Bridgewater State University
Virtual Commons - Bridgewater State University

Honors Program Theses and Projects | Undergraduate Honors Program

5-12-2015

An Analysis of Numerical Methods on Traffic Flow Models

Terry Mullen

Follow this and additional works at: http://vc.bridgew.edu/honors_proj

Part of the Mathematics Commons

Recommended Citation
Mullen, Terry. (2015). An Analysis of Numerical Methods on Traffic Flow Models. In *BSU Honors Program Theses and Projects.* Item 105. Available at: http://vc.bridgew.edu/honors_proj/105
Copyright © 2015 Terry Mullen

This item is available as part of Virtual Commons, the open-access institutional repository of Bridgewater State University, Bridgewater, Massachusetts.

An Analysis of Numerical Methods on Traffic Flow Models

Terry Mullen

Submitted in Partial Completion of the
Requirements for Commonwealth Honors in Mathematics

Bridgewater State University

May 12, 2015

______________________________ ____________
Dr. Laura K. Gross, Thesis Director | Date

______________________________ ____________
Dr. Thomas Kling, Committee Member | Date

______________________________ ____________
Dr. Uma Shama, Committee Member | Date

An Analysis of Numerical Methods on Traffic Flow Models

By

Terry Mullen

Bridgewater State University

Bridgewater, MA

May 12, 2015

Acknowledgments

I would like to thank my mentor Dr. Laura Gross, for not only guiding me through this the process of research but also being so caring. Without her support, and knowledge this project would have never have been completed.

I would also like to thank my family for fully supporting me during my college career and listening to me drone on about traffic models for months.

An Analysis of Numerical Methods on Traffic Flow Models

Terry Mullen

Bridgewater State University
Bridgewater, MA
May 12, 2015

ABSTRACT

In this thesis, we implement Euler's method and the Runge-Kutta method to solve initial value problems. A goal of the project is to compare the two methods on preliminary problems illustrating limitations and advantages. We also apply the Runge-Kutta method to a mathematical model of traffic flow. This thesis sheds light on how the fourth-order Runge-Kutta method is implemented to solve the Optimal Velocity Model (Kurata & Nagatani, 2003). We identify initial conditions and base cases to run simulations of the model. We consider one-car and two-car systems to validate the application of the fourth-order Runge-Kutta method and the Optimal Velocity Model. Our simulations accurately capture practical traffic scenarios.

Table of Contents

1 Introduction **1**

2 Numerical Methods for Ordinary Differential Equations **2**

2.1 Euler's Method 3

2.1.1 Error Analysis in Euler's Method 5

2.2 Runge-Kutta 6

2.2.1 Error Analysis in the Runge-Kutta Method 9

3 Numerical Method for Systems and Higher-Order Methods **11**

3.1 Runge-Kutta on First-Order Systems 11

3.2 Runge-Kutta on Higher-Order Systems 13

4 Numerical Solution of a Traffic Model **15**

4.1 The Model 15

4.2 Implementation of Runge-Kutta on the Model 17

4.3 Simulation with a Stopped Object and One Car 18

4.4 Simulation with a Clear Road in Front of the Car 19

4.5 Simulation with an Object Traveling with a Slow Constant Speed 20

4.6 Simulation with a Stopped Object and Two Cars 22

4.7 Simulation with a Clear Road in Front of Two Cars 24

4.8 Simulation with an Object Traveling with a Slow Constant Speed and Two Cars 26

5 Conclusions **28**

6 Future Work **29**

References **30**

A Code Implementation **31**
A.1 Textbook Problem Codes 31
A.1.1 Euler's Method 31
A.1.2 Runge-Kutta Method 33
A.1.3 Runge-Kutta for Systems 34
A.2 Traffic Model Code 36
A.2.1 Traffic Model Equation Code 36
A.2.2 Runge-Kutta for Stopped Object and One Car 37
A.2.3 Runge-Kutta for a Clear Road in Front of the Car 40
A.2.4 Runge-Kutta for a Car Following an Object Traveling with a Slow Constant Speed 42
A.2.5 Runge-Kutta for Stopped Object and Two Cars 44
A.2.6 Runge-Kutta for a Clear Road in Front of Two Cars 47
A.2.7 Runge-Kutta for Two Cars Following an Object Traveling with a Slow Constant Speed 49

List of Figures

4.1 Position (left graph) and velocity (right graph) versus time for a car (solid line) approaching a stopped object (dashed line). 19

4.2 Position (left graph) and velocity (right graph) versus time for a car that starts out slow and accelerates because it has a clear path. 21

4.3 Position (left graph) and velocity (right graph) versus time for a car (solid line) with a slow moving constant speed object (dashed line) in front of it. 22

4.4 Position (left graph) and velocity (right graph) versus time for two cars (solid line and dashed line) approaching a stopped object. 24

4.5 Position (left graph) and velocity (right graph) versus time for two cars (solid line and dashed line) accelerating. 26

4.6 Position (left graph) and velocity (right graph) versus time for two car (solid line and dashed line) slowing down for a slow moving object (circle line). 27

List of Tables

2.1 Table for the Initial Value Problem 2.3 with values found by Euler's Method. . . . 4

2.2 Table for the Initial Value Problem 2.3 with values found by Euler's method and values found using the exact solution. . . . 4

2.3 Table for the Initial Value Problem 2.3 with values found by the Midpoint method and values found using the exact solution. . . . 7

2.4 Table for the Initial Value Problem 2.3 with values found by the fourth-order Runge-Kutta method and values found using the exact solution. . 8

2.5 Table to compare values of the Initial Value Problem 2.3 solved with Euler's method with $h = 0.025$, the Midpoint method with $h = 0.05$, and fourth-order Runge-Kutta with $h = 0.1$. . . . 10

3.1 Table for the Initial Value Problem 3.1 with values found by the fourth-order Runge-Kutta for systems and values found using the exact solution. 13

Chapter 1

Introduction

Numerical analysis is an important aspect of applied mathematics. Some equations can be solved using exact techniques, but other more complicated ones need to be solved using numerical estimation techniques. Many traffic modeling papers use numerical analysis to solve the models (Kurata & Nagatani, 2003; Chen, Peng, & Fang, 2014) but do not argue why they chose a specific method. They also do not fully explain the initial conditions or necessary base case conditions. There are a few ways the models could be solved. For example, some researchers use Euler's method (Chen et al., 2014). Others use Runge-Kutta (Kurata & Nagatani, 2003). For either method it is important that the reader understands the initial conditions and how the system is functioning.

Both methods are used to solve a differential equation with initial values by approximating points on the solution graph in a specific interval. A limitation of these numerical analysis techniques is the farther away from the initial value point the farther the approximation is from the actual value. Euler's method has a greater error than Runge-Kutta, but computations for Runge-Kutta are fairly complex. Euler's method has a greater error than the Runge Kutta method, but is useful as a stepping stone into more complex techniques. For more details on both methods and more comparisons of the methods see Chapter 2 and to see the MATLAB implementation see Appendix A.

Chapter 2

Numerical Methods for Ordinary Differential Equations

As previously stated, numerical methods like Euler's and Runge-Kutta approximate solutions to initial value problems on a specific interval:

$$\frac{dy}{dt} = f(t, y), \quad y(a) = \alpha, \quad a \leq t \leq b. \tag{2.1}$$

Both methods estimate the solution at N discrete points. The t values are evenly distributed through out the interval $[a, b]$. We use a step size

$$h = \frac{b - a}{N}. \tag{2.2}$$

Originating from the Taylor Series expansion, both methods rely on $f(t, y)$ to approximate the solution. (See Section 2.1.1.) The methods use a difference equation that uses the approximate solution w_i at current time step t_i to estimate the solution w_{i+1} at the next time step t_{i+1}. The difference equations produce N values $w_0, w_1, \ldots, w_i, \ldots, w_N$ that approximate $y_0, y_1, \ldots, y_i, \ldots, y_N$ at time $t_0, t_1, \ldots, t_i, \ldots, t_N$, respectively. It is a difference equation because it relies on the difference quotient $\frac{w_{i+1} - w_i}{h}$ $(h = t_{i+1} - t_i)$ as an approximation of $f(t, y)$. To show how these methods work we will use the following example problem (Burden & Faires, 2003):

$$y' = 1 + y/t, \ 1 \leq t \leq 2, \quad y(1) = 2, \ \text{with } h = 0.25. \tag{2.3}$$

Let us look now at two specific methods.

2.1 Euler's Method

We will define w_i to be the approximate solution of Initial Value Problem 2.1 at mesh point i, while t_i will be the t value at the ith mesh point. Euler's method generates the approximate solution point (t_i, w_i) using the following method:

$$t_0 = a, \quad w_0 = \alpha,$$

$$w_{i+1} = w_i + hf(t_i, w_i), \quad i = 0, 1, \ldots, N-1, \tag{2.4}$$

$$t_{i+1} = t_i + h,$$

where α is the initial value of the exact solution y as given in Equation 2.1. Equation 2.4 is the difference equation of Euler's method (Burden & Faires, 2003). The method yields N points (t_i, w_i).

