

Better Algorithms for Numerical Integration

Let us consider a general differential equation involving some function $y(t)$:

$$\text{DE: } \frac{dy}{dt} = f(y) \quad (1)$$

along with the initial conditions IC: $y(0) = y_0$.

Here the *rate_function* $f(y)$ is given to us as a function or a graph or a table of values, and the initial value y_0 is given to us as a number. What we want to do is calculate the variation of y with time t , i.e. obtain $y(t)$ for subsequent times. If you have been following the forward Euler method that we have used so far, what we have been doing is to estimate the value of y at the next instant by adding a correction to the current value of y . In the forward Euler method, the correction term is obtained rather simplistically. We assume that the new value of y after a time t can be obtained by simply multiplying the rate at which y is changing *now* by the time interval t . This will be exactly correct only if over the time interval t , the rate at which y is changing stays constant. If not, then clearly we will be either over estimating or underestimating the new value of y (see fig. 1). This is why we need to keep t very small. But this can become computationally tiresome. Also, it did not work well for the spring-block oscillator no matter how small a time step we chose. Can we do better? Well to find the answer to this, first let us ask why we chose the *current* rate of change of y (i.e. the current slope of the curve $y(t)$)? For one thing, that is all we really know. But can we use the current rate to get a preliminary estimate for the next y , and then figure out a better value for the rate at which y is changing? We can try!

Forward Euler method: Suppose at step i we have a *current_value* y_i . (I will use y_i to mean $y(t_i)$ etc as a convenient shorthand notation.) We feed the *current_value* to the *rate_function* to get the *current_rate* $k_i = f(y_i)$. And then we estimate the *next_value* y_{i+1} through:

$$\begin{aligned} k_i &= f(y_i) && \dots \text{current_rate using current_value in rate_function} \\ y_{i+1} &= y_i + dt * k_i && \dots \text{next_value=current_value+timstep*current_rate} \\ &&& \{ \text{Always read equations using meaningful words.} \} \end{aligned}$$

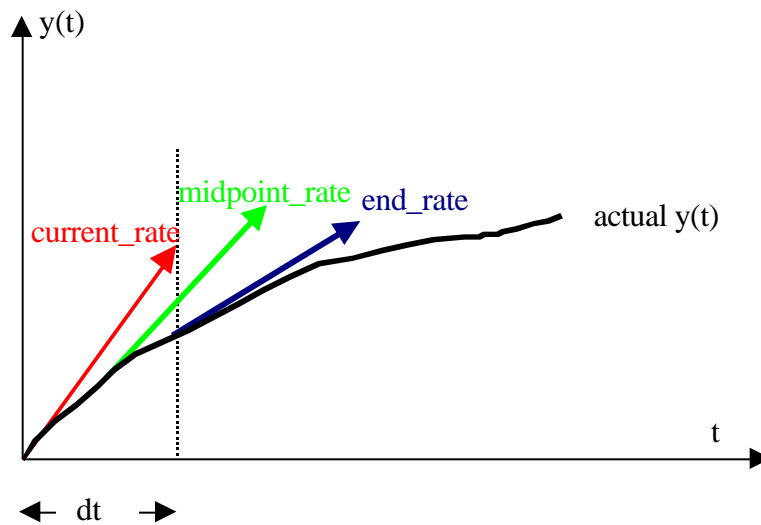


Figure 1: Getting from current_value to next_value

Average Rate method: Now what if we use the current_rate k_1 to get a *preliminary* estimate for the next_value y_{i+1} as in the forward Euler method, but then use this preliminary next_value to get an estimate for the rate of change of y at the *other* end; ie obtain the value of the end_rate $f(y_{i+1}) = f(y_i + dt * k_1)$. We can then make a new and better estimate for y_{i+1} by using the *average* rate at which y changes over this interval dt . That is:

$$\begin{array}{ll}
 k_1 = f(y_i) & \dots \text{current_rate} \\
 y_{i+1}^{(approx)} = y_i + dt * k_1 & \dots \text{first approximation for next_value} \\
 k_2 = f(y_{i+1}^{(approx)}) & \dots \text{estimated end_rate using approximate next_value} \\
 y_{i+1} = y_i + dt * \frac{k_1 + k_2}{2} & \dots \text{better next_value using average_rate.}
 \end{array}$$

