Franklin W. Olin College of Engineering

MTH3199 Applied Math for Engineers – Fall 2024
Assignment 4: Numerical Integration Continued

Assigned: Friday, October 18, 2024.
Lab Report Due: Monday, November 4, 2024 (11:59 PM EST).

Online Resources

- Variable Step Size Methods: LibreTexts
- Runge-Kutta Overview: Wikipedia, John D. Cook
- List of Runge-Kutta Methods Wikipedia
- Dormand-Prince: Wikipedia.

Generalization: Runge-Kutta Methods

(Friday, October 18th) activity. Please complete before class on Tuesday, October 22nd.
The methods that we've implemented so far (forward/backward Euler and explicit/implicit midpoint) belong to a broader category of integration methods called Runge-Kutta methods. Given some rate function, $\dot{X} = f(t, X)$, the value at the current time-step, X_n, and a time step length, h, an **explicit** Runge-Kutta method computes X_{n+1} at the next time-step using following sequence (equations sourced from Wikipedia):

$$k_1 = f(t_n, X_n) \tag{1}$$
$$k_2 = f(t_n + c_2 h, X_n + h(a_{21} k_1)) \tag{2}$$
$$k_3 = f(t_n + c_3 h, X_n + h(a_{31} k_1 + a_{32} k_2)) \tag{3}$$
$$\vdots \tag{4}$$
$$k_i = f\left(t_n + c_i h, X_n + h \sum_{j=1}^{i-1} a_{ij} k_j\right) \tag{5}$$
$$X_{n+1} = X_n + h \sum_{i=1}^{s} b_i k_i \tag{6}$$

Here, k_i act as a series of intermediate guesses for $X(t)$, which are combined at the end to form X_{n+1}. b_i, c_i, and a_{ij} are all constants that depend on the specific method being implemented. This equation probably seems like a lot of gobbledygook, so let's look at two examples:

Forward Euler: We can express forward Euler in the Runge-Kutta form as follows:

$$k_1 = f(t_n, X_n) \tag{7}$$
$$X_{n+1} = X_n + h\,(1 \cdot k_1) \tag{8}$$

In this case, we just have $c_1 = 0$, $b_1 = 1$, $a_{11} = 0$.

Explicit Midpoint: We can express explicit midpoint in the Runge-Kutta form as follows:

$$k_1 = f(t_n, X_n) \tag{9}$$
$$k_2 = f(t_n + .5h, X_n + h(.5k_1)) \tag{10}$$
$$X_{n+1} = X_n + hk_2 \tag{11}$$

In this case, we get:

$$c_1 = 0, \quad c_2 = .5, \quad a_{21} = .5, \quad b_1 = 0, \quad b_2 = 1 \tag{12}$$

Each Runge-Kutta method is specified by the constants a_{ij}, b_i, and c_j:

$$\begin{array}{c|cccccc} 0 & 0 & 0 & ... & 0 & 0 \\ c_2 & a_{21} & 0 & ... & 0 & 0 \\ c_3 & a_{31} & a_{32} & ... & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ c_s & a_{s1} & a_{s2} & ... & a_{s,s-1} & 0 \\ \hline & b_1 & b_2 & ... & b_{s-1} & b_s \end{array} \tag{13}$$

The table displayed above (and sourced from Wikipedia) is called a Butcher tableau, and is a very succinct way of visualized a Runge-Kutta method. The Butcher tableau for forward Euler is given by:

$$\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array} \tag{14}$$

The tableau for explicit midpoint is shown below:

$$\begin{array}{c|cc} 0 & 0 & 0 \\ .5 & .5 & 0 \\ \hline & 0 & 1 \end{array} \tag{15}$$

Your first task is to write a function that takes a Butcher tableau as input, and computes a single step for any arbitrary Runge-Kutta method:

```
%This function computes the value of X at the next time step
%for any arbitrary RK method
%INPUTS:
%rate_func_in: the function used to compute dXdt. rate_func_in will
%              have the form: dXdt = rate_func_in(t,X)  (t is before X)
%t: the value of time at the current step
%XA: the value of X(t)
%h: the time increment for a single step i.e. delta_t = t_{n+1} - t_{n}
%BT_struct: a struct that contains the Butcher tableau
%   BT_struct.A: matrix of a_{ij} values
%   BT_struct.B: vector of b_i values
%   BT_struct.C: vector of c_i values
%OUTPUTS:
%XB: the approximate value for X(t+h) (the next step)
%    formula depends on the integration method used
%num_evals: A count of the number of times that you called
%           rate_func_in when computing the next step
function [XB, num_evals] = explicit_RK_step(rate_func_in,t,XA,h,BT_struct)
    %your code here
end
```

A few implementation notes:

- It's probably cleaner to use a MATLAB struct to pass in the values of the Butcher tableau, but it's up to you (if you'd like the step function to take A, B, and C, as separate inputs, go for it).