Now let's look at the Example Problem 2.3. From the problem we can see, $f(t, w) = 1 + w/t$. Following the algorithm,

$$t_0 = 1, \quad w_0 = 2.$$

Then we must compute the difference equations

$$w_1 = w_0 + hf(t_0, w_0) = 2 + .25(1 + 2/1) = 2.750000, \quad t_1 = t_0 + h = 1.25,$$

$$w_2 = w_1 + hf(t_1, w_1) = 2.75 + .25(1 + 2.75/1.25) = 3.550000, \quad t_2 = t_0 + 2h = 1.5,$$

$$w_3 = w_2 + hf(t_2, w_2) = 3.55 + .25(1 + 3.55/1.5) = 4.391667, \quad t_3 = t_0 + 3h = 1.75,$$

$$w_4 = w_3 + hf(t_3, w_3) = 4.391667 + .25(1 + 4.391667/1.75) = 5.269047, \quad t_4 = t_0 + 4h = 2.$$

Therefore, Euler's method on Initial Value Problem 2.3 gives the approximation shown in Table 2.1.

Table 2.1: Table for the Initial Value Problem 2.3 with values found by Euler's Method.

i	t_i	w_i
0	1	2
1	1.25	2.750000
2	1.5	3.55
3	1.75	4.391667
4	2	5.269048

Being a problem from a textbook, Initial Value Problem 2.3 is a simple problem. In fact it has an exact solution. Using the linear equation method of differential equations you can find the exact solution to be $y(t) = t(\ln|t| + 2)$. Now we can assess how well Euler's method approximates the points on the solution curve. By adding a column with the exact values at the t steps to Table 2.1, we can also present the error $|y(t_i) - w_i|$. See Table 2.2.

Table 2.2: Table for the Initial Value Problem 2.3 with values found by Euler's method and values found using the exact solution.

i	t_i	w_i	$y(t_i)$	error
0	1	2	2	0
1	1.25	2.7500	2.7789	0.0289
2	1.5	3.55	3.6082	0.0582
3	1.75	4.3916	4.4793	0.0877
4	2	5.2690	5.3863	0.1172

As you can see the error increases as we step farther away from the initial time.

Error analysis is an important part of evaluating an numerical method like Euler's Method. Burden and Faires (2003) describe how the error in Euler's method behaves.

2.1.1 Error Analysis in Euler's Method

Because numerical methods like Euler's Method do not find exact solutions error becomes a large part of how we choose the method to use. Many of the methods are based on the Taylor series expansion of $y(t)$ about the point $t = t_0$. We know in Euler's method $t_i = t_0 + ih$. The nth degree Taylor polynomial has the form

$$P_n(t_i) = y(t_0) + y'(t_0)ih + \frac{y''(t_0)}{2!}i^2h^2 + \ldots + \frac{y^{(n)}(t_0)}{n!}i^nh^n. \tag{2.5}$$

The formula is using the same step size h as Euler's method. In the Taylor series n goes to infinity. The nth Taylor polynomial truncates the series to end at the $(n+1)$st term. When the series is truncated there is some error between the Taylor polynomial and $y(t)$. The error is called the remainder term or truncation error:

$$R_n(t_i) = \frac{y^{(n+1)}(\xi(t))}{(n+1)!}i^{n+1}h^{n+1}. \tag{2.6}$$

It is this term that helps us determine the error of a numerical method. Notice for Euler's method, since it uses the Taylor polynomial of order 1, the remainder term is

$$R_n(t_i) = y'(\xi(t))ih \tag{2.7}$$

Notice as the step size h decreases the remainder term and thus the error decreases as well so long as $y'(t)$ is bounded by M on $[a, b]$. This is one method to decrease error. The smaller the step size the more computations a computer will have to do. Because the remainder term is on the order of h we say that Euler's method is of order h denoted $\mathcal{O}(h)$.

2.2 Runge-Kutta

Runge-Kutta does a better job of estimation than Euler's method. Runge-Kutta has many different forms with the simplest being the second-order Midpoint method. We let the second-order Taylor Polynomial be estimated by the function

$$a_1 f(t+\alpha_1, y+\beta_1). \tag{2.8}$$

When solved using expanded Taylor Polynomials we get

$$a_1 = 1, \alpha_1 = \frac{h}{2}, \beta_1 = \frac{h}{2}.$$

The Midpoint method is

$$w_0 = \alpha, \tag{2.9}$$

$$w_{i+1} = w_i + hf(t_i + \frac{h}{2}, w_i + \frac{h}{2} f(t_i, w_i)), \text{for } i = 0, 1, \ldots, N-1. \tag{2.10}$$

Equation 2.10 is the difference equation. The Midpoint method uses the step size given in Equation 2.2. Just like Euler's method we iterate through the method finding w_i until we reach the end of the interval. Let's look at an example.

We will use the same example from Burden and Faires (2003) that we used to demonstrate Euler's Method, Equation 2.3. This time we will use the Midpoint method. From Equation 2.3 we see that $w_0 = 2$ and we can calculate $h = 0.25$. Using Equation 2.10 we find that the Midpoint method difference equation for this problem is

$$w_{i+1} = \frac{(4i^2 + 38i + 90)w_i + i^2 + 9i + 20}{4i^2 + 34i + 72}.$$

Now we can find the the values at each mesh point:

$$w_1 = \frac{(4(0)^2 + 38i + 90)2 + (0)^2 + 9(0) + 20}{4(0)^2 + 34(0) + 72} = 2.777778,$$

$$w_2 = \frac{(4(1)^2 + 38i + 90)2.7777778 + (1)^2 + 9(1) + 20}{4(1)^2 + 34(1) + 72} = 3.606060.$$

The process is continued for all N points as seen in Table 2.3. The exact solution for this problem is $y(t) = t(\ln|t| + 2)$. Using it we can see how the error changes as the we get closer to the end of the interval and farther away from the initial point.

Table 2.3: Table for the Initial Value Problem 2.3 with values found by the Midpoint method and values found using the exact solution.

i	t_i	w_i	$y(t_i)$	error
0	1	2	2	0
1	1.25	2.77778	2.7789	0.00115
2	1.5	3.60606	3.6082	0.00214
3	1.75	4.47630	4.4793	0.00303
4	2	5.38243	5.3863	0.00385

The Runge-Kutta method that is most popularly used is the fourth-order Runge-Kutta method. It is more accurate than the Midpoint method but is more complex. Again, Runge-Kutta uses the same step size as Equation 2.2, and α is the initial value of y. The method is

$$t_0 = a \quad w_0 = \alpha,$$

$$k_1 = hf(t_i, w_i),$$

$$k_2 = hf(t_i + \frac{h}{2}, w_i + \frac{1}{2}k_1),$$

$$k_3 = hf(t_i + \frac{h}{2}, w_i + \frac{1}{2}k_2),$$

$$k_4 = hf(t_i + h, w_i + k_3),$$

$$w_{i+1} = w_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) \quad i = 1, 2, \ldots, N-1. \tag{2.11}$$

$$t_i = a + ih$$

We can use fourth-order Runge-Kutta method on our Example Problem 2.3 on page 2. The first iteration is as follows:

$$w_0 = 2,$$

$$k_1 = 0.25f(1, 2) = 0.25(1 + 2/1) = 3/4 = 0.750000,$$

$$k_2 = 0.25f(1{+}0.125, 2{+}0.5(0.75)) = 0.25f(1.125, 2.375) = 0.25(1{+}2.375/1.125) = 0.777778,$$

$$k_3 = 0.25f(1.125, 2{+}0.5(0.777778)) = 0.25f(1.125, 2.388889) = 0.25(1{+}2.388889/1.125) = 0.780642,$$

$$k_4 = 0.25f(1.25, 2 + 0.78064) = 0.25(1 + 2.78064/1.25) = 0.806173,$$

$$w_1 = 2 + 1/6(0.75 + 0.777778 + 0.780642 + 0.806173) = 2.778909.$$

This whole process is repeated until t_i is at the end of the interval. As we have done be for the following table will compare this methods solutions with the exact solutions.

Table 2.4: Table for the Initial Value Problem 2.3 with values found by the fourth-order Runge-Kutta method and values found using the exact solution.

i	t_i	w_i	$y(t_i)$	error
0	1	2	2	0
1	1.25	2.778909	2.778929	1.9×10^{-5}
2	1.5	3.608164	3.608198	3.3×10^{-5}
3	1.75	4.479284	4.479328	4.3×10^{-5}
4	2	5.386243	5.386294	5.2×10^{-5}

As you can see from the three tables that compare the method values to the exact values the higher the order of the method the smaller the error. This phenomenon can be explained by the how the methods are derived.

2.2.1 Error Analysis in the Runge-Kutta Method

As in Euler's method, Runge-Kutta methods are based off of Taylor series polynomial. Second-order Runge-Kutta methods like the Midpoint method use the second-order Taylor series polynomial so the remainder term would be

$$R_n(t_i) = \frac{y^{(2)}(\xi(t))}{2!} i^2 h^2. \tag{2.12}$$

Thus the Midpoint method has error of $\mathcal{O}(h^2)$ so long as the derivative $y^{(2)}$ is bounded by M on $[a, b]$. Because $h^2 < h$ for step sizes less than 1, then the error of the Midpoint method is less than the error of Euler's method.

The fourth-order Runge-Kutta uses the fourth-order Taylor polynomial so the method's remainder term is

$$R_n(t_i) = \frac{y^{(4)}(\xi(t))}{4!} i^4 h^4. \tag{2.13}$$

The error is $\mathcal{O}(h^4)$, clearly better than all other methods presented so far so long as the derivative $y^{(4)}$ is bounded by M on $[a, b]$. The fourth-order Runge-Kutta method has a better error than using Euler's method or the Midpoint method with a much larger step size. To illustrate this we will compare Problem 2.3 solved with Euler's method with a $h = 0.025$, the Midpoint method with a $h = 0.05$, and fourth-order Runge-Kutta with a $h = 0.1$. Notice how even though the fourth-order Runge-Kutta method has the largest step size it has the smallest error. This is because even though the step size is larger $.1^5 < .025^2$. So since the error of Euler's method is $\mathcal{O}(h^2)$ and the fourth-order Runge-Kutta is $\mathcal{O}(h^5)$ then the Runge-Kutta error should be less as shown by the table. It should also be said that fourth-order Runge-Kutta will have less computations than the other methods to attain similar error. Since the step size can be larger the computer will be able to go through the method few times to get accurate estimations.