Midpoint method: Now, we can get a bit more creative. What we did in the Average Rate method was to average the rate at the beginning of the interval and at the (approximated) end of the interval. We could instead try to find the rate at the approximate *middle* of the interval. All we really know is y_i , but we can approximate the value at the middle of the interval as $y_i + dt * k_1 / 2$. That is:

$$\begin{aligned}
 k_1 &= f(y_i) && \dots \text{current_rate} \\
 k_2 &= f\left(y_i + \frac{dt}{2} * k_1\right) && \dots \text{estimated midpoint_rate} \\
 y_{i+1} &= y_i + dt * k_2 && \dots \text{next_value using midpoint_rate.}
 \end{aligned}$$

Runge-Kutta method: A lot of work has been done on the numerical integration of differential equations. One of the best compromises between accuracy and simplicity turns out to be the so-called 4th order Runge-Kutta method. It works like this:

$$\begin{aligned}
 k_1 &= f(y_i) && \dots \text{current_rate} \\
 k_2 &= f\left(y_i + \frac{dt}{2} * k_1\right) && \dots \text{first approximate midpoint_rate using current_rate} \\
 k_3 &= f\left(y_i + \frac{dt}{2} * k_2\right) && \dots \text{better midpoint_rate using approx midpoint_rate} \\
 k_4 &= f\left(y_i + dt * k_3\right) && \dots \text{end_point rate} \\
 y_{i+1} &= y_i + dt * \left(\frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}\right) && \dots \text{next_value using weighted average of rates}
 \end{aligned}$$

It is beyond the scope of this course to explain why this is such a good method, and why the weighting factors (1/6, 1/3, 1/3, 1/6) work so well. You'll learn about these things in EA4 and other courses. All I want you to do is to appreciate what is going into these approximate methods. You will not have to create new methods yourself (that task is pretty much all done now.) However, we will make use of the RK algorithm extensively in the rest of the course.

The Spring-Block Oscillator Revisited: Consider the spring-block oscillator:

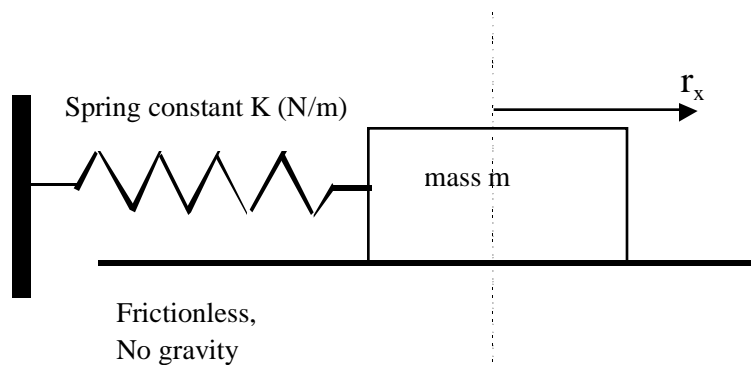


Figure 2: Undamped spring-block oscillator

Let the position of the block when the spring is unstretched ('relaxed' position of the spring) be the origin of our coordinate system (see Fig.2). If the block is now moved by an amount r_x as shown, the spring stretches (or compresses) and therefore starts to pull (or push) on the block with a force $F_x = -Kr_x$ (we are considering a linear elastic spring which you have seen in EA2). From the free-body diagram of the block, we find that the equations of motion of block are given by:

$$\begin{aligned} \frac{dv_x}{dt} &= \frac{1}{m} F_x = -\frac{K}{m} r_x \\ \frac{dr_x}{dt} &= v_x \end{aligned} \quad (\dagger)$$

Let us say that at time $t=0$, the block is gently moved to r_{x0} and let go with zero velocity at this time, ie $r_x(t=0) = r_{x0}$; $v_x(t=0) = v_{x0} = 0$. Given these initial conditions, we want to find the subsequent motion of the block.

It is customary to re-cast Equations (\dagger) in matrix form:

Define: a **state vector** $\underline{X} = \begin{matrix} r_x \\ v_x \end{matrix}$ which contains a list of things we want to track. Then

Eqns (\dagger) can be written in matrix form as:

$$\begin{matrix} \dot{r}_x \\ \dot{v}_x \end{matrix} = \begin{matrix} 0 & 1 \\ -\frac{K}{m} & 0 \end{matrix} \begin{matrix} r_x \\ v_x \end{matrix} \quad \dot{\underline{X}} = \underline{A}\underline{X}$$

where superposed dot is shorthand for time derivative; and \underline{A} is called the system matrix.