- It may be convenient to think of following sums as dot products:

$$\sum_{j=1}^{i-1} a_{ij}k_j, \qquad \sum_{i=1}^{s} b_i k_i \tag{16}$$

The first sum being the dot product of the vector of k's, $K = [k_1; k_2; ...; k_s]$ with a row of A, and the second sum being the dot product of K and B. As such, I would recommend that you begin your implementation by initializing a $m \times s$ matrix of zeros for K (where m is the height of X and s is the number of stages for that method), and then updating each column sequentially. This will allow you to compute the sums as follows:

```
sum_val1 = K*(A(i,:)');
sum_val2 = K*B;
```

- How can you figure out what num_evals should be?

Now that you have a single step implemented, it's time to write a fixed-step integrator:

```
%Runs numerical integration arbitrary RK method
%INPUTS:
%rate_func_in: the function used to compute dXdt. rate_func_in will
%              have the form: dXdt = rate_func_in(t,X)  (t is before X)
%tspan: a two element vector [t_start,t_end] that denotes the integration endpoints
%X0: the vector describing the initial conditions, X(t_start)
%h_ref: the desired value of the average step size (not the actual value)
%BT_struct: a struct that contains the Butcher tableau
%   BT_struct.A: matrix of a_{ij} values
%   BT_struct.B: vector of b_i values
%   BT_struct.C: vector of c_i values
%OUTPUTS:
%t_list: the vector of times, [t_start;t_1;t_2;...;.t_end] that X is approximated at
%X_list: the vector of X, [X0';X1';X2';...;(X_end)'] at each time step
%h_avg: the average step size
%num_evals: total number of calls made to rate_func_in during the integration
function [t_list,X_list,h_avg, num_evals] = explicit_RK_fixed_step_integration ...
                                            (rate_func_in,tspan,X0,h_ref,BT_struct)
    %your code here
end
```

This really shouldn't be any different from the fixed-step integration schemes that you have written so far.

Testing

So far, our test ODE's have all been nice and linear. It's time to throw a nonlinear problem into the mix: orbiting bodies. Consider a planet of mass m_p that is orbiting (in the xy plane) around a star of mass m_s. The star, which is located at the origin $(0, 0)$, has significantly more mass than the planet, and can therefore be assumed to be stationary. By Newton's law of universal gravitation, the force acting on the planet is given by:

$$m_p \frac{d^2\vec{r}}{dt^2} = \vec{F} = -\frac{m_p m_s G}{|\vec{r}|^3}\vec{r} \tag{17}$$

where $\vec{r} = (x_p, y_p)$ is the current position of the planet, and G is the gravitational constant. Write the corresponding rate function in MATLAB:

```
%Rate function describing Newton's law of gravitation
%INPUTS:
%t: the current time
%V: the vector of the position and velocity of the planet
%   V = [x_p; y_p; dxdt_p; dydt_p]
```

```
%orbit_params: a struct describing the system parameters
%   orbit_params.m_sun is the mass of the sun
%   orbit_params.m_planet is the mass of the planet
%   orbit_params.G is the gravitational constant
%OUTPUTS:
%dVdt: a column vector describing the time derivative of V:
%   dVdt = [dxdt_p; dydt_p; d2xdt2_p; d2ydt2_p]
function dVdt = gravity_rate_func(t,V,orbit_params)
    %your code here
end
```