Table 2.5: Table to compare values of the Initial Value Problem 2.3 solved with Euler's method with $h = 0.025$, the Midpoint method with $h = 0.05$, and fourth-order Runge-Kutta with $h = 0.1$.

Time	fourth-order Runge Kutta	Midpoint	Eulers	Exact
1	2	2	2	2
1.1	2.304840926	2.304821327	2.30360114	2.30484120
1.2	2.618785383	2.618747702	2.616304963	2.61878587
1.3	2.941072883	2.941018294	2.937351185	2.94107354
1.4	3.27106032	3.27098975	3.266096842	3.27106113
1.5	3.608196718	3.608110911	3.601991061	3.60819766
1.6	3.952004743	3.951904306	3.9445565847	3.95200581
1.7	4.302066851	4.301952286	4.293375926	4.30206803
1.8	4.658014717	4.657886441	4.648080806	4.65801600
1.9	5.019521004	5.019379371	5.008343924	5.01952238
2	5.386292886	5.386138196	5.37387248	5.38629436

Chapter 3

Numerical Method for Systems and Higher-Order Methods

3.1 Runge-Kutta on First-Order Systems

Runge-Kutta can also be extended for systems of initial value problems. A system of initial value problems with m equations has the form

$$\begin{aligned}
\frac{du_1}{dt} &= f_1(t, u_1, u_2, \ldots, u_m), \\
\frac{du_2}{dt} &= f_2(t, u_1, u_2, \ldots, u_m), \\
&\vdots \\
\frac{du_m}{dt} &= f_m(t, u_1, u_2, \ldots, u_m), \qquad (3.1)
\end{aligned}$$

for $t \in [a, b]$, with the initial conditions

$$u_1(a) = \alpha_1, u_2(a) = \alpha_2, \ldots, u_m(a) = \alpha_m.$$

The way the Runge-Kutta for system works is you choose an N for the number of mesh points you want to solve. Now just like Euler's and Runge-Kutta before, the step size is defined by Equation 2.2 and $t_j = a + jh$ for each $j = 0, 1, \ldots, N$. We will denote w_{ij} to be the approximate solution of u_i in the system of Equations 3.1 at the jth mesh point

t_j (Burden & Faires, 2003). So the initial conditions give us

$$w_{1,0} = \alpha_1, w_{2,0} = \alpha_2, \ldots, w_{m,0} = \alpha_m.$$

Now like with the fourth-order Runge-Kutta method in order to find w_{j+1} using known w_j values we need to compute k_1, k_2, k_3, k_4 values. For systems we need to calculate each k value for each equation in the system before moving on to the next k value. So for each $i = 1, 2, \ldots, m$ calculate

$$k_{1,i} = hf_i(t_j, w_{1,j}, w_{2,j}, \ldots, w_{m,j}),$$

then for each $i = 1, 2, \ldots, m$ calculate

$$k_{2,i} = hf_i(t_j + \frac{h}{2}, w_{1,j} + \frac{1}{2}k_{1,1}, w_{2,j} + \frac{1}{2}k_{1,2}, \ldots, w_{m,j} + \frac{1}{2}k_{1,m}),$$

then for each $i = 1, 2, \ldots, m$ calculate

$$k_{3,i} = hf_i(t_j + \frac{h}{2}, w_{1,j} + \frac{1}{2}k_{2,1}, w_{2,j} + \frac{1}{2}k_{2,2}, \ldots, w_{m,j} + \frac{1}{2}k_{2,m}),$$

then for each $i = 1, 2, \ldots, m$ calculate

$$k_{4,i} = hf_i(t_j + \frac{h}{2}, w_{1,j} + k_{3,1}, w_{2,j} + k_{3,2}, \ldots, w_{m,j} + k_{3,m}),$$

and then for each $i = 1, 2, \ldots, m$

$$w_{i,j+1} = w_{i,j} + \frac{1}{6}(k_{1,i} + 2k_{2,i} + 2k_{3,i} + k_{4,i}).$$

Again, like any of the other methods it can help to see a simple example. However,

to show the calculations for one step in a system of three equations you would need to calculate twelve k values and three difference equations. The best way to show this method is to implement it on a computer and analyze the results. We implement the systems code in MATLAB for the specific problem

$$u_1' = u_2 - u_3 + t, u_1(0) = 1;$$

$$u_2' = 3t^2, u_2(0) = 1;$$

$$u_3' = u_2 + e^{-t}, u_3(0) = -1 \tag{3.2}$$

for $t \in [0, 1]$. We used a step size of $h = 0.1$ and were given the exact solutions of $u_1(t) = -0.05t^5 + 0.25t^4 + t + 2 - e^{-t}$,
$u_2(t) = t^3 + 1$, and $u_3(t) = 0.25t^4 + t - e^{-t}$. The fourth -rder Runge-Kutta for system of equations produced the values for w_{ij} as seen in Table 3.1. To see the MATLAB implementation see Appendix A.1.3.

Table 3.1: Table for the Initial Value Problem 3.1 with values found by the fourth-order Runge-Kutta for systems and values found using the exact solution.

j	t_i	w_{1j}	u_{1j}	w_{2j}	u_{2j}	w_{3j}	u_{3j}
0	0	1	1	1	1	-1	-1
4	.3	1.561084867	1.561085279	1.027	1.027	-0.438793212	-0.438793221
7	.6	2.079699546	2.079700364	1.216	1.216	0.08358838	0.083588364
9	.8	2.436685951	2.436687036	1.512	1.512	0.453071055	0.453071036
11	1	2.832119208	2.832120559	2	2	0.882120581	0.882120559

3.2 Runge-Kutta on Higher-Order Systems

All of the methods described thus far are to solve ordinary differential equations. Unfortunately, as we will see in Chapter 4, the Optimal Velocity Model that we are working towards is a system of second-order differential equations. We have to do is convert each

second-order differential equation to two first-order differential equations.

Say we have the second-order differential equation:

$$\frac{d^2y}{dt^2} = f\left(t, y, \frac{dy}{dt}\right), \quad y(a) = \alpha_1, \quad \frac{dy}{dt}(a) = \alpha_2. \tag{3.3}$$

Then we make the system of equations as follows:

$$\frac{dy}{dt} = v,$$

$$\frac{dv}{dt} = f\left(t, y, v\right).$$

Now this is just a simple system of two coupled first-order differential equations. Now we can use the fourth-order Runge-Kutta method for systems.

Chapter 4

Numerical Solution of a Traffic Model

In this chapter, we consider the Optimal Velocity Model or OVM for traffic flow (Kurata & Nagatani, 2003). The initial-value problem cannot be solved analytically. We will approximate the solution using the Runge-Kutta method. Kurata and Nagatani (2003) name this approach to the problem. We implement the method and discuss how and why the Runge-Kutta method described in Section 3.1 can be extended to apply to the model of traffic flow.

4.1 The Model

We describe the movement of car i using the following equation (Kurata & Nagatani, 2003):

$$\frac{d^2x_i(t)}{dt^2} = a\left(V(\Delta x_i(t)) - \frac{dx_i(t)}{dt}\right). \tag{4.1}$$

The position of car i at time t is $x_i(t)$. We define $\Delta x_i(t)$ to be the distance between car i and the car in front of it at time t. So,

$$\Delta x_i(t) = x_{i+1}(t) - x_i(t). \tag{4.2}$$

This is often referred to as the the headway of car i at time t. The constant a is the sensitivity parameter. It describe how sensitive the average driver is to the motion of

the car in front of him. The sensitivity parameter is actually the inverse of the time lag between a front car changing speeds and the car behind it reacting and then adjusting to the front car. The velocity of the car i is given by $\frac{dx_i(t)}{dt}$. The function $V(\Delta x_i(t))$ is the optimal velocity of the car. The optimal velocity is given by

$$V(\Delta x_i(t)) = \frac{v_{max}}{2}[\tanh(\Delta x_i(t) - x_c) + \tanh(x_c)]. \tag{4.3}$$

If $x_c = 4$, then as the headway $\Delta x_i(t)$ approaches infinity, the optimal velocity $V(\Delta x_i(t))$ approaches a value within .0004% of v_{max}. Thus, cars will go approximately the maximal velocity when the headway is sufficiently larger than the safety distance. If the safety distance becomes infinitely large the value of $V(\Delta x_i(t))$ will approach zero.

The Optimal Velocity Model 4.1 has the acceleration of car i, $\frac{d^2x_i(t)}{dt^2}$, on the left-hand side. When the optimal velocity in Equation 4.3 is greater than the current velocity $\frac{dx_i(t)}{dt}$ of the car, the acceleration is positive. The velocity of the car will increase in this case. When the optimal velocity is equal to the current velocity then the acceleration is zero. Thus the car's speed stays constant. When the current velocity is greater than the optimal velocity the acceleration is negative and the car will slow down.

The acceleration has a proportionality parameter a. Since the sensitivity parameter is the inverse of the time lag it takes for a driver to react to a car in front of it, a high sensitivity parameter is a quick response time. A high sensitivity parameter in turn leads to a higher acceleration.