We have previously seen that the forward Euler algorithm does not do a good job of calculating the spring-block oscillator motion (see chapter 3). Now let us try to obtain solutions using our new improved methods.

It is actually a simple matter to code any and all of the above methods in MATLAB. I am enclosing an m-file *spring_block_comp.m* and a companion file *rate_fn.m* which calculate the motion of a spring-block oscillator using the forward Euler, the midpoint, the average rate and the Runge-Kutta methods along with the analytical solution. Measures of error (percent error) for each are also obtained. Note that since we have to repeatedly get the rate (current_rate, midpoint_rate, end_rate) from the rate function, we actually code that in a separate m-file and use function calls in the main program.

```

%Spring-block system with no damping or friction
%%-----
%% Matrix implementation of the spring-mass system
%% Compares Forward Euler, Midpoint Rate, Average Rate, and Runge-Kutta algorithms
%% also includes analytical solution
%%-----

clear all;
close all;
global A; %share this with function rate_fn which computes Xdot
%%
%% System parameters
m = 40; %mass of block
K = 10; %spring constant of spring
w=sqrt(K/m); %calculate angular frequency of oscillation
period = 2*pi/w; %calculate period of oscillation
%
% Initial conditions
i = 1; %time index
rx0 = 0.2; %initial position
vx0 = 0; %initial velocity

%% store calculated positions in **_x(1,i) and calculated velocities in **_x(2,i)
%% for different times pointed to by index 'i'
rk_x(1,i)= rx0; rk_x(2,i) = vx0; %stores runge kutta solution
fe_x(1,i)= rx0; fe_x(2,i) = vx0; %stores forward euler solution
avg_x(1,i) = rx0; avg_x(2,i) = vx0; %stores average rate solution
mp_x(1,i) = rx0; mp_x(2,i) = vx0; %stores midpoint rate solution
an_x(1,i) = rx0; an_x(2,i) = vx0; %stores analytical solution values for
comparison

%error calculations
amplitude_E=sqrt((vx0/w)^2+rx0^2); %amplitude of oscillation
error_rk(i) = (rk_x(1,i)-an_x(1,i))/amplitude_E; % stores normalized error using Runge Kutta
error_avg(i) = (avg_x(1,i)-an_x(1,i))/amplitude_E; % stores normalized error using Average rate
error_mp(i) = (mp_x(1,i)-an_x(1,i))/amplitude_E; % stores normalized error using Midpoint rate
error_fe(i) = (fe_x(1,i)-an_x(1,i))/amplitude_E; % stores normalized error using Forward Euler

%
% time parameters
start_time = 0;
dt = period/100; %time step chosen as a fraction of one period
final_time = 10*period; %end calculation after ten periods
time(i) = start_time; %let us keep a vector of times for plotting

%
A = [0, 1; -K/m, 0]; % for use in equation Xdot = A X

%
% Integrate equations of motion using the various methods

for t = start_time+dt:dt:final_time

%% Analytical solution
an_x(1,i+1) = (vx0/w)*sin(w*t)+rx0*cos(w*t); %analytical position
an_x(2,i+1) = vx0*sin(w*t)-w*rx0*cos(w*t); %analytical velocity

