MECH454 MATLAB Information

Fall 2016

Background

For many of the assignments in this class, we will be asking you to simulate the behavior of a vehicle under a specified set of conditions. As you might guess, we can model the behavior of the vehicle with an ordinary differential equation. Thus, programming the simulation will require you to write code that will find the solution to this ode. We could use MATLAB's family of ode solvers (ode45, ode15s, etc). These are very powerful solvers, with a host of options for customizing their behavior. However, we have found that for the purposes of MECH454, these solvers reduce the transparency of students' code and make it more difficult to debug wayward simulations.

As a result, we are going to have you write your own code to solve the ode's. We will keep it simple, utilizing a forward Euler integration technique. Hopefully you have already been exposed to this construct, but if not, read on!

Euler Integration

We've already said that our vehicle models are going to be expressed in terms of differential equations. Thus, the vehicle states are simply functions of time with some initial conditions, i.e. $y(t) = f(t)$ with $y(t_0) = y_0$. We'll assume that we know the vehicle states at some time (usually zero). If we can figure out how to propagate this function forward (or backward) in time, we can determine the vehicle behavior at any time.

Now, more generically, if we have a function $f(t)$, we can express it by using its Taylor expansion.

$$f(t) = f(a) + \frac{f'(a)}{1!}(t-a) + \frac{f''(a)}{2!}(t-a)^2 + \dots$$

If, instead of having $f(t)$ and a solution at a, we say that we have $y(t)$ which is parameterized by t and that y_1 is the solution to the differential equation at some known time t_1, we again have the general solution to our differential equation. We will drop the second-order and higher terms since they are quadratic (and higher) in the small quantity $(t-a)$, meaning that they become very small when raised to higher powers. Hence, we're left with

$$y(t) \approx y(t_1) + (t - t_1)\dot{y}$$

which is an approximation for the function $y(t)$ based on knowledge of the function value y_1 at time t_1. This would be exact if we hadn't dropped the higher order terms, but since we did, we know that as we get farther away from time t_1, we should expect our approximation to the solution to have more error. Finally, if we define an interval $\Delta t = t - t_1$ and say that we have a discrete update to the system states at every time step Δt, we can write the state approximation as $y(k+1) = y(t+\Delta t)$. Then the update equation for each successive time step is

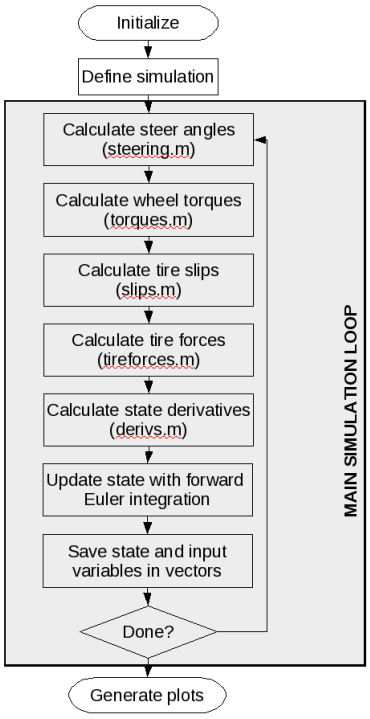
$$y(k+1) \approx y(k) + \dot{y}\Delta t$$

which merely says that the value of the state at the next time step is equal to the current value of the state plus the derivative multiplied by the length of the time step. We'll use this result to program our simulations. Remember the little detail about the error in our approximation. Later on in the course, we will have you program a simulation that includes the spin dynamics of the wheels. When we get there, you will find out that those errors can add up quite quickly and you will need to use a very short time step (on the order of $\Delta t = 0.0001$) to get accurate results. This is where the MATLAB ode solvers do a nice job. Since they incorporate a variable step size integration technique, they can get high accuracy without using the extremely small step throughout the entire simulation. This improves speed, but our simulations are small enough that the benefit of the advanced solver is overwhelmed by its lack of transparency and most PC's will have plenty of processing power and memory to handle our Euler integration approach.

Implementation

Now, on to some of the details. We will be asking you to write code to find solutions to quite a few combinations of vehicle and tire models under many different conditions throughout the quarter. As such, we're going to help you set up a structure for your code and files that will help you to maximize the utility of each line of code you write as well as make it easier for you to debug and us to help you debug. Since we're spending the time to set this all up, we will be strongly encouraging you to follow our code standards. We won't stop you from developing your own system, but we can only provide you assistance with your code in office hours if you adhere to our standards.

One very important programming practice is to break up your code into small "atomic" blocks. These are blocks of code that have one specific function. If you write them well, you can debug them once and then use them over and over again with confidence. We're going to have you write several of these "atomic" blocks of code for the first assignment that you'll be able to reuse throughout the quarter. The following flowchart shows how you will use five of these files to implement your Euler integration scheme.