When we implemented the fourth-order Runge-Kutta method on the model we let the parameters be the following:

$$a = 1,$$

$$v_{max} = 4,$$

$$x_c = 4.$$

We ran each Runge-Kutta method with a step size of .001 over the t interval $[0, 6]$.

4.2 Implementation of Runge-Kutta on the Model

The Optimal Velocity Model 4.1 is a system of second-order differential equations. The system of Equations 3.1 in Section 3.1 is a system of first-order equations. We need to change each second-order differential equation into two first-order differential equations as described in Subsection 3.2, where

$$\frac{dx_i(t)}{dt} = y_i(t) \text{ and } \frac{dy_i(t)}{dt} = a(V(x_{i+1}(t) - x_i(t)) - y_i(t)). \tag{4.4}$$

Each car i has two differential equations associated with it. Thus when there are M cars there are $2M$ equations in our system.

Now in order to implement a fourth-order Runge-Kutta method we need to start with some initial condition for each equation. In this model, we require a starting position and velocity of each car in the system. Kurata and Nagatani (2003) do not explicitly define initial conditions. Without initial conditions there is no way to replicate their exact simulation. We define our own initial conditions.

The other issue is that the Optimal Velocity Equation 4.3 involves a car in front of the ith car. The last equation in the system is

$$\frac{dy_M(t)}{dt} = a(V(x_{M+1}(t) - x_M(t)) - y_i(t)).$$

This relies on an $(M+1)$st car which does not exist. Thus how do we deal with motion of the Mth car which is a part of the system? This issue is not addressed by Kurata

and Nagatani (2003). The whole system relies on how to prescribe the behavior of the front car, which in turn affects the behavior of the following cars. We prescribe our own x_{M+1} values.

We decided to simplify the model down to just one car and then two cars to validate the application of the method and the model.

4.3 Simulation with a Stopped Object and One Car

This simulation illustrates the response of a car to a stopped object. We place the object close enough that the car has to slow down.

For this simulation we have the first-order system in Equation 4.4 for the position $x_1(t)$ and velocity $y_1(t)$ of a single car. In Section 4.1, we discuss $x_2(t)$ as the position of the second car in the system, but here we think of it as an object stopped at position 20. Throughout the simulation we use

$$x_2(t) = 20.$$

Thus, the system is

$$\frac{dx_1(t)}{dt} = y_1(t), \quad \frac{dy_1(t)}{dt} = a(V(20 - x_1(t)) - y_1(t)).$$

The car begins at position

$$x_1(0) = 0.$$

The velocity of the car is initially set to be the optimal velocity as follows:

$$\left.\frac{dx_1}{dt}\right|_{t=0} = V(\Delta x_1(0)),$$

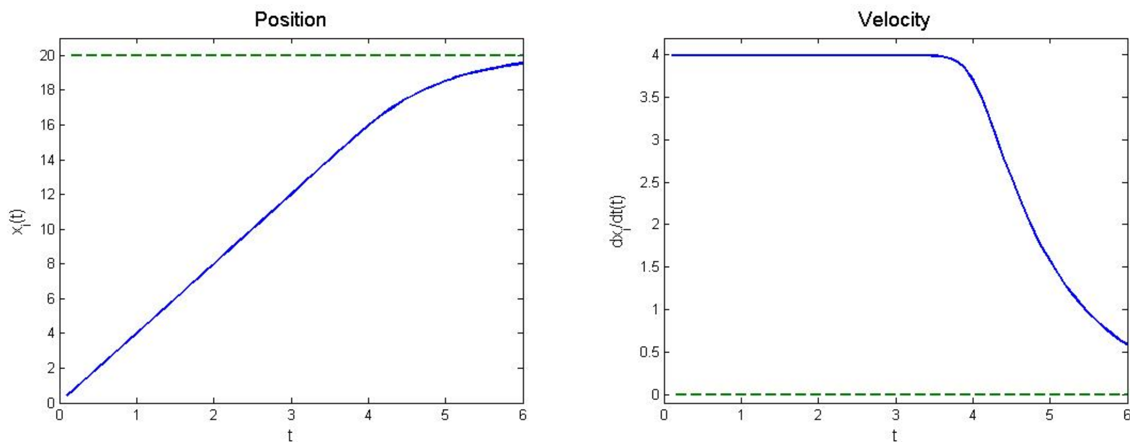

Fig. 4.1: Position (left graph) and velocity (right graph) versus time for a car (solid line) approaching a stopped object (dashed line).

where

$$\Delta x_1(0) = x_2(0) - x_1(0) = 20 - 0 = 20$$

is the headway at time zero.

Now we can use the fourth-order Runge-Kutta method to solve for the position $x_1(t)$ and the velocity $y_1(t)$ of the car. The code appears in Appendix A.2.2.

Figure 4.1 shows the results of the simulation. The left figure shows the position of the car and the stopped object as functions of time. The right figure shows the velocities. Both graphs in Figure 4.1 show the car slowing down as it approaches the object.

4.4 Simulation with a Clear Road in Front of the Car

This simulation illustrates the response of a car to a clear road ahead of it. We place the object so far away that it has no affect on the car.

For this simulation we have the first-order system in Equation 4.4 for the position $x_1(t)$

and velocity $y_1(t)$ of a single car. Just as in the stopped car simulation we think of $x_2(t)$ as an object stopped at the far away position of 100. Throughout the simulation we use $x_2(t) = 100$. Thus, the system is

$$\frac{dx_1(t)}{dt} = y_1(t), \quad \frac{dy_1(t)}{dt} = a(V(100 - x_1(t)) - y_1(t)).$$

The car begins at position

$$x_1(0) = 0.$$

The velocity of the car is initially set to be the slow velocity of 1, recall in Section 4.1 the v_max is four. Thus,

$$\left.\frac{dx_1}{dt}\right|_{t=0} = 1.$$

Starting the car so slow will allow it to speed up throughout the simulation.

Now we can use the fourth-order Runge-Kutta method to solve for the position $x_1(t)$ and the velocity $y_1(t)$ of the car. The code appears in Appendix A.2.3.

Figure 4.2 shows the results of the simulation. The left figure shows the position of the car as a function of time. The right figure shows the velocity. The right graph in Figure 4.2 show the car speeding up to reach the maximum velocity of four. The concavity of the right graph shows that the car is accelerating because it is concave up.

4.5 Simulation with an Object Traveling with a Slow Constant Speed

This simulation illustrates the response of a car to a slow moving vehicle in front of it. We start the cars close enough together that the back car will need to adjust its speed.

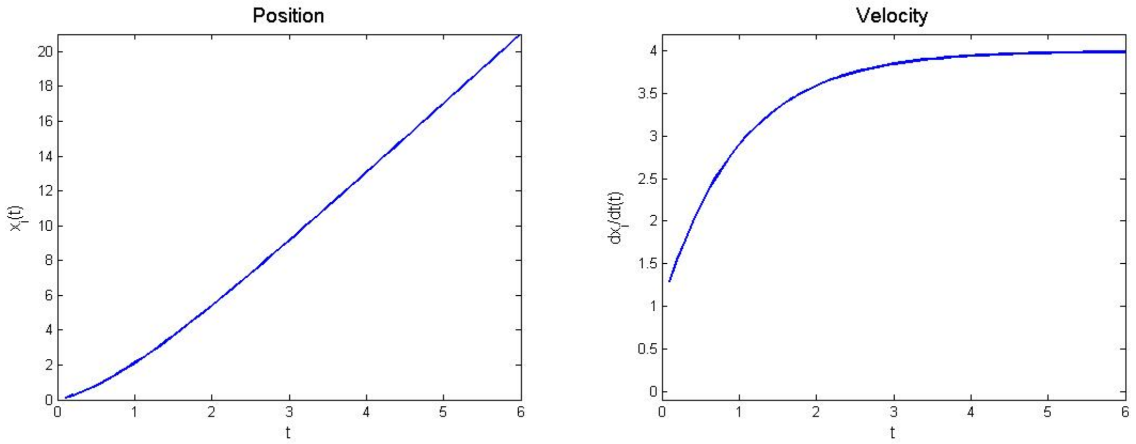

Fig. 4.2: Position (left graph) and velocity (right graph) versus time for a car that starts out slow and accelerates because it has a clear path.

For this simulation we have the first-order system in Equation 4.4 for the position $x_1(t)$ and velocity $y_1(t)$ of a single car. In this simulation, $x_2(t)$ does describe a car's movement. We give it a prescribed velocity of two throughout the simulation and a starting position of ten. Thus,

$$x_2(t) = 10 + 2t.$$

Thus, the system is

$$\frac{dx_1(t)}{dt} = y_1(t), \quad \frac{dy_1(t)}{dt} = a(V((10 + 2t) - x_1(t)) - y_1(t)).$$

Car 1 begins at position

$$x_1(0) = 5.$$

The velocity of the car is initially set to be the optimal velocity as follows:

$$\left.\frac{dx_1}{dt}\right|_{t=0} = V(\Delta x_1(0)),$$

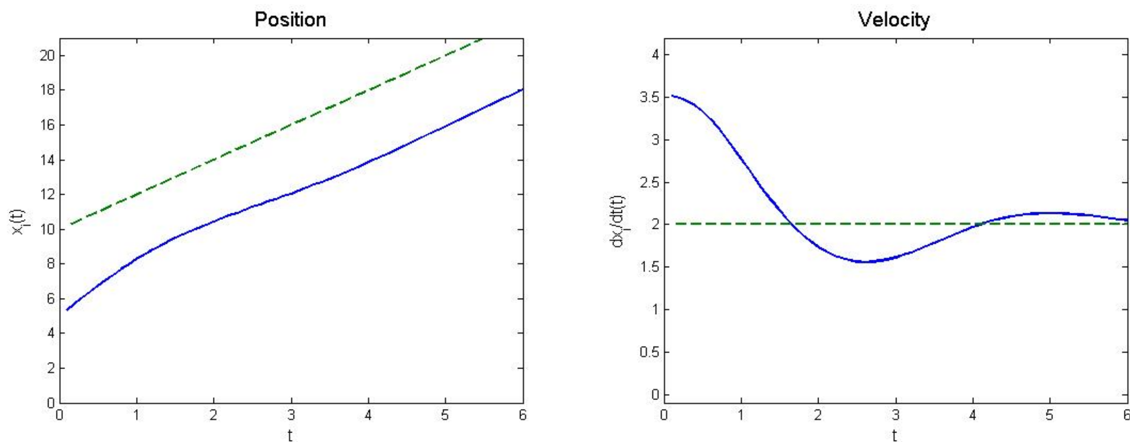

Fig. 4.3: Position (left graph) and velocity (right graph) versus time for a car (solid line) with a slow moving constant speed object (dashed line) in front of it.

where

$$\Delta x_1(0) = x_2(0) - x_1(0) = 10 - 5 = 5$$

is the headway at time zero.

