(temp2,temp3)) ;
    oldu = u;
    oldx = x;
    oldlambda = lambda;
end
```

Notice that the *while* loop is much shorter than the textbook version. That is due to the compartmentalization of the Runge-Kutta method. Both the forward sweep and the backward sweep have compartmentalized, as $x = RKXU(Ganon2, x0, t0, tf, N, oldu)$, and $lambda = transversecond(Hlanon2, x, oldu, tf, N)$ Our forward sweep is our x and our backward sweep is our λ. This is our pesudo code for the forward sweep function,

for $k = 1, \ldots, N_t$ **do**

$k_1 = g(t, x_k, u_k)$

$k_2 = g(t + \frac{h}{2}, x_k + \frac{h}{2}k_1, \frac{u_k + u_{k+1}}{2})$

$k_3 = g(t + \frac{h}{2}, x_k + \frac{h}{2}k_2, \frac{u_k + u_{k+1}}{2})$

$k_4 = g(t + h, x_k + hk_3, u_{k+1})$

$x_{k+1} = x_k + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$

end for

The matlab code can be found in appendix A1.4.1 With inputs of our function f, our initial value x_0, our t_0 and t_f, number of time step N, and initial guess u. We first input our h as our spacing between t_0 and tf equally by divided by N, then set out h_2 as our half of h. We then take an array of zeros of length x_0 and $N+1$ with the first column being the initial value of x_0. The *for* loop is our Runge-Kutta , where it runs N times, our t_i is our $t+h*i$ or t function. our k_1 is our function f while taking the values of the ith column of our x and u, to which it feeds to our k_2, to which takes a half step where (t_i+h_2) is our variable for the time step and our u is now $\frac{u(:,\ i)+(:,\ i+1)}{2}$. It is similar as we did for k_2 for k_3 but it takes the value of k_2 as input. Then k_4 takes the value of k_3 and solves it but with a full step similar to k_1. Finally it sums all of the column vectors of x_i and the sum k_1,k_2,k_3,k_4 multiply by $\frac{h}{6}$ and finds out $i+1$ column vector and gives our output x.

The final value of x then goes to backward sweep that uses the Runge-Kutta to solve for λ backwards in time. The matlabcode can be found in the appendix 1.4.2

Almost similar to the forward sweep, it takes inputs of partial derivative function of H with respect to x or Hx, our values from x that we have found, our u, which is our initial guess,the final time value of t_f, and our number of steps N. We also have in our code our initial time value as $t_0=0$ and our y is an array of zeros of size x with the initial column being equal to zero since we are dealing with 1 dimensional problems, later on we will explain the nth dimensional problems. Our f is our function with inputs $@(t,lambda,x,u)$ of function $H_x(tf-t,x,u,lambda)$, to which x and u are then flipped since we are going backwards and will start form the negatives and going upwards in values. Then it is plug in in our for loop, which like our forward sweep, uses our f function with inputs of t_k which is our time stepping of t_0+h*k from $k=1:N$ and since we flipped our x and u it means we start with negative values. With k_1 being a full step, k_2 and k_3 being half steps and k_4 being another full step with increments of $\frac{1}{N}$. After computing y it is then flip and plugged into lambda, to which is then used for the last half of the while loop.

The following is the 2nd half of the while loop,

```
u1 = uanon2(lambda);
```

```
    u = 0.5*(u1 + oldu);


    temp1 = delta * sum(abs(u))-sum(abs(oldu-u));
    temp2 = delta * sum(abs(x))-sum(abs(oldx-x));
    temp3 = delta * sum(abs(lambda))- ...
        sum(abs(oldlambda-lambda));
    test = min(temp1,min(temp2,temp3)) ;


    oldu = u;
    oldx = x;
    oldlambda = lambda;

end
```

As well similar we would need a representation of $\overrightarrow{u}$ with values of $\overrightarrow{\lambda}$, we will called $u1$ as a temporary vector. Then the last set of u is convex combination of the previous representation (u_{old}) and the new representation. Then it checks for convergence with $test = -1$ for a non-negative number; And with $temp_1$ being for u, $temp_2$ being for x and $temp_3$ being for λ. If convergence occurs then the while loop end, but if it hasn't occur, then x, u, λ will then be plugged into $x_{old}, u_{old}, \lambda_{old}$ and the loop will begin again till convergence has achieved.

2.3 Bounded Case Optimal Control Problem

For many realistic problems they require bounded controls, before we get into the example, let us go over the necessary conditions.

2.3.1 Necessary Conditions

I order to solve bounded cases, we must develop alternate necessary conditions. So lets consider the following problem

$$\begin{aligned}
&\max_u \int_{t_0}^{t_f} f(t,\, x(t),\, u(t))\, dt \, + \phi(x(t_f)) \\
&\text{subject to } x'(t) \, = g(t,\, x(t),\, u(t))\; x(t_0) = x_0 \\
&a \le u(t) \le b,
\end{aligned}$$

where a and b are real fixed constants and $a \; < \; b$. we let $J(u)$ be the value of the objective function at control u, where $x = x(u)$ is the state equation so we have the following

$$J(u) = \int_{t_0}^{t_f} f(t,\, x(t),\, u(t))\, dt \, + \phi(x(t_f)).$$

We then let $u^*,\ x^*$ be a fixed optimal pair. and let $h(t)$ be a piecewise continuous function where exists a positive constant ε_0 such that for all $\varepsilon \in (0, \varepsilon_0],\ u^\varepsilon(t) = u^*(t) + \varepsilon h(t)$ is admissible, such as

$$a \le u^\varepsilon(t) \le b \text{ for all } t.$$

Due to bounds on the controls, its derivative of the objective function nay not be zero, and since u^* may be at the bounds (end points of the range) at some points in time; we may only know the sign of the ε parameter. Let $x^\varepsilon(t)$ be the corresponding state variable for each $\varepsilon \in (0, \varepsilon_0]$ as it was done in I. First introducing a piecewise differentiable adjoint variable $\lambda(t)$ and then applying the Fundamental Theorem of Calculus with $J(u^\varepsilon)$, we get the following

$$\begin{aligned}
J(u^\varepsilon) \;\; &= \;\; \int_{t_0}^{t_f} [f(t,\, x^\varepsilon,\, u^\varepsilon) \, + \lambda(t)\, g(t, x^\varepsilon, u^\varepsilon) + x^\varepsilon \lambda'(t)]\, dt \\
&\quad - \lambda(t_0) x_0 + \lambda(t_1) x^\varepsilon(t_1) \, + \phi(x(t_1)).
\end{aligned} \tag{2.1}$$

The maximum of $J(u)$ with respect to u occurs at u^*,

$$0 \geq \frac{d}{d\varepsilon} J(u^\varepsilon) \mid_{\varepsilon=0} \lim_{\varepsilon \Rightarrow 0^+} \frac{J(u^\varepsilon) - J(u^*)}{\varepsilon}. \tag{2.2}$$

We must note that the constant ε was chosen to be positive, so that the limit can only be taken from one side. The numerator will be non-postive clearly, as u^* is the maximal. This gives the inequality shown, instead of equality as in I; As before we choose the adjoint carable such that

$$\lambda'(t) = -[f_x(t, x^*, u^*) + \lambda(t) g_x(t, x^*, u^*)],\ \lambda(t_1) = \phi'(x^*(t_1)).$$

Now 2.1 and 2.2 is reduced to

$$0 \ \geq \ \int_{t_0}^{t_f} (f_u + \lambda g_u)\, h\, dt, \tag{2.3}$$

which this inequality holds true for all h as described above.

Let s be a point of continuity of u^* with $a \ \leq \ u^*(s) \ < \ b$. Now suppose $f_u + \lambda g_u > 0$ at s. As u^* is continuous at s, so is $f_u + \lambda g_u$. Thus there is a small interval I, containing s, on which $f_u + \lambda g_u$ is strictly positive and $u^* <$ b. Now let

$$M = \max\{u^*(t) : t \in I\} < b.$$

Define a particular h by

$$h(t) = \begin{cases} b - M & \text{if } t \in I, \\ 0 & \text{if } t \notin I \end{cases}$$

Note that $h > 0$ in I. It can also be seen that $a \leq u^* + \varepsilon h \leq b$ for all $\varepsilon \in [0,1]$. But

$$\int_{t_0}^{t_f} (f_u + \lambda g_u)\, h\, dt \ = \int_I (f_u + \lambda g_u)\, h\, dt \ > \ 0$$

by choice of I and H. This will contradict 2.3 and it implies $f_u + \lambda g_u \leq 0$ at s. Similarly, let s be a point of continuity of u^* with $a < u^*(s) \leq b$. Now suppose $f_u + \lambda g_u \ < \ 0$ at s. As before,

there is a small interval I, containing s, on which $f_u + \lambda g_u$ is strictly negative and $u^* > a$. Let $m = \min\{u^*(t)\ : t \in I\ \}$, and define a variation function by $h = a - m$ on I and 0 off I. Then, $a \le u^* + \varepsilon h \le b$ for all $\varepsilon \in [0,\ 1]$. But,

$$\int_{t_0}^{t_f} (f_u + \lambda g_u)\, h\, dt\ = \int_I (f_u + \lambda g_u)\, h\, dt\ >\ 0$$

which again contradicts 2.3. So, $f_u + \lambda g_u \ge 0$ at s, This holds for all points of continuity of s. so in summary

$$\begin{aligned} u^*(t)\ &=\ a \text{ implies } f_u + g_u\ \le\ 0 \text{ at } t, \\ a\ <\ u^*(t)\ &<\ b\ \text{ implies } f_u + g_u\ =\ 0 \text{ at } t, \\ u^*(t)\ &=\ b \text{ implies } f_u + g_u\ \ge\ 0 \text{ at } t. \end{aligned} \tag{2.4}$$

These conditions of 2.4 are the same as

$$\begin{aligned} f_u + g_u\ <\ 0 \text{ at } t \quad &\text{implies} \quad u^*(t)\ =\ a \\ f_u + g_u\ =\ 0 \text{ at } t \quad &\text{implies} \quad a\ \le\ u^*(t)\ \le\ b \\ f_u + g_u\ >\ 0 \text{ at } t \quad &\text{implies} \quad u^*(t)\ =\ b. \end{aligned} \tag{2.5}$$

This hold true for all points of continuity t of u^*. As they are irrelevant to the objection function and the state equation. Neglecting the remaining points, the new necessary conditions can then be complied as before forming the following Hamiltonian

$$H(t,x,u,\lambda)\ =\ f(t,x,u)\ +\ \lambda(t)\ g(t,x,u),$$

the necessary conditions for x^* and λ remain unchange, namely for

$$\begin{aligned} x'(t) &= \frac{\partial H}{\partial \lambda} \quad , \quad x(t_0) = x_0, \\ \lambda'(t) &= -\frac{\partial H}{\partial x} \quad , \quad \lambda(t_1) = \phi'(x(t_1)). \end{aligned}$$

It then follows from the derivation as

$$\begin{cases} u^* = a & \text{if } \frac{\partial H}{\partial u} < 0 \\ a \leq u^* \leq b & \text{if } \frac{\partial H}{\partial u} = 0 \\ u^* = b & \text{if } \frac{\partial H}{\partial u} > 0. \end{cases} \tag{2.6}$$

The following problem that we have will be using is given in the book. The following problem that has been made is the weight parameter of B has been removed and only two weight parameter are being used, both A and C will be used in the following problem.

$$\begin{aligned} &\max_u \int_0^1 Ax(t) - u(t)^2 \, dt \\ &\text{subject to } x'(t) \; = -\frac{1}{2}x(t)^2 + Cu(t) \; x(0) = x_0 > -2 \\ &M_1 \leq u(t) \leq M_2, A > 0. \end{aligned}$$

With the following weight parameters being equal to $A = x_0 = 1$, and $C = 4$. the bounded parameters with $M_1 = 0$ and $M_2 = 2$

2.3.2 What We Have Constructed

We will now go over what we have constructed. Using $A = 1, C = 4, x0 = 1$. We plug in our optimal control problem; Our F function is max $F = Ax(t) - u(t)^2 \, dt$
, and our G function is $G = \; x'(t) \; = -\frac{1}{2} \; x(t)^2 \; + \; Cu(t)$. Our optimal control problem is the maximum of the integral of 0 to 1. So t_0 is 0 and our t_f is 1. As previous we will also be needing a maxiter and number pf iterations, to which it will remain the same as the previous cases i.e maxiter

is 40 and Nt is 1000. Finally the bounded parameters as $M_1 = 0$ and $M_2 = 2$.