We will be testing our integration methods on this ODE. For ground truth, I have written a function that uses secant method to solve for the exact position/velocity of the planet at any given time:

```
%this function computes the orbit of a planet about a sun
%the sun is assumed to located at the origin (0,0)
%and motion is restricted to the x-y plane
%INPUTS:
%t_list: a list of times to compute the position & velocity of the planet
%V0: initial condition.  V0 is a column vector consisting
%   of the initial position and velocity of the planet:
%       V0 = [x(0); y(0); dx/dt(0); dy/dt(0)]
%orbit_params: a struct describing the system parameters
%   orbit_params.m_sun: mass of the sun
%   orbit_params.m_planet: mass of the planet
%   orbit_params.G: gravitational constant
%       Force = -m_planet*m_sun*G/r^2
%OUTPUTS:
%V_list: the state of the planet at each time in t_list
%   if t_list is a list, then V_list is a Nx4 MATRIX
%   where each ROW has the form [x_i,y_i,dx/dt_i,dy/dt_i]
%   corresponding to the values at the ith time
%   if t_list is a SCALAR (i.e. t_list = t),
%   then V_list is a COLUMN VECTOR of the form:
%   [x(t); y(t); dx/dt(t); dy/dt(t)]
%NOTES:
%This function needs all the other functions in this file to run
%I HIGHLY RECOMMEND JUST SAVING THIS FUNCTION IN ITS OWN FILE
%DON'T COPY AND PASTE INTO YOUR CODE! IT'S NOT WORTH IT!
%
%USAGE EXAMPLE:
%At the bottom of this file is a function usage_example()
%which shows how to use compute_planetary_motion(...)
%You can start from there and then tweak it.
function V_list = compute_planetary_motion(t_list,V0,orbit_params)
    %Orion's special code here
end
```

The entire function is a bit too long for this document, so you can find it on the Canvas page. Choose **three or more** different explicit Runge-Kutta methods to test. You can find a list of methods on this Wikipedia page. Ideally, there will be a few different orders of truncation error that are represented from among the methods that you chose. Perform a set of experiments comparing the behavior of the different methods. This is a bit open ended. I really just want to make sure you validate that your implementation works before we move on to the next stage. Example deliverables include:

- A plot comparing the approximate solution for each method to the true solution for a given time step-length.
- A log log plot showing how the **local** truncation error (and the difference $|X(t+h) - X(t)|$) scales with the step-size, h (with fit-lines).
- A table showing how the **local** truncation error (and the difference $|X(t+h) - X(t)|$) scales with the step-size, h (i.e. the values of p).
- A log log plot showing how the **global** truncation error scales with the step-size, h (with fit-lines).

- A table showing how the **global** truncation error scales with the step-size, h (i.e. the values of p).
- A log log plot showing how the **global** truncation error scales with num_evals (with fit-lines).
- A table showing how the **global** truncation error scales with num_evals (i.e. the values of p).

Conservation of Physical Quantities

The mechanical energy, E, and angular momentum, H, of the planet are conserved throughout its orbit (these quantities do not change over time).

$$E = \frac{1}{2} m_p \left(\dot{x}^2 + \dot{y}^2\right) - \frac{m_s m_p G}{|\vec{r}|}, \qquad H = m_p \left(x\dot{y} - y\dot{x}\right) \tag{18}$$

Design one or more experiments to quantify how effective an integration method is at conserving a physical quantity. A few things to consider:

- How can we isolate our evaluation from a possible dependence on the initial conditions of the ODE?
- What does "effective" mean in this case?
- Reducing the step-size, h, probably results in smaller changes in the conserved quantities, regardless of the method. How can we control for that in our experiment(s)?
- It might be worthwhile (but not required) to test the effectiveness of one or both of the implicit methods that you've implemented.
- For those among you who might be interested, try implementing a leapfrog integrator and comparing its effectiveness with the other methods.

I don't have any particular experiment in mind here, so I expect you to come up with something good!

Introduction to Adaptive Step Sizes

(Tuesday, October 22nd) activity. Please complete before class on Friday, October 25th.
Now that we have completed our initial implementation of Runge-Kutta, it's time to add some dynamic step sizing. There are two clear advantages of being able to choose step sizes dynamically. First, you don't have to worry about having to figure out the necessary step size in order to achieve the desired level of accuracy. Second, we can dynamically shrink the step size during volatile regions of the integration, and increase the step size when not much is changing. We will implement this feature by computing two different estimates for our state variables at the next time step, and use the difference between these two estimates as a proxy for the local truncation error. To accomplish this, our first order of business is to implement an **embedded** Runge-Kutta method. Essentially, this is just like a regular RK method, except that we will be generating two different linear combinations of k_i at the end:

$$k_1 = f(t_n, X_n) \tag{19}$$
$$k_2 = f(t_n + c_2 h, X_n + h(a_{21} k_1)) \tag{20}$$
$$k_3 = f(t_n + c_3 h, X_n + h(a_{31} k_1 + a_{32} k_2)) \tag{21}$$
$$\vdots \tag{22}$$
$$k_i = f\left(t_n + c_i h, X_n + h \sum_{j=1}^{i-1} a_{ij} k_j\right) \tag{23}$$
$$X_{n+1} = X_n + h \sum_{i=1}^{s} b_{1,i} k_i \tag{24}$$
$$X^*_{n+1} = X_n + h \sum_{i=1}^{s} b_{2,i} k_i \tag{25}$$