```

```

%%Forward Euler method
k1 = rate_fn(fe_x(:,i)); %current rate
fe_x(:,i+1) = fe_x(:,i) + dt*k1; %next value using current rate

%%Mid-point Rate method
k1 = rate_fn(mp_x(:,i)); %current rate
k2 = rate_fn(mp_x(:,i)+dt*k1/2); %estimate for mid-point rate
mp_x(:,i+1) = mp_x(:,i) + dt*k2; %next value using midpoint rate rate

%%Average Rate method
k1 = rate_fn(avg_x(:,i)); %current rate
k2 = rate_fn(avg_x(:,i)+dt*k1); %estimate for next rate
avg_x(:,i+1) = avg_x(:,i) + dt*(k1+k2)/2; %next value using average rate

%%Runge Kutta method
k1 = rate_fn(rk_x(:,i)); % current rate
k2 = rate_fn(rk_x(:,i) + dt*k1/2); % estimated mid-point rate
k3 = rate_fn(rk_x(:,i) + dt*k2/2); % better mid-point rate
k4 = rate_fn(rk_x(:,i) + dt*k3); % excellent end-point rate
rk_x(:,i+1) = rk_x(:,i) + dt*(k1/6 + k2/3 + k3/3 + k4/6);% next value using weighted average of rates

%%Error in position using various algorithms
error_rk(i+1) = (rk_x(1,i+1)-an_x(1,i+1))/amplitude_E; %error using Runge Kutta
error_avg(i+1) = (avg_x(1,i+1)-an_x(1,i+1))/amplitude_E; %error using Average Rate
error_mp(i+1) = (mp_x(1,i+1)-an_x(1,i+1))/amplitude_E; %error using Midpoint Rate
error_fe(i+1) = (fe_x(1,i+1)-an_x(1,i+1))/amplitude_E; %error using Forward Euler

%%increment time and do next step
time(i+1) = time(i) + dt;
i = i+1;
end

%Plot the position and error
subplot(4,1,1), plot(time(:),rk_x(1,:),time(:),an_x(1,:)), title('Runge Kutta'), grid
subplot(4,1,2), plot(time(:),avg_x(1,:),time(:),an_x(1,:)), title('Average Rate'), grid
subplot(4,1,3), plot(time(:),mp_x(1,:),time(:),an_x(1,:)), title('Midpoint Rate'), grid
subplot(4,1,4), plot(time(:),fe_x(1,:),time(:),an_x(1,:)), title('Forward Euler'),xlabel('Time'), grid
figure
subplot(4,1,1),plot(time(:),error_rk(:)), title('Runge-Kutta Error'), grid
subplot(4,1,2),plot(time(:),error_avg(:)), title('Average Rate Error'), grid
subplot(4,1,3),plot(time(:),error_mp(:)), title('Midpoint Rate Error'), grid
subplot(4,1,4),plot(time(:),error_fe(:)), title('Forward Euler Error'), xlabel ('Time'), grid

```

The results are shown in two sets of plots in figs. 3,4. The first set shows the calculated position versus time for each of the methods. The analytical solution (see chapter 3) is also plotted in each graph. As you can see, the Forward Euler method fails miserably, but the other three seem to perform quite well. It is hard to see any difference between these numerical solutions and the analytical one from the position versus time plots. To get a better idea of accuracy, we also plot the normalized error for each method in fig. 3. (Look at the vertical scale to see the fractional error for each method.) You can see that the RK method performs by far the best with error exceedingly small compared to the other methods.