Let's walk through the code execution described by this flowchart. All of the code for a simulation should be implemented in a separate file. For example, for homework 1, you might be working on problem 2. Thus, you should create a file called something like `hw1_prob2.m`. The first thing you want to do in this script file (after putting your name, the date, and a description in the header comments) is to clear everything, close any old plots, and clear the command line.

After doing this, you can move on to setting up the simulation. This can include things such as setting simulation parameters, creating a time vector, and loading your vehicle parameters from an external file. Once all of the initialization is complete, you will need to define a "for loop" where you loop through all the entries in the time vector, computing the desired vehicle dynamics. As shown in the flowchart, the main loop consists mainly of making calls to external functions. This allows you to make use of your "atomic" blocks of code. Hint: if your "for loop" contains more than 10 lines of code, you're probably not taking full advantage of this structure.

One way that we will make your code highly flexible is to consider the behavior of all four wheels, regardless of the model that we are using. To make things easy, we're going to introduce the following enumeration for the wheels.

Wheel	**Abbreviation**	**Number**
Left Front	lf	1
Right Front	rf	2
Left Rear	lr	3
Right Rear	rr	4

This will be easily implemented by including the following line of code in all of your functions and scripts:

```
lf=1; rf=2; lr=3; rr=4;
```

This will allow you to reference a specific element such as

```
Fy(lr) = -Ca(lr)*alpha(lr);
```

and even write a "for loop" for all four wheels as

```
for wheel=lf:rr
  Fy(wheel) = -Ca(wheel)*alpha(wheel);
end
```

which will loop through all four wheels, performing the calculation you specify inside the loop.

For some models , this will mean that you'll have to explicitly write the summation of the wheel forces when we treat them as a lumped axle. So, to implement this, you will use the steering function to get the steering angles for all four wheels, so the call will be something like `delta = steering(simtime,state,simulation,driver)` where simtime is the time step of the simulation for which you are calculating the steering angles. Simulation and driver are structures that define properties of the simulation and driver actions (and will be described more in the next subsection). The output variable, delta, is a vector of the steering angles, enumerated as given above. Similarly, torques should return a vector of the torques on each wheel, enumerated as described above. Now, armed with the vehicle state and the driver inputs, you can make your call to the slips function. This should be something like `kappa = slips(state,delta,torques,simulation,vehicle)` where delta is the output vector from the steering function and kappa is a vector of slips for each wheel (again, enumerated in the given order).

You will continue to make calls in the same manner to the tireforces and derivs functions. The tireforces function requires the slips at minimum, though more arguments will be added for more complex tire models. From the tireforces function, you will get out a vector of lateral forces for each wheel. This will, as you might have guessed, serve as input to your derivatives function along with the state vector. The output of your derivatives function provides the information you need to calculate the state at the next time step according to the Euler integration scheme. At this point, you start the loop execution again, using the states you just calculated to get the next time step going.

Note that as you go through the loop you will have access to all of the intermediate variables that you calculate. For some parts of assignments, you will find it useful to save these in vectors so that you can plot them or find minima and maxima. However, for any variable that you save in the loop, you should make sure that you preallocate space for the data. MATLAB will allow you to grow a matrix or vector in a loop, but every time it changes the size of the data, it copies it all over into a new location in memory. Your code will run much faster if you initialize enough space for the whole thing using a command such as `state = zeros(numStates,numTimeSteps)`. Similarly, MATLAB accesses data much more quickly if it is in columns, rather than rows. Therefore, if you have the choice, try to store your data in column vectors. Again, preallocation can help you here since MATLAB won't always automatically choose the best format for your data.

Useful MATLAB Constructs

In addition to these tricks to speed up code execution, we would like to point out some useful MATLAB features and programming constructs that we recommend using to write your code. The first is the idea of a MATLAB structure. These are very simple to implement. If you write `dog.color = white` and `dog.height = 12` then you have defined a structure dog that has fields (attributes) color and height that are part of dog.