This corresponds to an extra row at the bottom of the Butcher tableau:

$$\begin{array}{c|ccccc} 0 & 0 & 0 & \dots & 0 & 0 \\ c_2 & a_{21} & 0 & \dots & 0 & 0 \\ c_3 & a_{31} & a_{21} & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ c_s & a_{s1} & a_{s1} & \dots & a_{s,s-1} & 0 \\ \hline & b_{1,1} & b_{1,2} & \dots & b_{1,s-1} & b_{1,s} \\ & b_{2,1} & b_{2,2} & \dots & b_{2,s-1} & b_{2,s} \end{array} \tag{26}$$

```
%This function computes the value of X at the next time step
%for any arbitrary embedded RK method
%INPUTS:
%rate_func_in: the function used to compute dXdt. rate_func_in will
%              have the form: dXdt = rate_func_in(t,X)  (t is before X)
%t: the value of time at the current step
%XA: the value of X(t)
%h: the time increment for a single step i.e. delta_t = t_{n+1} - t_{n}
%BT_struct: a struct that contains the Butcher tableau
%   BT_struct.A: matrix of a_{ij} values
%   BT_struct.B: vector of b_i values
%   BT_struct.C: vector of c_i values
%OUTPUTS:
%XB1: the approximate value for X(t+h) using the first row of the Tableau
%XB2: the approximate value for X(t+h) using the second row of the Tableau
%num_evals: A count of the number of times that you called
%           rate_func_in when computing the next step
function [XB1, XB2, num_evals] = RK_step_embedded(rate_func_in,t,XA,h,BT_struct)
    %your code here
end
```

Once this has been constructed, we need to answer a few questions:

- How do XB1, XB2, and their difference $|\text{XB1} - \text{XB2}|$ scale with the step size, h?
- Which of the two approximations is more accurate?
- Is the difference, $|\text{XB1} - \text{XB2}|$ a good proxy for the local truncation error?

Choose an embedded RK method (list here) to evaluate. I've provided the Butcher Tableau's for a few of these:

```
DormandPrince = struct();
DormandPrince.C = [0, 1/5, 3/10, 4/5, 8/9, 1, 1];
DormandPrince.B = [35/384, 0, 500/1113, 125/192, -2187/6784, 11/84, 0;...
                5179/57600, 0, 7571/16695, 393/640, -92097/339200, 187/2100, 1/40];
DormandPrince.A = [0,0,0,0,0,0,0;
                1/5, 0,  0, 0,0,0,0;...
                3/40, 9/40, 0, 0, 0, 0,0;...
                44/45, -56/15, 32/9, 0, 0, 0,0;...
                19372/6561, -25360/2187, 64448/6561, -212/729, 0, 0,0;...
                9017/3168, -355/33, 46732/5247, 49/176, -5103/18656, 0,0;...
                35/384, 0, 500/1113, 125/192, -2187/6784, 11/84,0];

Fehlberg = struct();
Fehlberg.C = [0, 1/4, 3/8, 12/13, 1, 1/2];
Fehlberg.B = [16/135, 0, 6656/12825, 28561/56430, -9/50, 2/55;...
              25/216, 0, 1408/2565, 2197/4104, -1/5, 0];
Fehlberg.A = [0,0,0,0,0,0;...
              1/4, 0,0,0,0,0;...
              3/32, 9/32, 0,0,0,0;...
              1932/2197, -7200/2197, 7296/2197, 0,0,0;...
```

```
              439/216, -8, 3680/513, -845/4104, 0,0;...
              -8/27, 2, -3544/2565, 1859/4104, -11/40, 0];

HeunEuler = struct();
HeunEuler.C = [0,1];
HeunEuler.B = [1/2,1/2;1,0];
HeunEuler.A = [0,0;1,0];

FehlbergRK1 = struct();
FehlbergRK1.C = [0,1/2,1];
FehlbergRK1.B = [1/512, 255/256, 1/512;...
                 1/256, 255/256, 0];
FehlbergRK1.A = [0,0,0;1/2,0,0;1/256,255/256,0];

Bogacki = struct();
Bogacki.C = [0,1/2, 3/4, 1];
Bogacki.B = [2/9, 1/3, 4/9, 0; 7/24, 1/4, 1/3, 1/8];
Bogacki.A = [0,0,0,0; 1/2,0,0,0; 0,3/4,0,0; 2/9,1/3, 4/9, 0];
```