```
A = 1;

C = 4;

x0 =1;

M1 = 0;

M2 = 2;

syms t x u

F = A*x - u.^2;

G = -(1/2)*x.^2 + C*u;

Nt = 1000;

t0 = 0;

tf = 1;

maxiter = 40;
```

Then we plug in our function

```
[x,lambda,u] = OptControl8(F,G,x0,t0,tf,maxiter,Nt,M1,M2);
```

The following is the pseudo-code we have constructed

for $k = 0, \cdots$ **do**

Solve the State Equation

$x'(t) = g(t, x, u_k)$ for $t \in \ [t_0, t_f]$

$x(t_0) = x_0$

Solve the Transversality Condition

$\lambda'(t) = -\frac{\partial H}{\partial x}(t, \lambda, x, u_k)$

$\lambda(t_f) = \nabla\phi(x(t_f))$

Solve for u

solve $\frac{\partial H}{\partial u}(t, u, x, \lambda) = 0$ for u^* between M_1, and M_2

$u_{k+1} = \frac{1}{2}u_k + \frac{1}{2}u^*$

if Convergence critera is satisfied then

$u = u_{k+1}$

end if

end for

Similar as the previous cases we would need a representation of $\overrightarrow{u}$ with values of $\overrightarrow{\lambda}$, we will called u_1 as a temporary vector. in line 35 before the last set of u is made, we must need the minimum of the highest bound and the maximum of the lowest bound with the function uanon2 with the result of λ given from the transverse condition. Thus the processes is repeated till convergence has achieved.

CHAPTER III

FORWARD-BACKWARD SWEEP METHOD: MULTI DIMENSIONAL

Before we head into Multi-dimensional optimal control theory let us review Optimal control with several variables.

3.1 Optimal Control with several variables

This method is developed for one control and state and can be easily extended to multiple state and control variables. so consider the a problem that has n state variables, m control variables and a payoff function ϕ, we have the following equation

$$\begin{aligned} \max_{u_1,\cdots,u_m} \quad & \int_{t_0}^{t_f} f(t,x_1(t),\cdots,x_n(t),u_1(t),\cdots,u_m(t))dt + \phi(x_1(t_f),\cdots,x_n(t_f)) \\ \text{subject to} \quad & x_i'(t) = g_i(t,x_1(t),\cdots,x_n(t),u_1(t),\cdots,u_m(t)), \\ & x_i(t_0) = x_{i0} \quad \text{for} \quad i = 1,2,\cdots,n, \end{aligned}$$

where functions f, g_i are continuously differentiable in all variables. There are no requirements on m,n. Note that, $m < n,\ m = n,\ m > n$ are acceptable. Using vector notation to change the problem to be more a familiar form. We let $\vec{x}(t) = [x_1(t),\cdots,x_n(t)],\ \vec{u}(t) =]u_1(t),\cdots,u_m(t)]),$ $\vec{x}_0 = [x_{10},\cdots,x_{n0}]$, and $\vec{g}(t,\vec{x},\vec{u}) = [g_1(t,\vec{x},\vec{u}),\cdots,g_n(t,\vec{x},\vec{u})]$. We then can write it as the following

$$\begin{aligned} \max_{\vec{u}} \quad & \int_{t_0}^{t_f} f(t,\vec{x}(t),\vec{u}(t))dt + \phi(\vec{x}(t_f)) \\ \text{subject to} \quad & \vec{x}'(t) = \vec{g}(t,\vec{x}(t),\vec{u}(t)), \vec{x}(t_0) = \vec{x}_0. \end{aligned}$$

3.2 Multi-Dimensional Theory

Multi-Dimensional theory is a concept that has existed in mathematics. Suggesting that there exits a dimension beyond three dimensions. To understand about multi-dimension theory, we must have a strong knowledge of linear algebra and vector spaces. In a tradition representation on the three-dimension space is usually $(x,\ y,\ z)$ and the vectors are usually expressed using components in their coordinate system; But in multi-dimensional theory the number of coordinate points is increased and vectors also increases and becomes more numerous. So if there were to be a multi-dimension defined as n-dimension vector space, where n is the number of dimensions, then there would be n coordinates.

Multi-Dimensional theory allows us to explore more into complex systems that cannot be described in three dimensions. Applications can be found in quantum mechanics and string theory. Another tool that is used in multi-dimensional theory is linear transformation. A linear transformation is taking vector spaces from place to another, and by doing this it maps out points and objects in different dimensions. Which is a major role in multi-dimensional theory.

3.3 Multi-dimensional optimal control problem one

The following problem that we will using is the Epidemic Model. Optimal control is used in this to find a vaccination schedule for an epidemic disease. Lets say that a micro-parasitic infectious disease is considered. Now suppose that a permanent immunity to the disease can be achieved through natural recovery or immunization; But we know that during birth immunity is not achieved hence everyone is born susceptible, thus our goal is to minimize the number of a infectious population and the overall cost of the vaccine during a fixed time period.

To model this, we used a standard SEIRN model. so we let $S(t), I(t)$, and $R(t)$ represent the number of susceptible,infectious, and recovered or immune representatives over a time t. This model allows for an incubation period for the disease inside its host,where an infected person latent for some time before becoming infectious,which then creates an exposed class. So we let $E(t)$ be the number of exposed or latent individuals over a time t.

As for $N(t)$, it is the number of people in the population, so that $N(t) = S(t) + E(t) + I(t) + R(t)$.

So let (t) be the control as the percentage of susceptible individuals being vaccinated over a period of time. As the vaccination of the entire susceptible population is impossible, we would bound the control with $0 \leq u(t) \leq 0.9$. As well let b be the natural birth rate of the population and d be the natural death rate. The incidence of the disease can be described by the term of $cS(t)I(t)$. the parameter e is the rate of exposed individuals that become infectious, and g is the rate of infectious individual recovered. Therefore $\frac{1}{e}$ is the mean latent period, and $\frac{1}{g}$ is the mean infectious period before recovery, if recovery occurs. The parameter a is the death rate due to the disease in the infections individuals. We then have the following problem,

$$\min_u \int_0^T AI(t) + u(t)^2 dt \iff \max \int_0^T -AI(t) - u(t)^2 dt$$

subject to

$$\begin{aligned}
&S'(t) = bN(t) - dS(t) - cS(t) - u(t)S(t), \quad S(0) = S_0 \geq 0,\\
&E'(t) = cS(t)I(t) - (e+d)E(t), \quad E(0) = E_0 \geq 0,\\
&I'(t) = eE(t) - (g+a+d)I(t), \quad I(0) = I_0 \geq 0,\\
&R'(t) = gI(t) - dR(t) + u(t)S(t), \quad R(0) = R_0 \geq 0,\\
&N'(t) = (b-d)N(t) - aI(t), \quad N(0) = N_0,\\
&0 \leq u(t) \leq 0.9
\end{aligned}$$

Note that when solving for R it only appears in the R' differential equation so we can ignore R Since it does not depend on any other variable

3.4 Multi-dimensional optimal control problem two

The next problem we used is a HIV treatment model. Optimal control is used to find an optimal strategy in the treatment for human immunodeficiency virus or HIV. This time is is based off the immune system of the person.

We consider the treatment of reverse transcription inhibitors, for example AZT, which effects " infectivity" of the life cycle of HIV of the host cell.

This treament is assumed to act to reduce the infectivity of the virus for a finite time t until the resistance occurs. SO this is measured bas on the increase of the CD4^{+}T cell count. So we let $T(t)$ and $T_i(t)$ be the concentration of unifected and infected CD4^{+}T cells. We let $V(t)$ be the concentration of free virus particles, which is refered to the the population count per volume. So we have

$$\frac{s}{1+V(t)}$$

be the source term of the rate of generation of new CD4^{+}T cells. Let r be the growth rate of T cells per day. T_{max} is the maximum level of T cells, which implys that $T(t)$ growth is logistic. Let $m1, m2, m3$ is the natural death of uninfected cells (T), infected cells (T_i) and free virus particles (V). N is the average number of virus particles produced before the host cell dies. $u(t)$ is the control or it describes the strength of the chemotherapy, where $u(t) = 0$ is the maximum and $u(t) = 1$ is no therapy. We then have the following problem,

$$\max_u \int_0^{t_{final}} AT(t) - (1-u(t))^2 dt$$

subject to

$$\begin{aligned}
T'(t) &= \frac{s}{1+V(t)} - m_1 T(t) + rT(t)\left[1 - \frac{T(t)+T_i(t)}{T_{max}}\right] \\
&\quad -u(t)kV(t)T(t), \\
T_i'(t) &= u(t)kV(t)T(t) - m_2 T_i(t), \\
V'(t) &= Nm_2 T_i(t) - m_3 V(t),
\end{aligned}$$

$T(0) = T_0 > 0, \quad T_i(0) = T_{i0} > 0, \quad V(0) = V_0 > 0, 0 \leq u(t) \leq 1.$

CHAPTER IV

MULTI DIMENSIONAL SWEEP

4.1 Multi Dimensional

As explain in the previous chapter we will be using the SERIN model as our example, the following pseudo code is our multi-optimal control Matlab Code that we have constructed

for $k = 0, \cdots$ **do**

Solve the State Equation

$x'(t) = g(t, x, u_k)$ for $t \in [t_0, t_f]$

$x(t_0) = x_0$

Solve the Transversality Condition

$\lambda'(t) = -\frac{\partial H}{\partial x}(t, \lambda, x, u_k)$

$\lambda(t_f) = \nabla\phi(x(t_f))$

Solve for u

$\frac{\partial H}{\partial u}(t, u, x, \lambda) = 0$ for u^*

$u_{k+1} = \frac{1}{2}u_k + \frac{1}{2}u^*$

if Convergence critera is satisfied then

$u = u_{k+1}$

end if

end for

We will now go over what we have constructed. Using the SEIRN values as $A = 0.1, b = 0.525, c = 0.0001, d = 0.5, e = 0.5, g = 0.1, a = 0.2, T = 20$.Our bounded parameters are $M1 = 0$ and $M2 = 0.9$. For our initial values of our SEIRN model will be $S0 = 1000, E0 = 100, I0 = 50, R0 = 15, N0 = S0 + E0 + I0 + R0$; and since we need an $x0$ of the size of 1 by n matrix transpose with the last entry being $N0$ we have it set as $x0 = [1000; 100; 50; N0]$, we must need a identity matrix the length of $x0$. We then plug in our optimal control problem in F and G function; Our F function is labeled as max $F = A*(I(:,3)'*x) + u.^2;\ dt$, and our G function is labeled as $G = [b*(I(:,4)'*x) - d*(I(:,1)'*x) - c*(I(:,1)'*x)*(I(:,3)'*x) - u*(I(:,1)'*x);\ c*(I(:,1)'*x)*(I(:,3)'*x) - (e+d)*(I(:,2)'*x);\ e*(I(:,2)'*x) - (g+a+d)*(I(:,3)'*x);\ (b-d)*(I(:,4)'*x) - a*(I(:,3)'*x)];$. Our optimal control problem is the maximum of the integral of 0 to T. So t_0 is 0 and our t_f is T. We will also be needing a maxiter of 1 due to the fact it will run a bit longer than our other cases, so it we have a max number of iteration that is passes through our **while** loop. Finally a number of time steps call it Nt. And since we have vectors x and u based of t, then we have a symbolic vector t, u and a symbolic matrix the size of $x0$. we then have the following labeling