Now we can use the fourth-order Runge-Kutta method to solve for the position $x_1(t)$ and the velocity $y_1(t)$ of the car. The code appears in Appendix A.2.4.

Figure 4.3 shows the results of the simulation. The left figure shows the position of the car and the constant moving car as functions of time. The right figure shows the velocities. Both graphs in Figure 4.3 show the car slowing down to match the speed of the slower moving car.

4.6 Simulation with a Stopped Object and Two Cars

This simulation illustrates the response of two cars to a stopped object. This is similar to the simulation in Section 4.3, but we add another car to see how the dynamics change.

For this simulation we have the first-order system in Equation 4.4 for the positions $x_i(t)$

and velocities $y_i(t)$ of when i is one and then two. Analogous to $x_2(t)$ in Section 4.3, here $x_3(t)$ is the position of a stopped object at 20. Throughout the simulation we use

$$x_3(t) = 20.$$

Thus, the system is

$$\frac{dx_i(t)}{dt} = y_i(t), \quad \frac{dy_i(t)}{dt} = a(V(x_{i+1}(t) - x_i(t)) - y_i(t)), \quad \text{where } i = 1, 2.$$

The headways of the cars are

$$\Delta x_1(t) = x_2(t) - x_1(t) \text{ and } \Delta x_2(t) = 20 - x_2(t).$$

The cars begin at positions

$$x_1(0) = 0 \text{ and } x_2(0) = 5.$$

The velocity of the cars is initially set to be the optimal velocity as follows:

$$\left.\frac{dx_i}{dt}\right|_{t=0} = V(\Delta x_i(0)), \quad \text{where } i = 1, 2.$$

Now we can use the fourth-order Runge-Kutta method to solve for the positions $x_i(t)$ and the velocities $y_i(t)$ where $i = 1, 2$. The code appears in Appendix A.2.5.

Figure 4.4 shows the results of the simulation. The left figure shows the position of the cars. The right figure shows the velocities. Both graphs in Figure 4.4 show car 2 (dashed line) slowing down as it approaches the stopped object. Car 1 (solid line) starts slightly slower than the car 2, approaches maximum velocity, and then slows down.

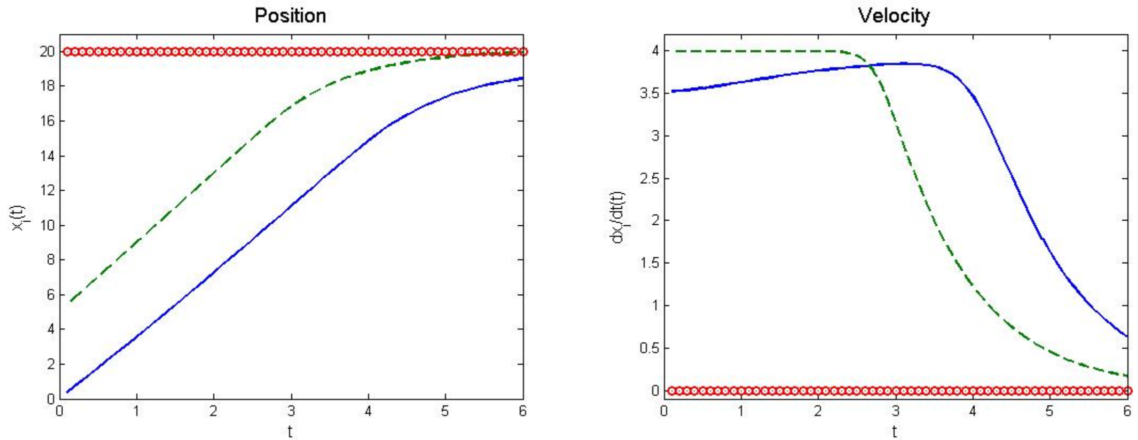

Fig. 4.4: Position (left graph) and velocity (right graph) versus time for two cars (solid line and dashed line) approaching a stopped object.

4.7 Simulation with a Clear Road in Front of Two Cars

This simulation illustrates the response of two cars to an open road. This is similar to the simulation in Section 4.4, but we add another car to see how the dynamics change.

For this simulation we have the first-order system in Equation 4.4 for the positions $x_i(t)$ and velocities $y_i(t)$ of when i is one and then two. In analogy with Section 4.4 we consider $x_3(t)$ to be the position of a far away object at 100. Throughout the simulation we use

$$x_3(t) = 100.$$

Thus, the system is

$$\frac{dx_i(t)}{dt} = y_i(t), \quad \frac{dy_i(t)}{dt} = a(V(x_{i+1}(t) - x_i(t)) - y_i(t)), \quad \text{where } i = 1, 2.$$

The headways of the cars are

$$\Delta x_1(t) = x_2(t) - x_1(t) \text{ and } \Delta x_2(t) = 100 - x_2(t).$$

The cars begin at positions

$$x_1(0) = 0 \text{ and } x_2(0) = 5.$$

The velocity of the car 1 is initially set to be the optimal velocity as follows:

$$\left.\frac{dx_1}{dt}\right|_{t=0} = V(\Delta x_1(0)),$$

where

$$\Delta x_1(0) = 5.$$

The velocity of car 2 is given the slow initial velocity of 1. Thus,

$$\left.\frac{dx_2}{dt}\right|_{t=0} = 1.$$

Now we can use the fourth-order Runge-Kutta method to solve for the positions $x_i(t)$ and the velocities $y_i(t)$ where $i = 1, 2$. The code appears in Appendix A.2.6.

Figure 4.5 shows the results of the simulation. The left figure shows the position of the cars as functions of time. The right figure shows the velocities. Both graphs in Figure 4.5 show car 2 (dashed line) speeding up toward the maximum velocity and car 1 (solid line) following suit. Car 1 needs to slow down at first because the two cars are so close, but then it speeds up as car 2 gains speed.

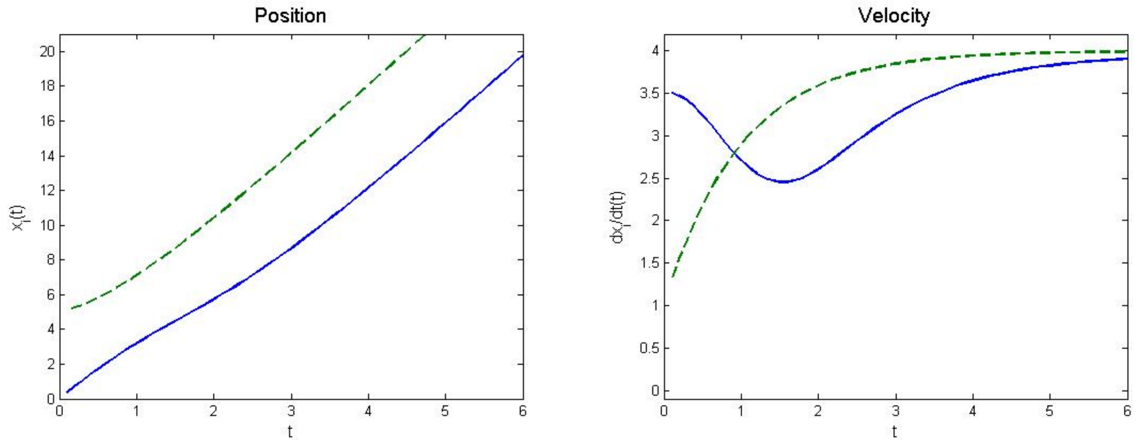

Fig. 4.5: Position (left graph) and velocity (right graph) versus time for two cars (solid line and dashed line) accelerating.

4.8 Simulation with an Object Traveling with a Slow Constant Speed and Two Cars

This simulation illustrates the response of two cars to a slow constant moving object. This is similar to the simulation in Section 4.5, but we add another car to see how the dynamics change.