We have three specific structures in mind for your simulation code. The first is for the vehicle. Obviously, the vehicle will have physical properties such as mass and length, etc. that we will want to use. Defining a single structure to contain all of these properties will allow you to pass the structure as an argument into functions where you can then use any of the properties at will.

vehicle:
vehicle.m = 1737
vehicle.Ca = [90000; 90000; 108000; 108000]
vehicle.Cx = [40000; 40000; 40000; 40000]
vehicle.Izz = 3600
vehicle.L = 2.709
vehicle.a = 1.168
vehicle.b = 1.541
vehicle.mu_peak = 1.2
vehicle.mu_slide = 0.8

The second structure is for the simulation itself. Defining fields to keep track of the vehicle model, tire model, and even convenient physical constants, such as the acceleration due to gravity (useful for converting mass to weight) will also help you to keep your simulations clean and flexible.

simulation:
simulation.vmodel = {'bike'}
simulation.tmodel = {'linear','fiala_lat'}
simulation.speed = 20
simulation.g = 9.81

Note that the vmodel and tmodel fields can take on a few discrete values. These values are used to keep track of the model being used and will also be used in your "atomic" functions to determine which snippet of code to use to calculate the desired vehicle quantities. The third structure is intended to represent the driver. We will be simulating the driver in many of our simulations, either in open-loop maneuvers or with a simple model of the driver following a path. Storing maneuver parameters and driver gains in this structure will help you to modify your simulations as the complexity of the driver's action grows. For the first assignment, you will use the following fields in your structures:

driver:
driver.steer_mode = {'step','data'}
driver.delta0 = 0
driver.deltaf = $\frac{2\pi}{180}$
driver.steertime = 0.1

Here the steer_mode variable can be used to tell the steering function which type of open loop steering maneuver to perform or can also be used to set a steering controller, like lanekeeping. The rest of the fields will change to provide the parameters and data necessary for the steering mode. In this case, we have the data for the step steer maneuver. If steer_mode is set to 'data', then the driver structure should contain a pair of vectors to the steering angles and the associated times. The steering controller can then interpolate to get the steering angles for any time in the simulation.

Now that we've defined the structures we want you to use, the second MATLAB construct is the **switch** statement. This construct allows you to switch between different segments of code depending on the value of a single variable. In MATLAB, you would implement a **switch** statement like this:

```
switch lower(driver.steer_mode)
    case 'step'
        % Code to implement step steer goes here
    case 'data'
        % Use interp1(...) to interpolate for steer angles
    otherwise
        error('Steering mode not implemented')
end
```

This switch statement, dependent on the variable driver.steer_mode, will check to see if there is a match to step, or data. The function lower adds robustness, since it ensures that the variable driver.steer_mode is a lowercase string. If one of these matches, the appropriate steering angles are calculated. If no matches are made, then execution falls through to the default case, where the message indicates this. A switch statement can be written without a default case, but it is generally a bad idea since no code will be executed and you may not even realize it.

A third MATLAB construct that you may not have seen before is a variable argument list. A MATLAB function is defined in the first line at the top of the file. It must have an output argument list as well as an input argument list. However, MATLAB also provides us with a way to allow the function to accept or output argument lists that may change depending on the context in which the function is used. The general syntax for this is

```
function varargout = functionName(firstArgument,secondArgument,varargin)
```

What this tells MATLAB is to construct a cell-array for any inputs beyond the first two. On the output side, you can choose how many outputs you want to send back out of the function, held in another cell-arry. A cell-array is just like a vector, except that each element can itself be anything (text, a vector, a matrix, etc.). You can reference an element of a cell-array by writing `cellArrayName{k}` where k is the element number you want (note the curly braces). So for the general function that we've given above, there are two static arguments, followed by the variable argument list. Thus, to get the third input argument, you can write `varargin{1}`. This will allow you to write a function (like tireforces) that will take different arguments, depending on mode in which you call the function.

This means the function will allow you to call the function with 2 or more arguments. You'll get an error if you have less than 2, but none if you have more. Thus, you should

make sure you do your own error checking when using varargin. After you get done error checking and calculating your outputs, you need to tell MATLAB how to send the values back out of the function. Again, this is easy since all you need to do is assign them to the correct locations in the varargout variable. So if your function call is

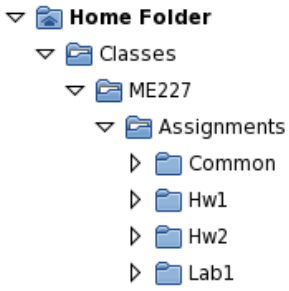
```
[output1,output2] = functionName(firstArgument,secondArgument,thirdArg)
```

then you will need to put the value you want to end up in output1 in `varargout{1}` and the value for output2 in `varargout{2}`. This will be useful as you add functionality to tireforce.m and a few other functions since you will start out initially calculating only the lateral tire forces, but soon move to calculating both lateral and longitudinal forces.