```
b = 0.525;
d = 0.5;
c = 0.0001;
e = 0.5;
g = 0.1;
a = 0.2;
M1 = 0;
M2 = 0.9;
T = 20;
A = 0.1;
S0 = 1000;
E0 = 100;
```

```
I0 = 50;

R0 = 15;

N0 = S0+E0+I0+R0;

x0 = [1000;

    100;

    50;

    N0];

I = eye(length(x0));

syms t u

syms x [size(x0)] matrix

F = A*(I(:,3)'*x) + u.^2;

phi = 0;

G =

[b*(I(:,4)'*x) - d*(I(:,1)'*x) - c*(I(:,1)'*x)*(I(:,3)'*x) - u*(I(:,1)'*x);

c*(I(:,1)'*x)*(I(:,3)'*x) - (e+d)*(I(:,2)'*x);

e*(I(:,2)'*x) - (g+a+d)*(I(:,3)'*x);

(b-d)*(I(:,4)'*x) - a*(I(:,3)'*x)];

t0 = 0;

tf =T;

maxiter = 1;

Nt =100;
```

Then we plug in our function

```
[x, lambda, u] = multiOptControl(F,G,x0,t0,tf,maxiter,Nt,phi,M1,M2)
```

Remember like the previous cases numerically since our t is symbolic for the time being. let call it tn and it is for the number of space nodes from $t0$ to tf with the number of time steps $Nt+1$. and since we have a maxiter, we must have a starting iter called *iter* and it is set to 0. Again since we

have a symbolic t,u vector and our symbolic x is a symbolic matrix the size of $x0$ as well our λ is a symbolic matrix the size of x we will name our variables $u0$ and our $u0$ will be our initial guess and must be an array of zeros, and as well our $xn.$ but similar to the book our xn will have the initial guess of $x0$ in a for loop of 1 to the length of $x0$ since it has to go in every entry and replace every beginning entry with $x0$. Since we also need a $\overrightarrow{\lambda}$ vector, but a numerical λ vector called λ_0. We then have the following labeling,

```
    test = -1;
delta = 0.001;
tn = linspace(t0,tf,Nt+1);
iter = 0;
u0 = zeros(1,Nt+1);
xn = zeros(length(x0),Nt+1);
for i = 1:length(x0)
    xn(i) = x0(i);
end
lambda0 = zeros(length(x0),Nt+1);
syms t u
syms x [size(x0)] matrix
syms lambda [size(x)] matrix
```

Now that we have our G we must make symbolic matrix to a array of symbolic scalar variables we call it Gmatrix, We then make into a matlabFunction called Gmf, then with Gmf and u,x we put them into another function that we have created called geval, in which we will explain later on in the chapter as well as the other subcodes that we have constructed for this, it also set to $@(u,x)$ in Ganon followed by Ganon2 being a function of Ganon of u,x at $t,x,u.$ The same procedure follows as well for F.

Now given if ϕ is zero or a non-zero it will then follow an *if* loop. If $\phi = 0$ then Lfxanon will then be 0, else if non-zero then it will make phi into a matlabFunction and set it $@(x)$ to which

it will then differentiate the function with respect to x and make it into a another matlabFunction. Then it follows as Lfxanon2 as a function of Lfxanon of x $@(x)$.

We then make the Hamiltonian by the sum of F and the product of λ transpose and G.

```
Gmatrix = symmatrix2sym(G);
Gmf = matlabFunction(Gmatrix);
Ganon = @(u,x) geval(Gmf,u,x);
Ganon2 = @(t,x,u) Ganon(u,x);
Fmatrix = symmatrix2sym(F);
Fanon= matlabFunction(Fmatrix);
Fanon2 = @(t,x,u) Fanon(u,x);
if phi == 0
    Lfxanon = 0;
elseif phi ~= 0
    Lfanon = matlabFunction(phi);
    Lfanon2 = @(x) Lfanon(x);


    Lfx = diff(Lfanon2,x);
    Lfxanon = matlabFunction(Lfx);
    Lfxanon2 = @(x) Lfxanon(x);
end
H = F + lambda.'*G;
```

Once the Hamiltonian is made, we then take the gradient with respect to the following; For Hx we take the gradient of H with respect to λ, for Hl we take the gradient of H with respect to x, and finally Hu is the gradient of H with respect to u, then they all become a symbolic matrix to a array of symbolic scalar variables. Hx is then a matlabFunction of Hxmatrix called Hxanon and then converted to a function at (t,x,u) called Hxanon2. For Hl it is then inputted into a evaluation function that we have constructed called leval with inputs Hlmatrix and the length of $x0$, this is to

find the indices of both x and λ, whose outputs are labeled as xi and li, followed by being inputted into another function that is a "wrapper" namely to coverts the cell array into their own separate cells and its inputs are Hlmatrix,λ, u, x, xi, and li. which becomes a function at $t, x, u, lambda$ into a variable Hlanon. As for Hu it is then solved for u when Humatrix is equal to zero and then enters the same process as it did for Hlmatrix by being evaluated and then being inputted into a "wrapper."

Followed by $oldu = u0$, $oldx = xn$, and $oldlambda = lambda0$.

```
Hl = gradient(H,x);
Hlmatrix = symmatrix2sym(Hl);
[xi, li] = leval(Hlmatrix,length(x0));
Hlanon = @(t,x,u,lambda) lwrap(Hlmatrix,lambda,u,x,xi,li);
Hu = gradient(H,u);
Humatrix = symmatrix2sym(Hu);
usol = solve( Humatrix == 0 ,u);
[uli, uxi] = ueval(usol,length(x0));
uanon = @(lambda,x) uwrap(usol,lambda,x,uli,uxi);
   oldu = u0;
   oldx = xn;
   oldlambda = lambda0;
```

We then start the while loop with test being less than 0 to check for convergence as well iter being less than maxiter to check how many iterations it will take. As in the previous examples, we then input variables Ganon2, $x0, t0, Nt, oldu$ into the froward-backward sweep or the the Rung-kutta to find x. Then our Lfxanon is then inputed into a if loop to see if it equals to zero. Note that this is due to $\lambda_j(t_1) = \phi_{x_j}(t_1)$, for $j = 1, \cdots, n$ .If our Lfxanon is equal to zero then our lambdaf which is our Lfxanon. If it does not equal to zero then it enters another if loop, this time if the number of function input arguments are given in the call to the currently executing function this case zero, if so then it is then is then evaluated to its name or its handle, and will be our lambdaf, else then lambdaf will be the evaluation of differentiation function we have created earlier with our input being the

last column of x resulted earlier. To which lambdaf will be our ϕ function for our backward-forward sweep or our transverse condition followed by our other inputs Hlanon, $x, oldu, tf, Nt$ to find our λ

```
while (test < 0 && iter < maxiter )
iter = iter +1;
x = RKXU(Ganon2,x0,t0,tf,Nt,oldu);
if Lfxanon == 0
    lambdaf = Lfxanon;
elseif Lfxanon ~= 0
    if nargin(Lfxanon) == 0
        lambdaf = feval(Lfxanon);
    else
        lambdaf = Lfxanon2(x(:,end));
    end
end
lambda = transversecondN(Hlanon,x,oldu,tf,Nt,lambdaf);
```

similar to our bounded cases we would need a representation of $\overrightarrow{u}$ with values of $\overrightarrow{\lambda}$, we will called $u1$ as a temporary vector. Before the last set of u is made, we must need the minimum of the highest bound and the maximum of the lowest bound with the function uanon2 with the result of λ given from the transverse condition to find our new representation and adding our previous representation multiplied by 0.5 our convex combination is given. Then it checks for convergence with $test = -1$ for a non-negative number; And with $temp1$ being for u, $temp2$ being for x and $temp3$ being for λ.Thus the processes is repeated till convergence has achieved, with $oldu, oldx$,and $oldlambda$ being replaced with u, x, λ.