For this simulation we have the first-order system in Equation 4.4 for the positions $x_i(t)$ and velocities $y_i(t)$ of when i is one and then two. As in Section 4.5 we consider $x_3(t)$ to be the position of a car moving with a constant velocity of three and starting at position ten. Thus,

$$x_3(t) = 10 + 3t$$

throughout the simulation. Thus, the system is

$$\frac{dx_i(t)}{dt} = y_i(t), \quad \frac{dy_i(t)}{dt} = a(V(x_{i+1}(t) - x_1(t)) - y_i(t)), \quad \text{where } i = 1, 2,$$

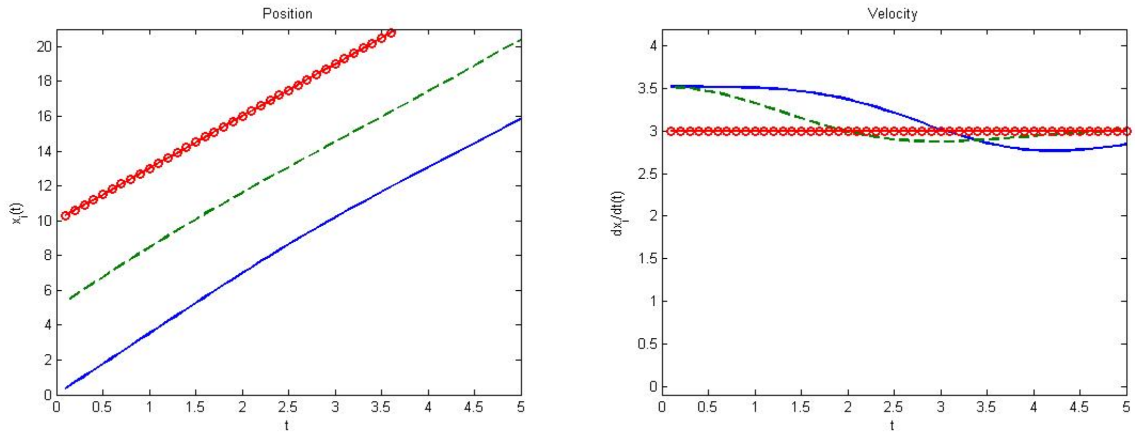

Fig. 4.6: Position (left graph) and velocity (right graph) versus time for two car (solid line and dashed line) slowing down for a slow moving object (circle line).

where

$$\Delta x_1(t) = x_2(t) - x_1(t) \text{ and } \Delta x_2(t) = (10 + 3t) - x_1(t).$$

The cars begin at positions

$$x_1(0) = 0, x_2(0) = 5, \text{ and } x_3(0) = 10.$$

The velocity of the cars is initially set to be the optimal velocity as follows:

$$\left.\frac{dx_i}{dt}\right|_{t=0} = V(\Delta x_i(0)).$$

Now we can use the fourth-order Runge-Kutta method to solve for the positions $x_i(t)$ and the velocities $y_i(t)$ where $i = 1, 2$. The code appears in Appendix A.2.7.

Figure 4.6 shows the results of the simulation. The left figure shows the position of the cars as functions of time. The right figure shows the velocities. Both graphs in Figure 4.6 show car 2 (dashed line) slowing down to adjust to the slower speed of the object ahead (circle line) and car 1 (solid line) following suit.

Chapter 5

Conclusions

In this work we shed light on how to implement the fourth-order Runge-Kutta method on the Optimal Velocity Model (Kurata & Nagatani, 2003). We defined and discussed initial conditions and base cases of the lead car. We also expanded the dynamics explored by Kurata and Nagatani (2003) by imposing a variety of behaviors for the leading car as well as a variety of initial conditions. By considering a small number of cars in one lane we have validated the Optimal Velocity Model by accurately simulating traffic scenarios. Since the traffic simulations replicated real life well, the performance of the fourth-order Runge-Kutta method produced good results for the Optimal Velocity Model.

Chapter 6

Future Work

In future work, we would like to replicate the updating system of Kurata and Nagatani (2003) based on distance in between the last car and the start of the system. Kurata and Nagatani (2003) added a car when the last car reached a certain distance. We would like to explore a time updating system. So instead of adding a car once the last car reached a certain distance, add a car each time the system advanced a specified amount of time.

Another interesting simulation would be to set the lead car to have a velocity that slows downs and speeds up. An equation like $v(t) = 2 + \alpha \sin \omega t$, where α and ω are constants would achieve this scenario.

Another area of interest would be analyzing the lane changing conditions presented by Kurata and Nagatani (2003) and how they interact with the implementation of the Runge-Kutta. In the literature, it is not explicitly described how to deal with such a system.

Chen et al. (2014) uses Euler's method on the Full Velocity Difference model. An interesting project would be to repeat the process done here to explicitly state the initial conditions and base cases for the lead car.

References

Burden, R. L., & Faires, J. D. (2003). *Numerical analysis.* Boston, MA: Brooks/Cole.

Chen, J., Peng, Z., & Fang, Y. (2014). Effects of car accidents on three-lane traffic flow. *Mathematical Problems in Engineering*, *2014*(413852).

Kurata, S., & Nagatani, T. (2003). Spatio-temporal dynamics of jams in two-lane traffic flow with a blockage. *Physica A: Statistical Mechanics and Its Applications*, *318*, 537–550.

MATLAB. (2014). *Version 8.3 (r2014a).* Natick, Massachusetts: The MathWorks Inc.

Appendix A
Code Implementation

This appendix contains the MATLAB (2014) code writen for the thesis.

A.1 Textbook Problem Codes

A.1.1 Euler's Method

Equations

The function y1 is the right-hand side of the differential equation 2.1.

```
function [ c ] = y1( t,y )
c=1+y/t;

end
```

The function y1exact is the exact solution to the initial value problem 2.3.

```
function [ c ] = y1exact( t )
c=t*(log(abs(t))+2);

end
```

Method

The function Eulers performs Euler's Method with N mesh points on the initial value problem 2.1. The input a is the starting t value, b is the ending t value, N is the number of mesh points, alpha is the initial condition, f is the right-hand side of the differential equation 2.1, and the function fe is the exact solution.

```
function [ c ] = Eulers(a,b,N,alpha,f,fe  )
h=(b-a)/N;
t=a;
w=alpha;
c=[t w fe(t)];
for i=1:N
    w=w+h*f(t,w);
    t=a+i*h;
    c(1+i,:)=[t w fe(t)];
end
 x=1:.01:2;
 for i=1:101 y(i)=y1exact(x(i)); end
 figure
 plot(x,y,c(:,1),c(:,2),'o-')
 title('Euler''s Method')
 ylabel('y')
 xlabel('x')

end
```

A.1.2 Runge-Kutta Method

Equations

The function y1 is the right-hand side of the differential equation 2.1.

```
function [ c ] = y1( t,y )
c=1+y/t;

end
```

The function y1exact is the exact solution to the initial value problem 2.3..

```
function [ c ] = y1exact( t )
c=t*(log(abs(t))+2);

end
```

Method

The function Runge4th performs Fourth Order Runge-Kutta Method with N mesh points on an initial value problem 2.1. The input a is the starting t value, b is the ending t value, N is the number of mesh points, alpha is the initial condition, f is the right-hand side of the differential equation 2.1, and the function fe is the exact solution.

```
function [ c] = Runge4th(a,b,N,alpha,f,fe  )
h=(b-a)/N;
t=a;
w=alpha;
```

```
c=[t w];
for i=1:N
    k1=h*f(t,w);
    k2=h*f(t+h/2,w+k1/2);
    k3=h*f(t+h/2,w+k2/2);
    k4=h*f(t+h,w+k3);
    w=w+(k1+2*k2+2*k3+k4)/6;
    t=a+i*h;
    c(1+i,:)=[t w];
end

x=1:.01:2;
for i=1:101 y(i)=fe(x(i)); end
figure
plot(x,y,c(:,1),c(:,2),'o-');
 title('Fourth Order Runge-Kutta Method')
 ylabel('y')
 xlabel('x')

end
```

A.1.3 Runge-Kutta for Systems

Equations

The functions f1, f2, and f3 are the right-hand sides of the equations 3.1.

```
function z=f1(t,u1,u2,u3)
z=u2-u3+t;
end
```

```
function z= f2(t, u1, u2, u3)
z=3*t^2;
end

function z=f3(t, u1, u2, u3)
z=u2+exp(-t);
end
```

Method

The function FourthOrderSystem will perform the Fourth Order Runge-Kutta Method on the above system of equations. The input a is the starting t value, b is the ending t value, m is the number of equations, N is the number of mesh points, the vector alpha is the initial condition, f is the a vector whose components are the functions f1, f2, f3 above.

```
function [c] = FourthOrderSystem( a,b,m,N,alpha,f)

h=(b-a)/N;
t= a:h:b;
solutions=zeros((N+1),(6+1));
w=alpha;
k=zeros(4,m);
solutions(:,1)=t;

for j= 1:m
    solutions(1,(j+1))= alpha(j);
end
for i=1:N
```

```
        for j=1:m
            k(1,j)=h*f{j}(t(i),w(1),w(2),w(3));
        end
        for j=1:m
            k(2,j)=h*f{j}(t(i)+h/2,w(1)+1/2*k(1,1),w(2)+1/2*k(1,2),w(3)+1/2*k(1,3));
        end
        for j=1:m
            k(3,j)=h*f{j}(t(i)+h/2,w(1)+1/2*k(2,1),w(2)+1/2*k(2,2),w(3)+1/2*k(2,3));
        end
        for j=1:m
            k(4,j)=h*f{j}(t(i)+h,w(1)+k(3,1),w(2)+k(3,2),w(3)+k(3,3));
        end
        for j=1:m
             w(j)=w(j)+(k(1,j)+2*k(2,j)+2*k(3,j)+k(4,j))/6;
             solutions(i+1,j+1)=w(j);
             solutions(i+1,5:7)=[exactf1(t(i+1)) exactf2(t(i+1)) exactf3(t(i+1))];
        end
    end
    c=solutions;
    end
```

A.2 Traffic Model Code

A.2.1 Traffic Model Equation Code

These three MATLAB functions make up the Optimal Velocity Model of traffic flow. They can be called by a numerical method such as the Fourth Order Runge-Kutta method to run simulations.