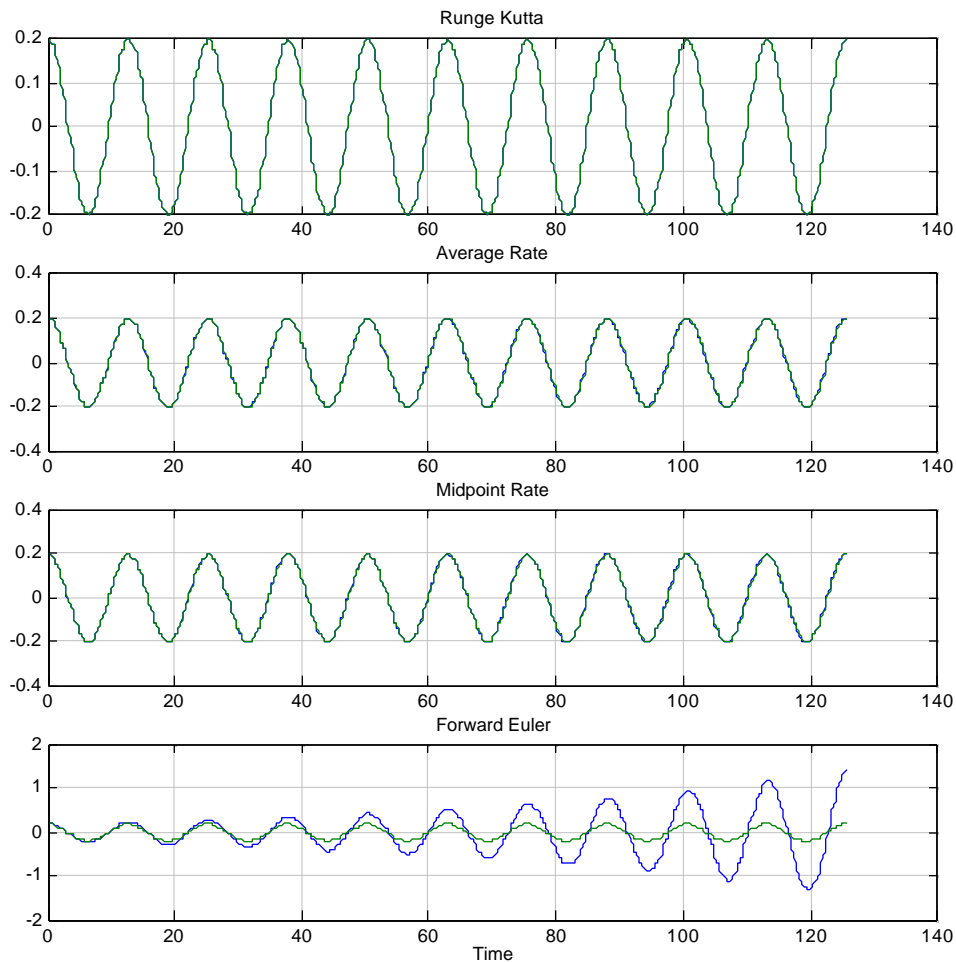


Figure 3: Spring-block oscillator solved numerically using the various methods

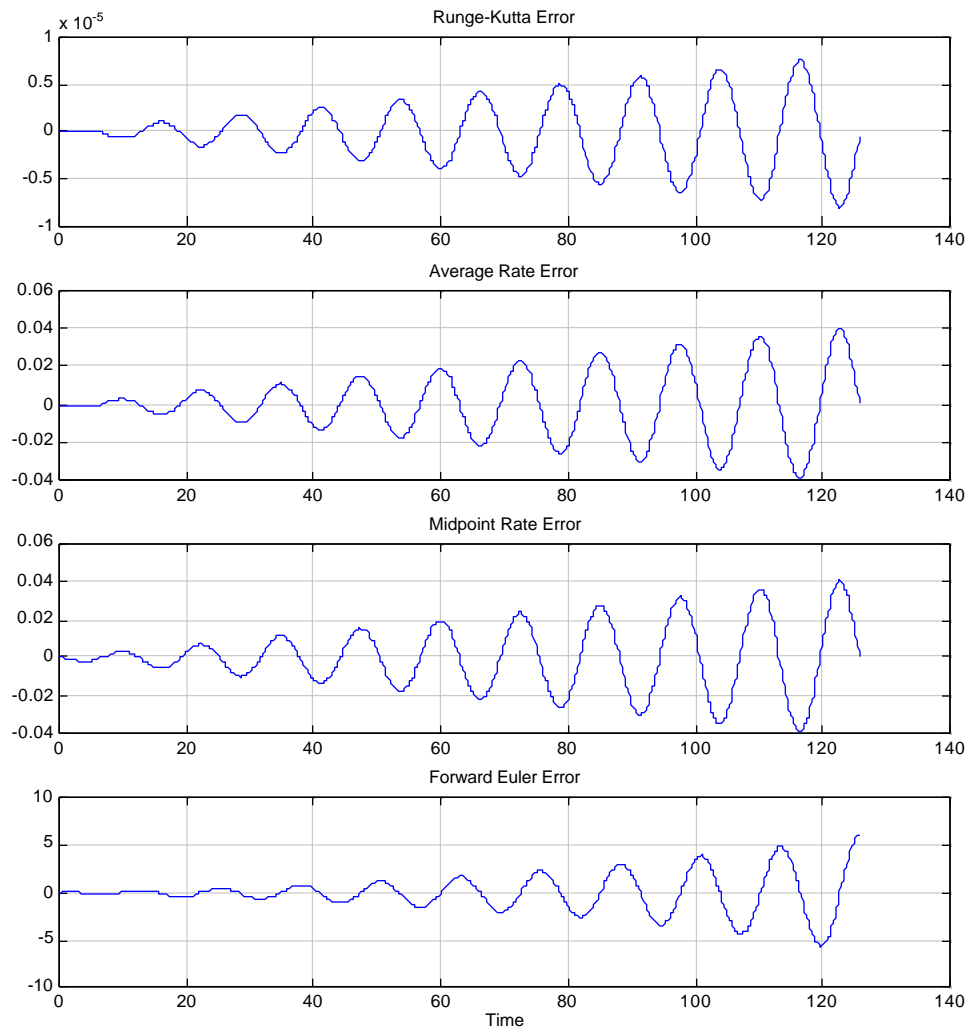


Figure 4: Spring-block oscillator solved numerically using the various methods – Error in solutions