```
u1 = min(M2, max(M1,uanon(lambda,x)));
u = 0.5*(u1 + oldu);
temp1 = delta * sum(abs(u),2)-sum(abs(oldu-u),2);
```

```
    temp2 = delta * sum(abs(x),2)-sum(abs(oldx-x),2);
    temp3 = delta * sum(abs(lambda),2)- ...
        sum(abs(oldlambda-lambda),2);
    test = min( [temp1; temp2; temp3] ) ;
    oldu = u;
    oldx = x;
    oldlambda = lambda;

end
```

To which it is then graphed with x, u, and λ against time. We will show the results in the next chapter.

4.2 Multi Dimensinal Greedy

The following the pesdo code for a muti-dimensional optimal control problem with a convex combination parameter of 1/2 or a greedy update. This was researched by a fellow UTRGV student by the name of Elina Seppala [5]. The following is the pesdo code for an Greedy Automatic Optimal control Solver.

for $i = 0, \cdots$ **do**

    Solve the State Equation $(x' = g(t, x, u_k))$

    Solve the Transversality Condition $(\lambda' = -H_x(t, \lambda, x, u_k))$

    Solve $\frac{\partial H}{\partial u}(t, u, x, \lambda) = 0$ for u^*

    for $\phi_j = \frac{1}{N_c}, \frac{2}{N_c}, \ldots, 1$ **do**

        $u = \phi_j u^* + (1 - \phi_j) u_k$

        $F_j = \int_{t_0}^{t_f} f(t, u, x) dt + \phi(x(t_f))$

    end for

      $j =$ index that maximizes F_j

$u_{k+1} = \phi_j u^* + (1-\phi_j)u_k$

if Convergence criteria is satisfied then

$u = u_{k+1}$

end if

end for

For the Multi Dimensional Greedy, we will still be using the SEIRN model; It is similar to the previous algorithm that we have constructed but with some minor changes, as well as adding new sub-codes. Such as the following.

Everything remains the same as previous, however we added a new variable called $Nc = 2$ this is our representation of our convex combination parameter.

```
test = -1;
delta = 0.001;
tn = linspace(t0,tf,Nt+1);
iter = 0;
Nc = 2;
u0 = zeros(1,Nt+1);
```

Similar to the previous algorithm, everything remains, with changes to the after F becomes a symbolic matrix variable to an array of symbolic scalar variables, it is then inputted into a function called xeval for the length of $x0$, this is to find which indices of x are non zero, whose output is xi. Is then plugged into a "wrapper" as well with other inputs Fmatrix, u,x, to which evaluates the inputs giving it numerical value. It then follows the same as the previous algorithm.

```
Gmatrix = symmatrix2sym(G);
Gmf = matlabFunction(Gmatrix);
Ganon = @(u,x) geval(Gmf,u,x);
Ganon2 = @(t,x,u) Ganon(u,x);
```

```
Fmatrix = symmatrix2sym(F);
[xi] = xeval(Fmatrix,length(x0));
Fanon2 = @(x,u) fwrap(Fmatrix,u,x,xi);


if phi == 0
    Lfxanon = 0;
elseif phi ~= 0
    Lfanon = matlabFunction(phi);
    Lfanon2 = @(x) Lfanon(x);


    Lfx = diff(Lfanon2,x);
    Lfxanon = matlabFunction(Lfx);
    Lfxanon2 = @(x) Lfxanon(x);
end
H = F + lambda.'*G;
Hmatrix = symmatrix2sym(H);
Hanon = matlabFunction(Hmatrix);
Hanon2 = @(t,x,u,lambda) Hanon(lambda,u,x);
```

As previously mention, even in the while loop remains the same till we reached our convex combination. After finding our new representation i.e $u1$, we then enter a for loop to find our convex combination quicker. First we let a θ index be set to zero, then for θ from $\frac{1}{Nc}$ to 1 in $\frac{1}{Nc}$ time steps. Then the index of θ is equal to $thindex = thindex + 1$ this is to see how many iterations it takes. Then set $u_t ry$ equal to the sum of $(1-\theta) * u1$ (which is our new representation) and $\theta * oldu$ (which is our old representation). Then it is inputted in the Rung-kutta algorithm to find our x this time with $u_t ry$. Then another for loop is made this time from $k = 1 : Nt + 1$, this is to see which arrays of fmatrix is numerical for each last k-th column of x and for each k-th value of $U_t ry$. Once our array is made, it is then inserted into a Trapezoidal method along with the initial time and final

time value, which finds our objective function f. We then find the minimum of f and outputs are fmin, and its indices or ind. Then we set our u to be our $u_t ry(\theta)$ which is optimized by $\frac{ind}{Nc}$ i.e ($u = (1-th) * u1 + th * oldu$). Thus we have our convex combination and follows the same algorithm as previous.

```
u1 = min(M2, max(M1,uanon(lambda,x)));


thindex = 0;
for th = (1/Nc):(1/Nc):1
    thindex = thindex+1;
    u_try = (1-th)*u1 + th*oldu;
    x = RKXU(Ganon2,x0,t0,tf,Nt,u_try);
    for k = 1:Nt+1
        Farray(k) = Fanon2(x(:,k),u_try(k));
    end
    f(thindex) = Trapmethod(Farray,t0,tf);
end
[fmin, ind]  = min(f);
th = ind/Nc;
u = (1-th)*u1 + th*oldu ;


temp1 = delta * sum(abs(u),2)-sum(abs(oldu-u),2);
temp2 = delta * sum(abs(x),2)-sum(abs(oldx-x),2);
temp3 = delta * sum(abs(lambda),2)- ...
    sum(abs(oldlambda-lambda),2);
test = min( [temp1; temp2; temp3] ) ;
oldu = u;
oldx = x;
```