The function vdot is the factor multiplying the sensitivity parameter a in the Optimal

Velocity Model, equation 4.1.

```
function x = vdot( u1,u2, y)
x=V(u1, u2)-y;
end
```

The function V is the Optimal Velocity Equation, equation 4.3.

```
function c= V(u1,u2)
c=4/2*(tanh((u2-u1)-4)+tanh(4));
end
```

The function xdot returns the value of the velocity of the car.

```
function u = xdot( y )
u=y;

end
```

A.2.2 Runge-Kutta for Stopped Object and One Car

The function RungeKuttaStopped performs a Fourth Order Runge-Kutta on the Optimal Velocity Model. This program creates a graph of a car starting at zero approaching an object stopped at 20.

```
function c = RungeKuttaStopped(  )

lane1=[0 20];
```

```
%lane1 is the initial positions of cars
v1=V(lane1(1),lane1(2));
%v are the initial velocity of the cars
h=.001;
disp(v1);
count=1;
fileID=fopen('RungeKuttaStopped.dat','w');
for t=1:6000

    kx2(1)=h*xdot(v1);
    kv2(1)=h*vdot(lane1(1),lane1(2), v1);



    kx2(2)=h*xdot(v1+1/2*kv2(1));
    kv2(2)=h*vdot(lane1(1)+1/2*kx2(1),lane1(2),v1+1/2*kv2(1));

    kx2(3)=h*xdot(v1+1/2*kv2(2));
    kv2(3)=h*vdot(lane1(1)+1/2*kx2(2),lane1(2),v1+1/2*kv2(2));

    kx2(4)=h*xdot(v1+kv2(3));
    kv2(4)=h*vdot(lane1(1)+kx2(3),lane1(2),v1+kv2(3));



    lane1(1)=lane1(1)+(kx2(1)+2*kx2(2)+2*kx2(3)+kx2(4))/6;


    v1=v1+(kv2(1)+2*kv2(2)+2*kv2(3)+kv2(4))/6;


    if (mod(t,100)==0)
        rkplot(count,:)=[t*h lane1(1) lane1(2) v1 0];
        count=1+count;
    end
```

```
    if (mod(t,100)==0)
       fprintf(fileID, '%4.7f  %4.7f   \r\n', lane1);

    end



end
fclose(fileID);
figure
plot(rkplot(:,1),rkplot(:,2),'-',...
    rkplot(:,1),rkplot(:,3),'--','LineWidth',2);
title('Position','FontSize',14)
xlabel('t','FontSize',12)
ylabel('x_i(t)','FontSize',12)
axis([0 6 0 21])

figure
plot(rkplot(:,1),rkplot(:,4),'-',...
    rkplot(:,1),rkplot(:,5),'--','LineWidth',2);
title('Velocity','FontSize',14)
xlabel('t','FontSize',12)
ylabel('dx_i/dt(t)','FontSize',12)
axis([0 6 -.1 4.2])

c=[lane1(1) lane1(2)];

end
```

A.2.3 Runge-Kutta for a Clear Road in Front of the Car

The function RungeKuttaClearPath performs a Fourth Order Runge-Kutta on the Optimal Velocity Model. This program creates a graph of a car starting at zero approaching a far away object at 100.

```
function c = RungeKuttaClearPath(  )

lane1=[0 100];
%lane1 is the initial positions of cars
v1=1;
%v are the initial velocity of the cars
h=.001;
disp(v1);
count=1;
fileID=fopen('RungeKuttaStopped.dat','w');
for t=1:6000

    kx2(1)=h*xdot(v1);
    kv2(1)=h*vdot(lane1(1),lane1(2), v1);



    kx2(2)=h*xdot(v1+1/2*kv2(1));
    kv2(2)=h*vdot(lane1(1)+1/2*kx2(1),lane1(2),v1+1/2*kv2(1));


    kx2(3)=h*xdot(v1+1/2*kv2(2));
    kv2(3)=h*vdot(lane1(1)+1/2*kx2(2),lane1(2),v1+1/2*kv2(2));


    kx2(4)=h*xdot(v1+kv2(3));
    kv2(4)=h*vdot(lane1(1)+kx2(3),lane1(2),v1+kv2(3));
```

```
    lane1(1)=lane1(1)+(kx2(1)+2*kx2(2)+2*kx2(3)+kx2(4))/6;

    v1=v1+(kv2(1)+2*kv2(2)+2*kv2(3)+kv2(4))/6;

    if (mod(t,100)==0)
        rkplot(count,:)=[t*h lane1(1) lane1(2) v1 0];
        count=1+count;
    end

    if (mod(t,100)==0)
       fprintf(fileID, '%4.7f  %4.7f   \r\n', lane1);

    end


end
fclose(fileID);
figure
plot(rkplot(:,1),rkplot(:,2),'-',...
    'LineWidth',2);
title('Position','FontSize',14)
xlabel('t','FontSize',12)
ylabel('x_i(t)','FontSize',12)
axis([0 6 0 21])

figure
plot(rkplot(:,1),rkplot(:,4),'-',...
    'LineWidth',2);
title('Velocity','FontSize',14)
xlabel('t','FontSize',12)
ylabel('dx_i/dt(t)','FontSize',12)
```

```
axis([0 6 -.1 4.2])

c=[lane1(1) lane1(2)];

end
```

A.2.4 Runge-Kutta for a Car Following an Object Traveling with a Slow Constant Speed

The function RungeKuttaConstant performs a Fourth Order Runge-Kutta on the Optimal Velocity Model. This program creates a graph of a car starting at five approaching an object starting at ten moving at a speed of two.

```
function c = RungeKuttaConstant( )

lane1=[5 10];
%lane1 is the initial positions of cars
v1=V(lane1(1),lane1(2));
v2=2;
%v are the initial velocity of the cars
h=.001;
disp(v1);
count=1;
fileID=fopen('RungeKuttaConstant.dat','w');
for t=1:6000

    kx1(1)=h*xdot(v1);
    kv1(1)=h*vdot(lane1(1),lane1(2), v1);
```

```
    kx1(2)=h*xdot(v1+1/2*kv1(1));
    kv1(2)=h*vdot(lane1(1)+1/2*kx1(1),lane1(2),v1+1/2*kv1(1));

    kx1(3)=h*xdot(v1+1/2*kv1(2));
    kv1(3)=h*vdot(lane1(1)+1/2*kx1(2),lane1(2),v1+1/2*kv1(2));



    kx1(4)=h*xdot(v1+kv1(3));
    kv1(4)=h*vdot(lane1(1)+kx1(3),lane1(2),v1+kv1(3));




    lane1(1)=lane1(1)+(kx1(1)+2*kx1(2)+2*kx1(3)+kx1(4))/6;
    lane1(2)=10+v2*t*h;

    v1=v1+(kv1(1)+2*kv1(2)+2*kv1(3)+kv1(4))/6;

    if (mod(t,100)==0)
        rkplot(count,:)=[t*h lane1(1) lane1(2) v1 v2];
        count=1+count;
        disp(v2);
        disp(t*h);
        disp(lane1(2));
    end
    if (mod(t,100)==0)
       fprintf(fileID, '%4.7f  %4.7f   \r\n', lane1);

    end


end
```

```
fclose(fileID);
figure
plot(rkplot(:,1),rkplot(:,2),'-',...
    rkplot(:,1),rkplot(:,3),'--','LineWidth',2);
title('Position', 'FontSize', 14)
xlabel('t','FontSize',12)
ylabel('x_i(t)','FontSize', 12)
axis([0 6 0 21])

figure
plot(rkplot(:,1),rkplot(:,4),'-',...
    rkplot(:,1),rkplot(:,5),'--','LineWidth',2);
title('Velocity', 'FontSize', 14)
xlabel('t','FontSize',12)
ylabel('dx_i/dt(t)','FontSize',12)
axis([0 6 -.1 4.2])

c=[lane1(1) lane1(2)];

end
```