The final construct is the idea of a user-written function help. You probably already know that you can get help in MATLAB by typing `help <function>`, but you may not realize that you can help yourself by writing help text for your own functions. For functions that you write, if you type `help <your function>`, MATLAB will output the block of comments immediately following the function definition line. Any line beginning with a `%` will be printed, up until a blank line or a MATLAB command is given. To give an example, your tireforce function will take various different arguments, depending on which tire model you request with the variable simulation.vmodel. To save yourself from having to go open tiremodel.m every time you make the change, include the syntax for calling the tiremodel function in the help text. That way, you can just type `help tireforce` to have MATLAB remind you how to use the function. Try typing `help steering` with "Common" set as your MATLAB working directory for a demonstration of how this works. This can be extremely helpful down the road since you can put the syntax for calling the function in the help text. That way, you'll only have to do `help <your function>` to remember the order of inputs or outputs.

Files and Directories

Now, about that directory structure. Since we want you to be able to reuse as much code as possible, we strongly suggest the following structure. This relatively flat directory struc-



ture will allow you to reference files that you put in "Common" by using the command `addpath('../Common')` at the beginning of any script that resides in the homework or lab folders.

In the "Common" directory, you will need to have the following three files: steering.m, torques.m, slips.m, and tireforces.m. You will definitely be adding more files to "Common"

as you do the homeworks, but these three are absolutely necessary for the first simulation. These files should be written with great care, paying attention to all of the hints about structure and error checking that we give you. Time spent making the functions in these files flexible and powerful will be time saved over and over again during the quarter.

To speed up the development of your software, we are providing you with a zip archive that has the suggested directory structure and some skeleton files that will get you started on the code for the first assignment.

Getting Started

Now that you've read through all this, you're ready to get started on Simulation #1. Go to the Moodle website and download the zip archive. Choose your favorite location for MECH454 files and unpack the archive. Read through all the files to familiarize yourself with the skeleton files. There are quite a few hints in the comments and even in the file names and organization. Use these to your advantage, and good luck!

Summary of Constructs

Here is a summary of the constructs needed for this assignment. There are three main types: vectors, structures, and functions. If the usage of these constructs is unclear, please re-read the relevant sections in this document. We have tried to make things as clear as possible for you. If you still can't figure out how to make use of these constructs, please come see us in office hours. We'd rather that you take the time to learn how to use them now so that the rest of your time spent coding simulations this quarter will be productive and meaningful.

Vectors

You will need to define the following vectors for this assignment.

$$state = \begin{bmatrix} U_y \\ r \end{bmatrix} \quad delta = \begin{bmatrix} \delta_{lf} \\ \delta_{rf} \\ \delta_{lr} \\ \delta_{rr} \end{bmatrix} \quad alpha = \begin{bmatrix} \alpha_{lf} \\ \alpha_{rf} \\ \alpha_{lr} \\ \alpha_{rr} \end{bmatrix} \quad Fy = \begin{bmatrix} Fy_{lf} \\ Fy_{rf} \\ Fy_{lr} \\ Fy_{rr} \end{bmatrix} \quad Fz = \begin{bmatrix} Fz_{lf} \\ Fz_{rf} \\ Fz_{lr} \\ Fz_{rr} \end{bmatrix}$$

Remember that $lf = 1, rf = 2, lr = 3, rr = 4$.

Structures

The following structure fields will be necessary for this assignment.

vehicle:
vehicle.m = 1737
vehicle.Ca = [180000; 180000; 216000; 216000]
vehicle.Cx = [40000; 40000; 40000; 40000]
vehicle.Izz = 3600
vehicle.L = 2.709
vehicle.a = 1.168
vehicle.b = 1.541
vehicle.mu_peak = 1.2
vehicle.mu_slide = 0.8

simulation:
simulation.vmodel = 'bike'
simulation.tmodel = {'linear','fialalat'}
simulation.speed = 20
simulation.g = 9.81

Note that when you change tmodel from 'linear' to 'fialalat', you will also need to change the arguments to your tireforce(...) function.

driver:
driver.steer_mode = {'step'}
driver.delta0 = 0
driver.deltaf = $\frac{2\pi}{180}$
driver.steertime = 0.1

or

driver:
driver.steer_mode = {'data'}
driver.t_exp = {experimental data - time}
driver.delta_exp = {experimental data - delta}

Note that when you choose steer_mode to be 'step', you will need to define delta0 (the initial steer angle before the step), deltaf (the final steer angle after the step), and steertime (the time to do the step). If you choose steer_mode to be 'data', you will need the fields t_exp (the time vector for the input steering data) and delta_exp (the input steering data). Your steering code will use these two vectors to interpolate for the steer angles you need in your simulation.

Functions

The following function calls will be necessary for this assignment.

```
delta = steering(simulation,driver,state,simtime)
alpha = slips(simulation,vehicle,state,delta)
Fy = tireforces(simulation,vehicle,alpha)
```

or

```
Fy = tireforces(simulation,vehicle,alpha,Fz)
```

and

```
dxdt = derivs(simulation,vehicle,state,Fy)
```