```
oldlambda = lambda;
```

4.3 Subcodes

The following sections are pesdo-codes that were constructed used in the both Multi-Optimal and Multi-Optimal Greedy.

4.3.1 Geval

The Geval is a function that we have constructed to be able to evaluate the G function. Given inputs G which is our function matrix and inputs u and x. Which are the variables of the function matrix and converts them into a numeric array, then we plug them back into our function matrix G and evaluate out answer.

- Initial definition of g is a symbolic function in Matlab.
- Needed to covert g into a symbolic matrix type.
- Followed by converting into a Matlab function type.
- However if $x : \mathbb{R} \Rightarrow \mathbb{R}^n$ then this Matlab function expressed g as $g(t, x_1, x_2, \cdots, x_n, u)$.
- We used the `num2cell` command to convert the vector x into a cellular list.

4.3.2 Xeval

The Xeval is another function used to evaluate the our function F which is our control function and input m for being the length of x_0. We then create a real symbolic matrix x of size m by 1. In a **for** loop from 1 to the length of our symbolic matrix x called k, for each iteration of k in our variable fx will be the sum of the differation of F with respect to x. We then create a symbolic 0 called z and label i as our numeric 0. Then in a another **for** loop with k from 1 to the length of x, we check to see if each varaible of fx is equal to zero for each iteration of k and labeled as fxz. Then in a **if** loop we check to see if any indices from fxz are 0 and if they are zeros then our iteration count or i will be adding one per zero. Then we count each indices of i equal to our for loop variable k. We then end both the **for** and **if** loop and mark our array of indices into x_i

- f usually is not a function of every component of x

- If $\frac{\partial f}{\partial x_k}(t,x,u) = 0$, then x_k was not an input variable of f.

$$f(t, x_{n_1}, \cdots, x_{n_d}, u)$$

4.3.3 Fwrap

Then using the results that we have received in our xeval function. We then input the results into another function called the F Wrapper function. Along with our other inputs f which will be our function, our array of u, our initial x_0. Then we make our matrix f into a matlab function and labeled it as Fmf, then we check witch variables in x_0 are non-zeros by plugging in our results from our xeval into x_0 called nx. Once we have our variables needed we then make them into a numeric array that represents its numbers. Lastly we evaluate our variables into our matlabfucntion Fmf and store the results into Fn.

- Where x_{n_j} for $j = 1, \cdots, d$ are the state variables that f_x explicitly depends on.

- We needed to automatically determine which indices of x are kept.

- We kept the indices if $\frac{\partial f}{\partial x_j} \neq 0$

- This was determined using the symbolic `diff` command and comparing it to a symbolic 0

4.3.4 Leval

Almost similar to our xeval function by having the same purpose by evaluating our function to see which indices our non-zero. Except instead of just for one variable we are doing it for two variables. Starting with our inputs H_l which is our Hamiltonian but differentiated with respect to lambda and m in which it will be our length of x_0. We create two real matrices both x and λ both being the size of m by 1. In one **for** with k from 1 to the length of x, we get the sum of the differation of the Hamiltonian with respect to λ labled l and in another **for** loop with k from 1 to the length of λ we get the sum differation of the Hamiltonian with respect to x. We then create a

symbolic zero labled z and creat a numeric zero called j for the first **if** loop. Similar to what we did in the xeval functionwe just check if what variables in the array are zero or non-zero and store those variables into another variable. These being l_i for the indices of λ then we clear indices to have a clean state. Then repeat for x and store them into x_i.

- $H_x = \nabla_x H$
- If $\frac{\partial H_x}{\partial x_k} = 0$, then x_k was not an input variable of H_x.

$$H_x(t, x_{n_1}, \cdots, x_{n_d}, u, \lambda_{m_1}, \cdots, \lambda_{m_c})$$

- Where x_{n_j}, for $j = 1, \ldots, d$, and λ_{m_k}, where $k = 1, \ldots, c$, are the state and adjoint variables that H_x explicitly depends on

4.3.5 lwrap

The following is our L wrapper or lwrap, it serves the same purpose as our Fwrap but again instead of one variable, it will be with two variables which it will be our x and λ. With inputs being the H_l =, our initial variables λ_0, u_0,x_0 and our results from our leval function x_i and l_i. First we create a matlab function of H_l. Then we evaluate our initial values λ_0 and x_0 with x_i and λ_0 with our l_i. We then make them into character arrays that represent the the numbers, both labled *lnc* and *xnc*. once we have our arrays then we put the results into our matlabfuncion Hlanon with inputs *lnc*, u_0, and finally *xnc*, giving us our numeric λ.

- We needed to automatically determine which indices of λ and x are kept.
- We kept the indices if $\frac{\partial H_x}{\partial x_j} \neq 0$ and if $\frac{\partial H_x}{\partial \lambda_k} \neq 0$
- This was determinded using the symbolic `diff` command and comparing it to a symbolic 0
- Lastly, to evaluate at each step of R.K., we extracted the indices that we determined H_x explicitly uses of x and λ, then convert, via `num2cell`, those entries into cellular lists.

4.3.6 Ueval

The U evaluation code is similar as the L evaluation code. With only the difference being the inputs of u and m. And the differation **for** loops being the derivative of u with respect to λ and with respect to x. We then labeled them as *ul* and *ux*. As same as before we make a symbolic zero z and make numeric zero labeled j and first check for non-zero indices for lambda of *ul* and store them into a variable uli and then we clear the indices so we do not have the previous indices and repeat for the varaibles of x of *ux* and store them into *uxi*. We then have our outputs of uli and *uxi*

- Solving $\frac{\partial H}{\partial u}(t,u,x,\lambda) = 0$ for u^*
- If $\frac{\partial H_u}{\partial x_k}(t,u,x,\lambda) = 0$, then x_k was not an input variable of H_u.

$$H_u(t, x_{n_1}, \cdots, x_{n_d}, u, \lambda_{m_1}, \cdots, \lambda_{m_c})$$

- Where x_{n_j} for $j = 1, \cdots, d$ and λ_{m_k} where $k = 1, \cdots, c$ are the state and adjoint variables that H_u explicitly depends on.
- We needed to automatically determine which indices of λ and x are kept.
- We kept the indices if $\frac{\partial H_u}{\partial x_j} \neq 0$ and if $\frac{\partial H_u}{\partial \lambda_k} \neq 0$
- This was determined using the symbolic `diff` command and comparing it to a symbolic 0.
- Lastly to evaluate H_u for our solve step, we extracted the indices that we determined H_u explicitly uses of x and λ then convert, via `num2cell`, those entries into cellular lists.

4.3.7 Trapmethod

Finally one of of last subcodes that we have created is the Trapezodial method. Where it takes inputs of f, a, and b. Where we have the length of f minus one labeled n, this is to subtract the length of our function by 1 and by getting the difference of the upper and lower limit labeled a and b we then divide the result by n and store it into h. We then create a starting point zero labeled s

and in a **for** loop of i starting from two to n we add s with the arrays of f and store them into s. We then have our formula for the sum of the 1st entry of f and the last entry of f and the product of two multiplied by s. and the result of that is then multiplied by h over two and store our final resulet into T.

- To compute the $\int_{t_0}^{t_f} f(t,u,x)dt$ we used a composite Trapezoidal method

$$\int_{t_0}^{t_f} f(t,u,x)dt \approx \sum_{j=0}^{N_t} \omega_j\, f(t_j,u_j,x_j).$$

- where $\omega_j = \frac{1}{2}(\frac{t_f - t_0}{N_t}) \quad j = 0, n.$
- and $\omega_j = \frac{t_f - t_0}{N_t}, \quad j = 1, \ldots, n-1.$

CHAPTER V

RESULTS

The Following are the results from each of the problems that we have used in the previous chapters. Not that the Red-Dotted line is what we were able to produce, while the solid black line is what the book was able to produce.