A.2.5 Runge-Kutta for Stopped Object and Two Cars

The function RungeKuttaStopped2 performs a Fourth Order Runge-Kutta on the Optimal Velocity Model. This program creates a graph of a cars starting at zero and five approaching an object stopped at 20.

```
function c = RungeKuttaStopped2(  )

lane1=[0 5 20];
```

```
%lane1 is the initial positions of cars
v1=V(lane1(1),lane1(2));
v2=V(5,20);
%v are the initial velocity of the cars
h=.001;
disp(v1);
count=1;
fileID=fopen('RungeKuttaDecel.dat','w');
for t=1:6000

    kx1(1)=h*xdot(v1);
    kv1(1)=h*vdot(lane1(1),lane1(2), v1);
    kx2(1)=h*xdot(v2);
    kv2(1)=h*vdot(lane1(2),lane1(3), v2);

    kx1(2)=h*xdot(v1+1/2*kv1(1));
    kv1(2)=h*vdot(lane1(1)+1/2*kx1(1),lane1(2)+1/2*kx2(1),v1+1/2*kv1(1));
    kx2(2)=h*xdot(v2+1/2*kv2(1));
    kv2(2)=h*vdot(lane1(2)+1/2*kx2(1),lane1(3),v2+1/2*kv2(1));

    kx1(3)=h*xdot(v1+1/2*kv1(2));
    kv1(3)=h*vdot(lane1(1)+1/2*kx1(2),lane1(2)+1/2*kx2(2),v1+1/2*kv1(2));
    kx2(3)=h*xdot(v2+1/2*kv2(2));
    kv2(3)=h*vdot(lane1(2)+1/2*kx2(2),lane1(3),v2+1/2*kv2(2));

    kx1(4)=h*xdot(v1+kv1(3));
    kv1(4)=h*vdot(lane1(1)+kx1(3),lane1(2)+1/2*kx2(3),v1+kv1(3));
    kx2(4)=h*xdot(v2+kv2(3));
    kv2(4)=h*vdot(lane1(2)+kx2(3),lane1(3),v2+kv2(3));


    lane1(1)=lane1(1)+(kx1(1)+2*kx1(2)+2*kx1(3)+kx1(4))/6;
```

```
    lane1(2)=lane1(2)+(kx2(1)+2*kx2(2)+2*kx2(3)+kx2(4))/6;

    v1=v1+(kv1(1)+2*kv1(2)+2*kv1(3)+kv1(4))/6;
    v2=v2+(kv2(1)+2*kv2(2)+2*kv2(3)+kv2(4))/6;

    if (mod(t,100)==0)
        rkplot(count,:)=[t*h lane1(1) lane1(2) v1 v2];
        count=1+count;
    end
    if (mod(t,100)==0)
       fprintf(fileID, '%4.7f  %4.7f   \r\n', lane1);

    end

end
fclose(fileID);
figure
plot(rkplot(:,1),rkplot(:,2),'-',...
    rkplot(:,1),rkplot(:,3),'--',rkplot(:,1),20,'ro-','LineWidth',2);
title('Position','FontSize',14)
xlabel('t','FontSize',12)
ylabel('x_i(t)','FontSize',12)
axis([0 6 0 21])

figure
plot(rkplot(:,1),rkplot(:,4),'-',...
    rkplot(:,1),rkplot(:,5),'--',rkplot(:,1),0,'ro-','LineWidth',2);
title('Velocity','FontSize',14)
xlabel('t','FontSize',12)
ylabel('dx_i/dt(t)','FontSize',12)
axis([0 6 -.1 4.2])
```

```
c=[lane1(1) lane1(2)];

end
```

A.2.6 Runge-Kutta for a Clear Road in Front of Two Cars

The function RungeKuttaClearPathTwoCar performs a Fourth Order Runge-Kutta on the Optimal Velocity Model. This program creates a graph of a cars starting at zero and five approaching a far away object at 100.

```
function c = RungeKuttaClearPathTwoCar(  )

lane1=[0 5 10];
%lane1 is the initial positions of cars
v1=V(lane1(1),lane1(2));
v2=V(lane1(2), lane1(3));

%v are the initial velocity of the cars
h=.001;
disp(v1);
count=1;
fileID=fopen('RungeKuttaDecel.dat','w');
for t=1:6000

    kx1(1)=h*xdot(v1);
    kv1(1)=h*vdot(lane1(1),lane1(2), v1);
    kx2(1)=h*xdot(v2);
    kv2(1)=h*vdot(lane1(2),lane1(3), v2);

    kx1(2)=h*xdot(v1+1/2*kv1(1));
```

```
kv1(2)=h*vdot(lane1(1)+1/2*kx1(1),lane1(2)+1/2*kx2(1),v1+1/2*kv1(1));
kx2(2)=h*xdot(v2+1/2*kv2(1));
kv2(2)=h*vdot(lane1(2)+1/2*kx2(1),lane1(3),v2+1/2*kv2(1));

kx1(3)=h*xdot(v1+1/2*kv1(2));
kv1(3)=h*vdot(lane1(1)+1/2*kx1(2),lane1(2)+1/2*kx2(2),v1+1/2*kv1(2));
kx2(3)=h*xdot(v2+1/2*kv2(2));
kv2(3)=h*vdot(lane1(2)+1/2*kx2(2),lane1(3),v2+1/2*kv2(2));

kx1(4)=h*xdot(v1+kv1(3));
kv1(4)=h*vdot(lane1(1)+kx1(3),lane1(2)+1/2*kx2(3),v1+kv1(3));
kx2(4)=h*xdot(v2+kv2(3));
kv2(4)=h*vdot(lane1(2)+kx2(3),lane1(3),v2+kv2(3));



lane1(1)=lane1(1)+(kx1(1)+2*kx1(2)+2*kx1(3)+kx1(4))/6;
lane1(2)=lane1(2)+(kx2(1)+2*kx2(2)+2*kx2(3)+kx2(4))/6;
lane1(3)=10+3*t*h;

v1=v1+(kv1(1)+2*kv1(2)+2*kv1(3)+kv1(4))/6;
v2=v2+(kv2(1)+2*kv2(2)+2*kv2(3)+kv2(4))/6;

if (mod(t,100)==0)
    disp(lane1(3));
    rkplot(count,:)=[t*h lane1(1) lane1(2) v1 v2  lane1(3) 3];
    count=1+count;
end
if (mod(t,100)==0)
   fprintf(fileID, '%4.7f  %4.7f   \r\n', lane1);

end
```

```
end
fclose(fileID);
figure
plot(rkplot(:,1),rkplot(:,2),'-',...
    rkplot(:,1),rkplot(:,3),'--',rkplot(:,1),rkplot(:,6),'-o','LineWidth',2);
title('Position','FontSize',14)
xlabel('t','FontSize',12)
ylabel('x_i(t)','FontSize',12)
axis([0 6 0 21])

figure
plot(rkplot(:,1),rkplot(:,4),'-',...
    rkplot(:,1),rkplot(:,5),'--',rkplot(:,1),rkplot(:,7),'-o','LineWidth',2);
title('Velocity','FontSize',14)
xlabel('t','FontSize',12)
ylabel('dx_i/dt(t)','FontSize',12)
axis([0 6 -.1 4.2])



c=[lane1(1) lane1(2)];

end
```

A.2.7 Runge-Kutta for Two Cars Following an Object Traveling with a Slow Constant Speed

The function RungeKuttaConstant2 performs a Fourth Order Runge-Kutta on the Optimal Velocity Model. This program creates a graph of cars starting at zero and five approaching an object starting at ten moving at a speed of three.

```
function c = RungeKuttaConstant2(  )

lane1=[0 5 10];
%lane1 is the initial positions of cars
v1=V(lane1(1),lane1(2));
v2=V(lane1(2),lane1(3));
v3=3;
%v are the initial velocity of the cars
h=.001;
disp(v1);
count=1;
fileID=fopen('RungeKuttaConstant.dat','w');
for t=1:6000

kx1(1)=h*xdot(v1);
    kv1(1)=h*vdot(lane1(1),lane1(2), v1);
    kx2(1)=h*xdot(v2);
    kv2(1)=h*vdot(lane1(2),lane1(3), v2);

    kx1(2)=h*xdot(v1+1/2*kv1(1));
    kv1(2)=h*vdot(lane1(1)+1/2*kx1(1),lane1(2)+1/2*kx2(1),v1+1/2*kv1(1));
    kx2(2)=h*xdot(v2+1/2*kv2(1));
    kv2(2)=h*vdot(lane1(2)+1/2*kx2(1),lane1(3),v2+1/2*kv2(1));

    kx1(3)=h*xdot(v1+1/2*kv1(2));
    kv1(3)=h*vdot(lane1(1)+1/2*kx1(2),lane1(2)+1/2*kx2(2),v1+1/2*kv1(2));
    kx2(3)=h*xdot(v2+1/2*kv2(2));
    kv2(3)=h*vdot(lane1(2)+1/2*kx2(2),lane1(3),v2+1/2*kv2(2));

    kx1(4)=h*xdot(v1+kv1(3));
    kv1(4)=h*vdot(lane1(1)+kx1(3),lane1(2)+1/2*kx2(3),v1+kv1(3));
    kx2(4)=h*xdot(v2+kv2(3));
```

```
    kv2(4)=h*vdot(lane1(2)+kx2(3),lane1(3),v2+kv2(3));



    lane1(1)=lane1(1)+(kx1(1)+2*kx1(2)+2*kx1(3)+kx1(4))/6;
    lane1(2)=lane1(2)+(kx2(1)+2*kx2(2)+2*kx2(3)+kx2(4))/6;

    v1=v1+(kv1(1)+2*kv1(2)+2*kv1(3)+kv1(4))/6;
    v2=v2+(kv2(1)+2*kv2(2)+2*kv2(3)+kv2(4))/6;

    lane1(3)=10+v3*t*h;



    if (mod(t,100)==0)
        rkplot(count,:)=[t*h lane1(1) lane1(2) v1 v2 lane1(3)];
        count=1+count;
        disp(v2);
        disp(t*h);
        disp(lane1(2));
    end
    if (mod(t,100)==0)
       fprintf(fileID, '%4.7f  %4.7f   \r\n', lane1);

    end



end
fclose(fileID);
figure
plot(rkplot(:,1),rkplot(:,2),'-',...
    rkplot(:,1),rkplot(:,3),'--',rkplot(:,1),rkplot(:,6),'ro-','LineWidth',2);
title('Position','FontSize',14)
xlabel('t','FontSize',12)
```

```
ylabel('x_i(t)','FontSize',12)
axis([0 6 0 21])

figure
plot(rkplot(:,1),rkplot(:,4),'-',...
    rkplot(:,1),rkplot(:,5),'--',rkplot(:,1),3,'ro-','LineWidth',2);
title('Velocity','FontSize',14)
xlabel('t','FontSize',12)
ylabel('dx_i/dt(t)','FontSize',12)
axis([0 6 -.1 4.2])

c=[lane1(1) lane1(2)];

end
```