For your chosen method, do the following:

- For testing, choose one of the existing rate function + solution function pairs that have been provided in the or last assignment. Alternatively, you can write one of your own.
- Generate a loglog plot of the local truncation error for XB1 and XB2 as a function of the step size, h. On the same axes, plot $|\text{XB1} - \text{XB2}|$ as a function of h, as well as the difference $|f(t_{ref} + h) - f(t)|$.
- How do the local truncation errors of XB1 and XB2 scale with h (find the corresponding values of p). Which of the two has better accuracy? How does $|\text{XB1} - \text{XB2}|$ scale with h?
- Plot the local truncation errors of XB1 and XB2 as a function of their difference, $|\text{XB1} - \text{XB2}|$. How do the errors scale with $|\text{XB1} - \text{XB2}|$? Is $|\text{XB1} - \text{XB2}|$ a good proxy for the error (or at least, does it provide a good upper bound on the error?).

At this point, you will probably have noticed that $|\text{XB1} - \text{XB2}|$ scales with some integer power of the step-size:

$$|\text{XB1} - \text{XB2}| \propto h^p \tag{27}$$

If we use $|\text{XB1} - \text{XB2}|$ as a proxy for the error, ϵ, then we see that the estimated error will also scale in the same way with h:

$$\epsilon \propto h^p \tag{28}$$

Suppose that for the current step size, h_c, the local truncation error has been measured at ϵ_c, and that our desired truncation error is given by ϵ_d. We can compute our desired step size, h_d, as follows:

$$\epsilon_c = kh_c^p, \quad \epsilon_d = kh_d^p \;\rightarrow\; \frac{\epsilon_d}{\epsilon_c} = \frac{kh_d^p}{kh_c^p} = \left(\frac{h_d}{h_c}\right)^p \tag{29}$$

$$\left(\frac{\epsilon_d}{\epsilon_c}\right)^{1/p} = \frac{h_d}{h_c} \;\rightarrow\; \left(\frac{\epsilon_d}{\epsilon_c}\right)^{1/p} h_c = h_d \tag{30}$$

If we are using $|\text{XB1} - \text{XB2}|$ as a proxy for the local truncation error, ϵ_c, then we can plug it in to get:

$$\left(\frac{\text{desired error}}{|\text{XB1} - \text{XB2}|}\right)^{1/p} \cdot (\text{current step size}) = \text{desired step size} \tag{31}$$

This is our update rule for computing the step size at the next step. We will add a few performance tweaks to this update rule:

$$\min\left(.9\left(\frac{\text{desired error}}{|\text{XB1} - \text{XB2}|}\right)^{1/p}, \quad \alpha\right) \cdot (\text{current step size}) = \text{desired step size} \tag{32}$$

Here, the factor of .9 is a safeguard to help ensure that the truncation error at the next time step will be smaller than the desired error, ϵ_d. The minimum operation is used to make sure that the step size doesn't change too dramatically (try testing values of α in the range of $[1.5, 10]$). Note that, if the current truncation error, $|\text{XB1} - \text{XB2}|$ was larger than the desired value, ϵ_d, we will need to redo the step. Otherwise, we can just proceed. In both cases, the update rule will tell us the next step size to try.

Implement this update rule in MATLAB as part of a function that executes a single integration step:

```
%This function computes the value of X at the next time step
%for any arbitrary embedded RK method
%also computes the next step size to use, and whether or not to
%accept/reject the projected value of X(t+h)
%(for variable time step methods)
%INPUTS:
%rate_func_in: the function used to compute dXdt. rate_func_in will
%              have the form: dXdt = rate_func_in(t,X)  (t is before X)
%t: the value of time at the current step
%XA: the value of X(t)
%h: the time increment for a single step i.e. delta_t = t_{n+1} - t_{n}
%BT_struct: a struct that contains the Butcher tableau
%   BT_struct.A: matrix of a_{ij} values
%   BT_struct.B: vector of b_i values
%   BT_struct.C: vector of c_i values
%p: how error scales with step size (error = k*h^p)
%error_desired: the desired local truncation error at each step
%OUTPUTS:
%XB: the approximate value for X(t+h)
%num_evals: A count of the number of times that you called
%           rate_func_in when computing the next step
%h_next: the time-step size at the next iteration
%redo: False if the estimated error was less than error_desired
%       True if the estimated error was larger than error_desired
function [XB, num_evals, h_next, redo] = explicit_RK_variable_step...
                        (rate_func_in,t,XA,h,BT_struct,p,error_desired)
    %your code here
end
```