5.1 Max 1-dimensional case

These results were from the book by Lenheart [6] as we did in chapter 2 as the first example 2.1. Notice in the Graphs of Time vs Optimal and Time vs Control we have a decreasing effect while the Time vs State has a growth effect. This shows that U or our control begins strong pushing our x or our state upwards. As control decreases so does our optimal function, and our state begins to decrease towards the end

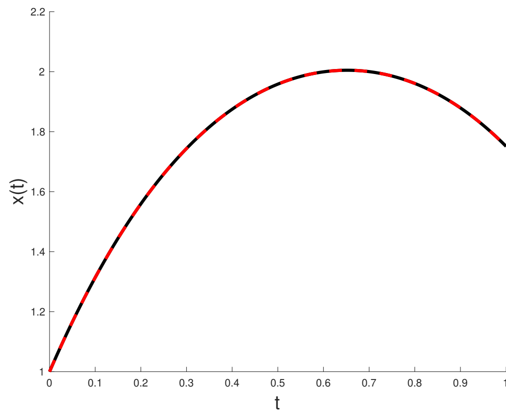

Figure 5.1: Time Vs X

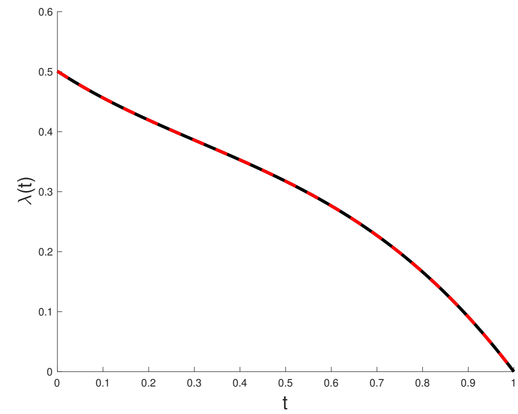

Figure 5.2: Time vs Lambda

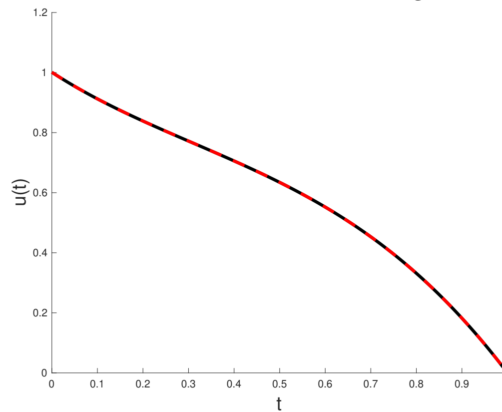

Figure 5.3: Time vs U

5.2 Bounded case

This is from when Our F function is max $F = Ax(t) - u(t)^2 \; dt$

, and our G function is $G = \; x'(t) \; = -\frac{1}{2} \, x(t)^2 \; + \; Cu(t)$. Similar to the maximum one dimension case. Where our control begins strong pushing our state upwards. As control decreases so does our optimal function, and our state begins to decrease towards the end

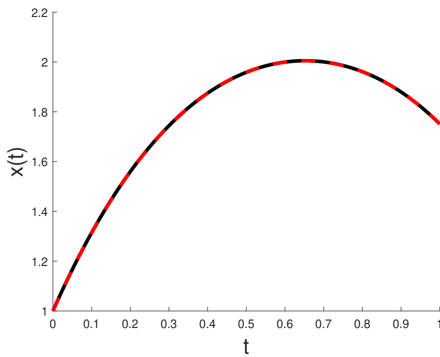

Figure 5.4: Time Vs X

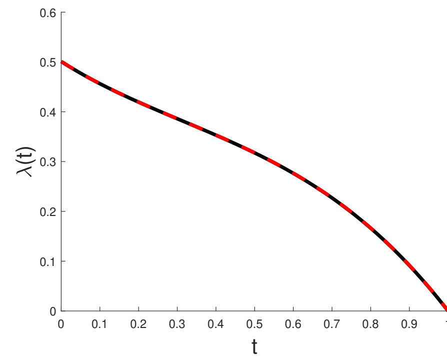

Figure 5.5: Time vs Lambda

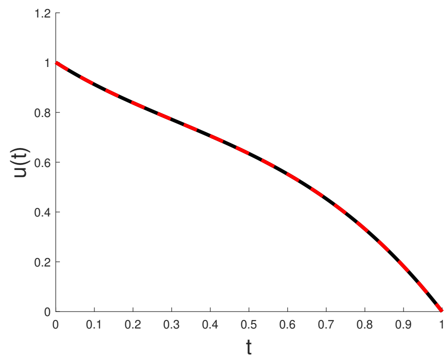

Figure 5.6: Time vs U

5.3 Multi-Dimensional case

In this section we will be showing results from both types of the regular multi-dimensional case and our greedy multi dimensional case. The results are from the SERIN model that we have explained from earlier.

5.3.1 Regular

For the multi-dimensional case, we ran at a 1000 iterations and a maxiter of 40, it was a bit longer than previous cases. But notice that for our control which represents the the vaccination, showing that the more people were vaccinated the less infections occurred. Our optimal shows that some variables start contestant at different starting points but eventually lead to 20.

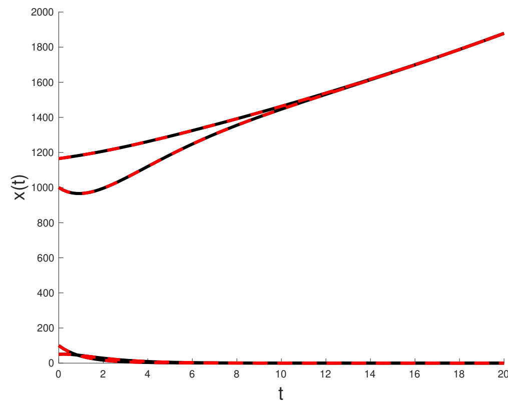

Figure 5.7: Time Vs X

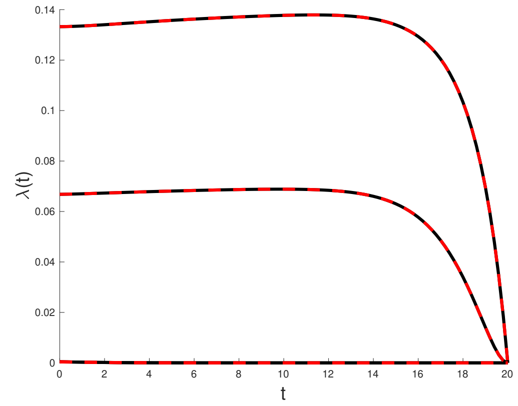

Figure 5.8: Time vs Lambda

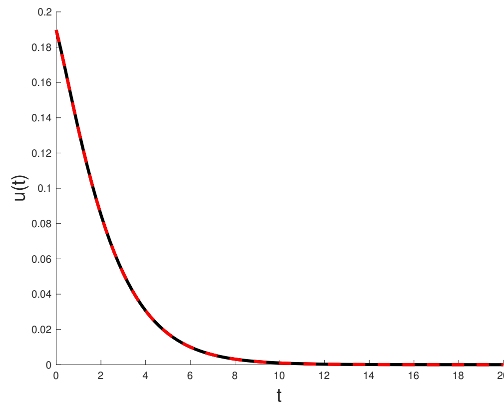

Figure 5.9: Time vs U

5.3.2 Greedy

For the Greedy method the results were similar but were achieved in a quicker Pace

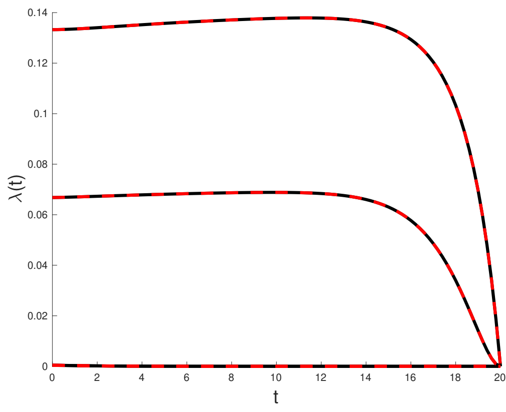

Figure 5.11: Time vs X

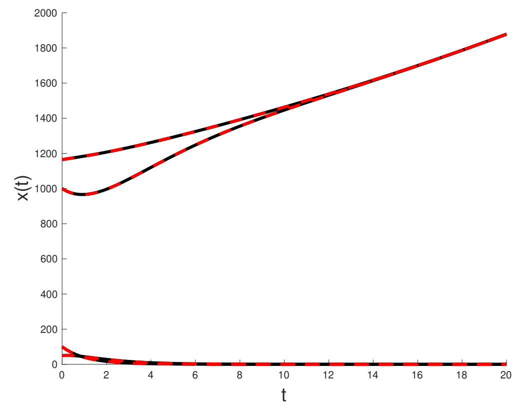

Figure 5.10: Time Vs Lambda

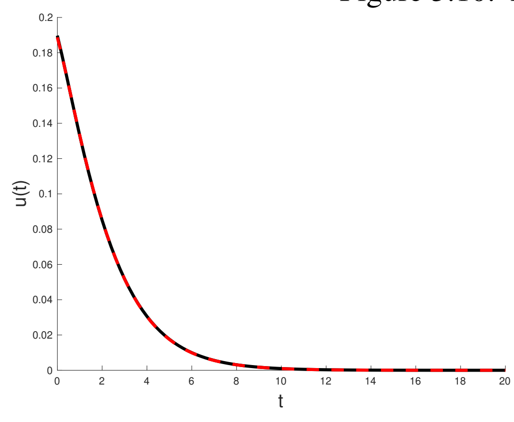

Figure 5.12: Time vs U

5.4 Multi-Dimensional case HIV Treatment

In this section we will be showing results for the regular multi-dimensional case and our greedy multi dimensional case for the HIV Treatment

5.4.1 Regular

For the multi-dimensional case, we ran at a 1000 iterations and a maxiter of 40, it was a bit longer than previous cases to run.

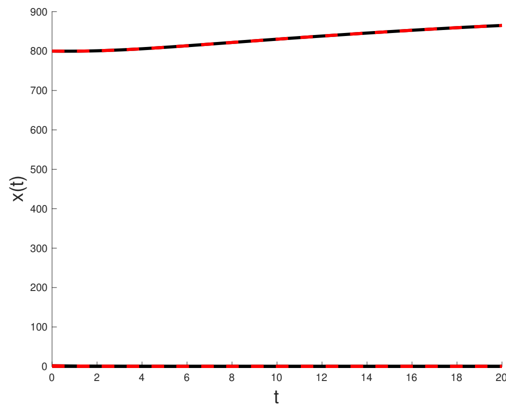

Figure 5.13: Time Vs X

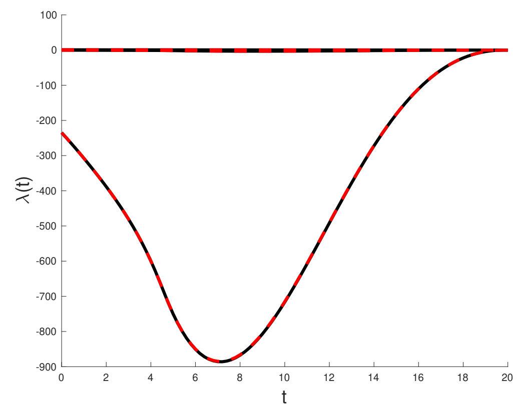

Figure 5.14: Time vs Lambda

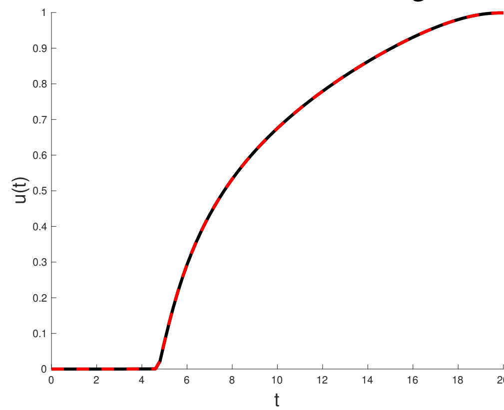

Figure 5.15: Time vs U

5.4.2 Greedy

For the Greedy method, same results were produced, but in a quicker pace.

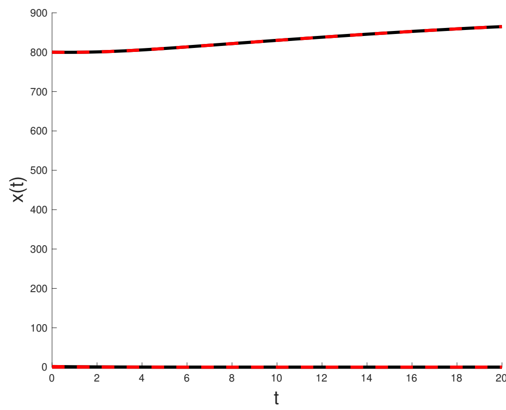

Figure 5.16: Time Vs X

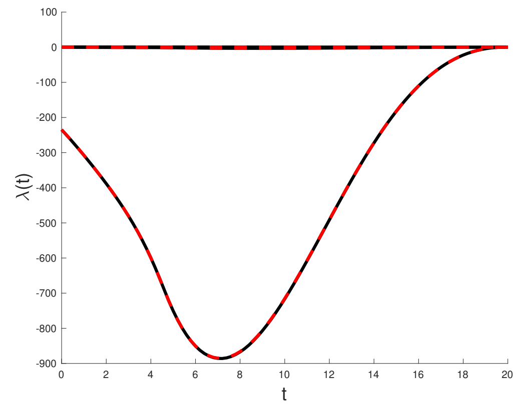

Figure 5.17: Time vs Lambda

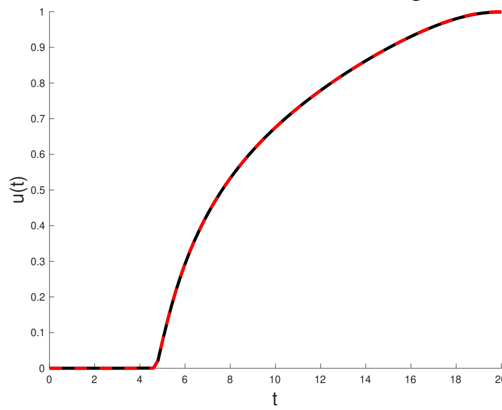

Figure 5.18: Time vs U

CHAPTER VI

CONCLUSION

After constructing and implementing the automatic solver into different cases,we can conclude that it functions for a one dimensional maximum case, a one dimensional bounded case and in the Multi-Dimensional Optimal Control case, as long the user implements the identity matrix to the size of $x0$ it and implements the G function or the state variable into a matrix form, as well if the objective function is minimized then the user must also put a negative sign before inputting the objective function, it should function as well.

This was not the first attempted in creating a automatic solver for optimal control problems. There have been other attempted into such coding such as ICLOCS2 by Yuanbo Nie [7]. Where it is a promising optimal control problem solver that offers advanced features, robust numerical techniques, and user-friendly interfaces. Its open-source nature, computational efficiency, and broad applicability make it a valuable tool for researchers and practitioners in the field of optimal control. Another attempted is GPOPS-II by Michael A. Patterson [3]. It can solve multiple-phase optimal control problems and the integration of hp-adaptive Gaussian quadrature collocation methods and sparse nonlinear programming techniques makes GPOPS II a powerful tool of use in solving optimal control problems,but this requires a fee depending on use such as government,university, academic and single use, in which the fee will vary. Lastley using Hybrid Runge-Kutta methods researched by Moosa Ebadi [1], this was approach combines the shooting method with the finite difference approximation and leverages the benefits of hybrid Runge-Kutta methods to enhance accuracy and computational efficiency.

This Research was inspired by Dr. Kristina P. Vatcheva and along with other professors in a research article called Social distancing and testing as optimal strategies against the spread

of COVID-19 in the Rio Grande Valley of Texas [2]. In this article, researchers were solving an optimal control problem related to COVID-19 by hand, when my advisor Josef Sifuentes thought an idea of an automatic solver for optimal control problems.

REFERENCES

[1] Moosa Ebadi. Fbsm solution of optimal control problems using hybrid runge-kutta based methods. *Journal of Mathematical Extension[pgs 1-35]*, 2021.

[2] Tamer Oraby Jose Campo Maldonado Timothy Huber María Cristina Villalobos Kristina P. Vatcheva, Josef Sifuentes. Social distancing and testing as optimal strategies against the spread of covid-19 in the rio grande valley of texas. 2021.

[3] Michael A. Patterson. Gpops ii: A matlab software for solving multiple-phase optimal control problems using hp-adaptive gaussian quadrature collocation methods and sparse nonlinear programming. *ACM Transactions on Mathematical Software, Vol. 41, No. 1, Article 1*, 2014.

[4] Halsey Royden. *Real Analysis*. Prentice Hall, Boston, 2010.

[5] ELINA SEPPÄLÄ. Masters thesis, a greedy update to an optimal control algorithm, 2023.

[6] John T Workman Suzanne Lenhart. *Optimal Control Applied to Biological Models*. Taylor and Francis, 2007.

[7] Eric C. Kerrigan Yuanbo Nie, Omar Faqir. Iclocs2: Try this optimal control problem solver before you try the rest. *UKACC 12th International Conference on Control (CONTROL)*, 2018.

APPENDIX A

APPENDIX A

APPENDIX

1.1 The Lebesgue Dominated Convergence Theorem

As stated from Royden [4], here is another version of the lebesgue dominated convergence theorem.

Let f_n be a sequence of measurable functions on E. Suppose there is a function g hat is integrable over E and dominates f_n on E in the sense that $|f_n| \leq g$ on E for all n.

$$\text{if } f_n \rightarrow f \text{ pointwise a.e. on } E, \text{then } f \text{ is integral over } E \text{ and} \lim_{n\to\infty} \int_E f_n = \int_E f$$

Proof Since $|f| \leq g$ on E and g is intergrable over E, then by the intergrable comparison test, f and each f_n are also integrable over E. Since f is intertable over E, then f is finite a.e on E. Which makes the excising from E a countable collection of sets of measure zero and using the countable addivitity of Lebesgue measure, we can assume that f and f_n is fintite on E. Now the function $g-f$ and for each n, the funtion $g-f_n$, are defined, nonnegative and measuravble. as well the sequence $|g-f_n|$ covnerges pointwie a.e on E to $g-f$. By Fatou's Lemma it tells us that

$$\int_E (g-f) \leq \lim inf \leq_E (g-f_n)$$

. Thus, by the linearity of integration for integrable functions,

$$\int_E g - \int_E f = \int_E (g - f_{\leq} \lim inf \int_E (g-f_n) = \int_E g - \lim sup \int_E f_n$$

, that is,

$$\lim sup \int_E f_n \leq \int_E f$$

. Similarly, considering the sequence $\{g + f_n\}$, we have recived the following

$$\int_E f \leq \lim inf \int_E f_n$$

. Hence Proof. □

1.2 One Dimensional

1.2.1 Maximum One Dimensional Code

```
function [x,lambda,u] = OptControl5(F,G,x0,t0,tf,maxiter,Nt)

test = -1;

delta = 0.001;

N = Nt;

tn = linspace(t0,tf,N+1);

iter = 0;

u0 = zeros(1,N+1);

xn = zeros(1,N+1);

xn(1) = x0;

lambda0 = zeros(1,N+1);

Ganon = matlabFunction(G);

Ganon2 = @(t,x,u) Ganon(u,x);

Fanon= matlabFunction(F);

Fanon2 = @(t,x,u,lambda) Fanon(lambda,x);

syms t x u

syms lambda

H = F + lambda.*G
```

```
Hanon = matlabFunction(H);
Hanon2 = @(t,x,u,lambda) Hanon(lambda,u,x);
Hx = diff(Hanon2,lambda);
Hl = diff(Hanon2,x);
Hu = diff(Hanon2,u);
Hlanon = matlabFunction(Hl);
Hlanon2 = @(t,x,u,lambda) Hlanon(x,lambda);
usol = solve(  Hu == 0 ,u)
uanon = matlabFunction(usol)
uanon2 = @(lambda) uanon(lambda)
   oldu = u0;
   oldx = xn;
   oldlambda = lambda0;
while (test < 0 && iter < maxiter )
    iter = iter +1;
    x = RKXU(Ganon2,x0,t0,tf,N,oldu);
    lambda = transversecond(Hlanon2,x,oldu,tf,N);
    u1 = uanon2(lambda);
    u = 0.5*(u1 + oldu);
    temp1 = delta * sum(abs(u))-sum(abs(oldu-u));
    temp2 = delta * sum(abs(x))-sum(abs(oldx-x));
    temp3 = delta * sum(abs(lambda))- ...
        sum(abs(oldlambda-lambda));
    test = min(temp1,min(temp2,temp3)) ;
    oldu = u;
    oldx = x;
    oldlambda = lambda;
```

```
end
y(1,:) = tn;
y(2,:) = x;
y(3,:) = lambda;
y(4,:) = u;
figure(7)
plot(tn,x)
title('Time vs X')
xlabel('Time')
ylabel('X')
figure(8)
plot(tn,u)
title('Time vs U')
xlabel('Time')
ylabel('U')
figure(9)
plot(tn,lambda)
title('Time vs Lambda')
xlabel('Time')
ylabel('Lambda')
```

1.2.2 Bounded Case Code

```
function [x,lambda,u] = OptControl8(F,G,x0,t0,tf,maxiter,Nt,M1,M2)
test = -1;
delta = 0.001;
N = Nt;
tn = linspace(t0,tf,N+1);
iter = 0;
```

```
u0 = zeros(1,N+1);
xn = zeros(1,N+1);
xn(1) = x0;
lambda0 = zeros(1,N+1);
Ganon = matlabFunction(G);
Ganon2 = @(t,x,u) Ganon(u,x);
Fanon= matlabFunction(F);
Fanon2 = @(t,x,u,lambda) Fanon(lambda,x);
syms t x u
syms lambda
H = F + lambda.*G;
Hanon = matlabFunction(H);
Hanon2 = @(t,x,u,lambda) Hanon(lambda,u,x);
Hx = diff(Hanon2,lambda);
Hl = diff(Hanon2,x);
Hu = diff(Hanon2,u);
Hlanon = matlabFunction(Hl);
Hlanon2 = @(t,x,u,lambda) Hlanon(x,lambda);
usol = solve(  Hu == 0 ,u);
uanon = matlabFunction(usol);
uanon2 = @(lambda) uanon(lambda);
   oldu = u0;
   oldx = xn;
   oldlambda = lambda0;
while (test < 0 && iter < maxiter )
    iter = iter +1;
    x = RKXU(Ganon2,x0,t0,tf,N,oldu);
```

```
    lambda = transversecond(Hlanon2,x,oldu,tf,N);
    u1 = min(M2, max(M1,uanon2(lambda)));
    u = 0.5*(u1 + oldu);
    temp1 = delta * sum(abs(u))-sum(abs(oldu-u));
    temp2 = delta * sum(abs(x))-sum(abs(oldx-x));
    temp3 = delta * sum(abs(lambda))- ...
        sum(abs(oldlambda-lambda));
    test = min(temp1,min(temp2,temp3)) ;
    oldu = u;
    oldx = x;
    oldlambda = lambda;
end
y(1,:) = tn;
y(2,:) = x;
y(3,:) = lambda;
y(4,:) = u;
figure(7)
plot(tn,x)
title('Time vs X')
xlabel('Time')
ylabel('X')
figure(8)
plot(tn,u)
title('Time vs U')
xlabel('Time')
ylabel('U')
figure(9)
```

```
plot(tn,lambda)
title('Time vs Lambda')
xlabel('Time')
ylabel('Lambda')
```

1.3 Multi-Dimensional

1.3.1 Multi-Dimensional Code

```
function [x, lambda, u] = multiOptControl(F,G,x0,t0,tf,maxiter,Nt,phi,M1,M2)
test = -1;
delta = 0.001;
tn = linspace(t0,tf,Nt+1);
iter = 0;
u0 = zeros(1,Nt+1);
xn = zeros(length(x0),Nt+1);
for i = 1:length(x0)
    xn(i) = x0(i);
end
lambda0 = zeros(length(x0),Nt+1);
syms t u
syms x [size(x0)] matrix
syms lambda [size(x)] matrix
Gmatrix = symmatrix2sym(G);
Gmf = matlabFunction(Gmatrix);
Ganon = @(u,x) geval(Gmf,u,x);
Ganon2 = @(t,x,u) Ganon(u,x);
Fmatrix = symmatrix2sym(F);
Fanon= matlabFunction(Fmatrix);
```

```
Fanon2 = @(t,x,u) Fanon(u,x);
if phi == 0
    Lfxanon = 0;
elseif phi ~= 0
    Lfanon = matlabFunction(phi);
    Lfanon2 = @(x) Lfanon(x);
    Lfx = diff(Lfanon2,x);
    Lfxanon = matlabFunction(Lfx);
    Lfxanon2 = @(x) Lfxanon(x);
end
H = F + lambda.'*G;
Hl = gradient(H,x);
Hlmatrix = symmatrix2sym(Hl);
[xi, li] = leval(Hlmatrix,length(x0));
Hlanon = @(t,x,u,lambda) lwrap(Hlmatrix,lambda,u,x,xi,li);
Hu = gradient(H,u);
Humatrix = symmatrix2sym(Hu);
usol = solve( Humatrix == 0 ,u);
[uli, uxi] = ueval(usol,length(x0));
uanon = @(lambda,x) uwrap(usol,lambda,x,uli,uxi);
   oldu = u0;
   oldx = xn;
   oldlambda = lambda0;
while (test < 0 && iter < maxiter )
    iter = iter +1
    x = RKXU(Ganon2,x0,t0,tf,Nt,oldu);
    if Lfxanon == 0
```

```
        lambdaf = Lfxanon;
    elseif Lfxanon ~= 0
        if nargin(Lfxanon) == 0
            lambdaf = feval(Lfxanon);
        else
            lambdaf = Lfxanon2(x(:,end));
        end
    end
    lambda = transversecondN(Hlanon,x,oldu,tf,Nt,lambdaf);
    u1 = min(M2, max(M1,uanon(lambda,x)));
    u = 0.5*(u1 + oldu);
    temp1 = delta * sum(abs(u),2)-sum(abs(oldu-u),2);
    temp2 = delta * sum(abs(x),2)-sum(abs(oldx-x),2);
    temp3 = delta * sum(abs(lambda),2)- ...
        sum(abs(oldlambda-lambda),2);
    test = min( [temp1; temp2; temp3] ) ;
    oldu = u;
    oldx = x;
    oldlambda = lambda;
end
figure(7)
plot(tn,x)
title('Time vs X')
xlabel('Time')
ylabel('X')
figure(8); hold on
plot(tn,u);
```

```
title('Time vs U')
xlabel('Time')
ylabel('U')
figure(9)
plot(tn,lambda)
title('Time vs Lambda')
xlabel('Time')
ylabel('Lambda')
```

1.3.2 Multi-Dimensional Greedy Code

```
function [x, lambda, u] = multiOptControlGreedy(F,G,x0,t0,tf,maxiter,Nt,phi,M1,M2)
test = -1;
delta = 0.001;
tn = linspace(t0,tf,Nt+1);
iter = 0;
Nc = 10;
u0 = zeros(1,Nt+1);
xn = zeros(length(x0),Nt+1);
for i = 1:length(x0)
    xn(i) = x0(i);
end
lambda0 = zeros(length(x0),Nt+1);
syms t u
syms x [size(x0)] matrix
syms lambda [size(x)] matrix
Gmatrix = symmatrix2sym(G);
Gmf = matlabFunction(Gmatrix);
Ganon = @(u,x) geval(Gmf,u,x);
```