Finally, write a function that runs the variable step integration:

```
%Runs numerical integration arbitrary RK method using variable time steps
%INPUTS:
%rate_func_in: the function used to compute dXdt. rate_func_in will
%              have the form: dXdt = rate_func_in(t,X)  (t is before X)
%tspan: a two element vector [t_start,t_end] that denotes the integration endpoints
%X0: the vector describing the initial conditions, X(t_start)
%h_ref: the desired value of the average step size (not the actual value)
%BT_struct: a struct that contains the Butcher tableau
%   BT_struct.A: matrix of a_{ij} values
%   BT_struct.B: vector of b_i values
%   BT_struct.C: vector of c_i values
%p: how error scales with step size (error = k*h^p)
%error_desired: the desired local truncation error at each step
%OUTPUTS:
%t_list: the vector of times, [t_start;t_1;t_2;...;.t_end] that X is approximated at
%X_list: the vector of X, [X0';X1';X2';...;(X_end)'] at each time step
%h_avg: the average step size
%num_evals: total number of calls made to rate_func_in during the integration
function [t_list,X_list,h_avg, num_evals] = explicit_RK_variable_step_integration ...
                                    (rate_func_in,tspan,X0,h_ref,BT_struct,p,error_desired)
    %your code here
end
```

We will perform more rigorous testing later. However, for the moment, make sure that your integrator works by testing it on the planetary system (and comparing your results with the true solution).

Adaptive Step Size Experiments

(Friday, October 25th) activity. Please complete before class on Tuesday, October 29th.
Let's now perform a few experiments to evaluate the performance of our adaptive time step method.

- Modify your adaptive time step integrator so that it also returns the fraction of failed steps:

$$\text{step failure rate} = \frac{\#\text{failed steps}}{\#\text{attempted steps}} \tag{33}$$

- We will use the planetary motion ODE as the test differential equation. For these experiments, choose an initial position/velocity that generates an eccentric elliptical orbit (less circular and more of a narrow ellipse).
- Run your adaptive step size integrator multiple times to simulate an orbiting planet, across a range of desired error values. Make sure to record the global truncation error, the average step size, the number of function evaluations, and the step failure rate.
- Run the fixed step integrator using the same Runge-Kutta method that you used for your adaptive step size method across a range of desired step sizes. For your fixed step integrator, remember to use the same row of B in the Butcher tableau that you use in the adaptive step integrator to compute X_{n+1}. Make sure to record the global truncation error, the average step size, and the number of function evaluations.
- On the same axes, plot the global truncation error as a function of the **average step size** for both the fixed and adaptive step size integrators. This should be a loglog plot. Which performed better?
- On the same axes, plot the global truncation error as a function of the **number of function evaluations** for both the fixed and adaptive step size integrators. This should be a loglog plot. Which performed better?
- Plot the failure rate as a function of the average step size, using a log scale for the horizontal axis. How bad is the failure rate? Do the failure rate and average step size seem correlated? How effective is the update rule at ensuring that the local truncation error at the next step is below the desired threshold?
- For a single desired error threshold, plot the position and velocity as a function of time (position vs. time and velocity vs. time should be on separate axes). Do this with both lines and dots:

```
%plots x vs. y using a red line & black dots
plot(x,y,'ro-','markerfacecolor','k','markeredgecolor','k','markersize',2)
```

  Try to adjust the error threshold so that the solution is reasonably accurate, but you can still see separate time points. Where do steps tend to cluster together?

- For a single desired error threshold, generate a scatter plot of the step size as a function of the distance between the planet and the sun, $r = \sqrt{x^2 + y^2}$. Note: you extract the list of step sizes by just taking the difference between adjacent times:

```
h_list = diff(t_list);
```

  You may want to try semilogX, semilogY, or loglog to see which visualizes the data best. How does the step size vary as a function of the distance? Why do you think that this is the case?