```
Ganon2 = @(t,x,u) Ganon(u,x);
Fmatrix = symmatrix2sym(F);
[xi] = xeval(Fmatrix,length(x0));
Fanon2 = @(x,u) fwrap(Fmatrix,u,x,xi);
if phi == 0
    Lfxanon = 0;
elseif phi ~= 0
    Lfanon = matlabFunction(phi);
    Lfanon2 = @(x) Lfanon(x);
    Lfx = diff(Lfanon2,x);
    Lfxanon = matlabFunction(Lfx);
    Lfxanon2 = @(x) Lfxanon(x);
end
H = F + lambda.'*G;
Hl = gradient(H,x);
Hlmatrix = symmatrix2sym(Hl);
[xi, li] = leval(Hlmatrix,length(x0));
Hlanon = @(t,x,u,lambda) lwrap(Hlmatrix,lambda,u,x,xi,li);
Hu = gradient(H,u);
Humatrix = symmatrix2sym(Hu);
usol = solve( Humatrix == 0 ,u);
[uli, uxi] = ueval(usol,length(x0));
uanon = @(lambda,x) uwrap(usol,lambda,x,uli,uxi);
   oldu = u0;
   oldx = xn;
   oldlambda = lambda0;
while (test < 0 && iter < maxiter )
```

```
iter = iter +1
x = RKXU(Ganon2,x0,t0,tf,Nt,oldu);
if Lfxanon == 0
    lambdaf = Lfxanon;
elseif Lfxanon ~= 0
    if nargin(Lfxanon) == 0
        lambdaf = feval(Lfxanon);
    else
        lambdaf = Lfxanon2(x(:,end));
    end
end
lambda = transversecondN(Hlanon,x,oldu,tf,Nt,lambdaf);
u1 = min(M2, max(M1,uanon(lambda,x)));
thindex = 0;
for th = (1/Nc):(1/Nc):1
    thindex = thindex+1;
    utry = (1-th)*u1 + th*oldu;
    x = RKXU(Ganon2,x0,t0,tf,Nt,utry);
    for k = 1:Nt+1
        Farray(k) = Fanon2(x(:,k),utry(k));
    end
    f(thindex) = Trapmethod(Farray,t0,tf);
end
[fmin, ind]  = min(f);
th = ind/Nc;
u = (1-th)*u1 + th*oldu ;
temp1 = delta * sum(abs(u),2)-sum(abs(oldu-u),2);
```

```
    temp2 = delta * sum(abs(x),2)-sum(abs(oldx-x),2);
    temp3 = delta * sum(abs(lambda),2)- ...
        sum(abs(oldlambda-lambda),2);
    test = min( [temp1; temp2; temp3] ) ;
    oldu = u;
    oldx = x;
    oldlambda = lambda;
end
figure(10)
plot(tn,x)
title(' Greedy Time vs X')
xlabel('Time')
ylabel('X')
figure(11); hold on
plot(tn,u);
title('Greedy Time vs U')
xlabel('Time')
ylabel('U')
figure(12)
plot(tn,lambda)
title('Greedy Time vs Lambda')
xlabel('Time')
ylabel('Lambda')
```

1.4 Sub-Codes

1.4.1 Forward-Backward Sweep Code

```
function x = RKXU(f,x0,t0,tf,N,u)
```

```
h = (tf-t0)/N;
h2 = h/2;
x = zeros(length(x0),N+1);
x(:,1) = x0;
for i = 1:N
    ti = t0 + h*i;
    k1 =f(ti, x(:,i), u(:,i));
    k2 =f((ti + h2),(x(:,i) + h2*k1) , (u(:,i)+ u(:,i+1))/2);
    k3 =f((ti + h2),(x(:,i) + h2*k2) , (u(:,i)+ u(:,i+1))/2);
    k4 =f((ti+h), (x(:,i)+h*k3) , u(:,i+1));
    x(:,i+1) = x(:,i) + (h/6)*(k1 + 2*k2 + 2*k3 + k4);
end
```

1.4.2 Transverse Condition Codes

Without Payoff Function Code

```
function lambda = transversecond(Hx,x,u,tf,N)
t0 = 0;
h = (tf-t0)/N;
h2 = h/2;
y = zeros(size(x));
y(:,1) = 0;
f = @(t,lambda,x,u) Hx(tf-t,x,u,lambda);
x = fliplr(x);
u = fliplr(u);
for k = 1:N
    tk = t0 + h*k;
    k1 = f((tk),(y(:,k)),(x(:,k)),(u(:,k)));
```

```
    k2 = f((tk+h2),(y(:,k)+h2*k1),((x(:,k)+x(:,k+1))/2),((u(:,k)+ u(:,k+1))/2));
    k3 = f((tk+h2),(y(:,k)+h2*k2),((x(:,k)+x(:,k+1))/2),((u(:,k)+ u(:,k+1))/2));
    k4 = f((tk+h2),(y(:,k)+h*k3),(x(:,k+1)),(u(:,k+1)));
    y(:,k+1) = y(:,k) + (h/6) *(k1 + 2*k2 + 2*k3 +k4);
end
lambda = fliplr(y);
```

With Payoff Function Code

```
function lambda = transversecondN(Hx,x,u,tf,N,phi)
t0 = 0;
h = (tf-t0)/N;
h2 = h/2;
y = zeros(size(x));
y(:,1) = phi;
f = @(t,lambda,x,u) Hx(tf-t,x,u,lambda);
x = fliplr(x);
u = fliplr(u);
for k = 1:N
    tk = t0 + h*k;
    k1 = f((tk),(y(:,k)),(x(:,k)),(u(:,k)));
    k2 = f((tk+h2),(y(:,k)+h2*k1),((x(:,k)+x(:,k+1))/2),((u(:,k)+ u(:,k+1))/2));
    k3 = f((tk+h2),(y(:,k)+h2*k2),((x(:,k)+x(:,k+1))/2),((u(:,k)+ u(:,k+1))/2));
    k4 = f((tk+h2),(y(:,k)+h*k3),(x(:,k+1)),(u(:,k+1)));
    y(:,k+1) = y(:,k) + (h/6) *(k1 + 2*k2 + 2*k3 +k4);
end
lambda = fliplr(y);
```

1.4.3 G Evaluation Code

```
function  Gn = geval(G,u,x)

xnc = num2cell(x);

Gn = G(u,xnc{:});
```

1.4.4 X Evaluation Code

```
function [xi] = xeval(f,m)

syms x [m 1] real

for k = 1:length(x)

    fx(k) = sum(diff(f,['x' num2str(k)]));

end

z = sym(0);

i = 0;

for k = 1:length(x)

    fxz = isequal(fx(k),z);

    if(fxz == 0)

        i = i+1;

        indices(i) = k;

    end

end

xi = indices;
```

1.4.5 F Wrapper Code

```
function Fn = fwrap(f,u,x0,xi)

Fmf= matlabFunction(f);

nx = x0(xi);

xnc = num2cell(nx);

Fn = Fmf(u,xnc{:});
```

1.4.6 L Evaluation Code

```
function [xi, li] = leval(Hl,m)

syms x [m 1] real

syms lambda [m 1] real

for k = 1:length(lambda)

    l(k) = sum(diff(Hl,['lambda' num2str(k)]));

end

for k = 1:length(x)

    lx(k) = sum(diff(Hl,['x' num2str(k)]));

end

z = sym(0);

j = 0;

for k = 1:length(lambda)

    lz = isequal(l(k),z);

    if(lz == 0)

        j = j+1;

        indices(j) = k;

    end

end

li = indices;

clear indices

i = 0;

for k = 1:length(x)

    xz = isequal(lx(k),z);

    if(xz == 0)

        i = i+1;

        indices(i) = k;
```

```
    end
end
xi = indices;
```

1.4.7 L Wrapper Code

```
function lambdan = lwrap(Hl,lambda0,u0,x0,xi,li)
Hlanon = matlabFunction(Hl);
    nl = lambda0(li);
    nx = x0(xi);
    lnc = num2cell(nl);
    xnc = num2cell(nx);
lambdan = Hlanon(lnc{:},u0,xnc{:});
```

1.4.8 U Evaluation Code

```
function [uli, uxi] = ueval(u,m)
syms x [m 1] real
syms lambda [m 1] real
for k = 1:length(lambda)
    ul(k) = diff(u,['lambda' num2str(k)]);
end
for k = 1:length(x)
```

```
    ux(k) = diff(u,['x' num2str(k)]);
end
z = sym(0);
j = 0;
for k = 1:length(lambda)
    ulz = isequal(ul(k),z);
    if(ulz == 0)
        j = j+1;
        Indices(j) = k;
    end
end
uli = Indices;
clear Indices
i = 0;
for k = 1:length(x)
    uxz = isequal(ux(k),z);
    if(uxz == 0)
        i = i+1;
        Indices(i) = k;
    end
end
uxi = Indices;
```

1.4.9 U Wrapper Code

```
function Un = uwrap(u,lambda0,x0,uli,uxi)
```

```
uslanon = matlabFunction(u);
for t_iter = 1:length(x0)
    nul = lambda0(uli,t_iter);
    nux = x0(uxi,t_iter);
    ulc = num2cell(nul);
    uxc = num2cell(nux);
Un(t_iter) = uslanon(ulc{:},uxc{:});
end
```

1.4.10 Trapezoidal Method Code

```
function T = Trapmethod(f,a,b)
n = length(f)-1;
h = (b-a)/n;
s = 0;
for i = 2:n
    s = s+f(i);
end
T = h/2*(f(1)+f(end)+2*s);
```

BIOGRAPHICAL SKETCH

The author Marcel E. Benitez was born in July 31, 1995, in Chicago, Illinois. He would later then moved to McAllen, Texas in 2000. He has two siblings, a late older brother Andree C. Benitez and a younger sibling Aimee E. Benitez with parents Miguel A. Benitez Jr. and Leonilla Benitez. To contact him, his email address will be mefren1599@aol.com.

From 2010-2014, he was enrolled in McAllen High school in Mcallen, Texas. After graduating in 2014, he would enroll to University of Texas Rio Grand Valley in Edinburg, Texas. He would be enrolled as an undergraduate student from 2014-2020. In 2020 he would then received a Bachelors in science in Applied Mathematics.

In 2020, he would continue his education at University of Texas Rio Grand Valley, He was then awarded the NSF S-STEM Mathematics Graduate Scholarship. He would enrolled as a graduate student from 2020-2023. In 2023, he would then received a Masters in science in Applied Mathematics from University of Texas Rio Grand Valley.