Using Numeric Integration to Solve Systems of ODEs

Nicholas Livingstone

MATH-375

Spring 2020

Given a system of ODEs, how might we go about solving them? A common approach is through linear algebra and diagnalization. However, sometimes it may not be possible to solve the system 'by hand'. One alternative to this is through numerical integration. By integrating a ODE system, we can estimate values of it's closed formula with respect to (*t*). In this project, we'll look at a matlab implementation of the midpoint approximation of integration and analyze its accuracy in two different ODE systems.

Consider the following ODE IVP

$$y_1' = -y_2 \quad y_2' = y_1$$

With initial conditions $y_1(t_0) = y_1(0) = 1 \quad y_2(t_0) = y_2(0) = 0$

In order to determine the accuracy of the program we must first determine the exact solution. We can use Linear Algebra as described earlier. Consider the system

$$Ay = y'$$

Where

$$A = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \quad y = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \quad y' = \begin{bmatrix} y_1' \\ y_2' \end{bmatrix}$$

First we must find the eigenvalues and eigen vectors

$$\det(A - \lambda I) = 0$$

$$\det \begin{pmatrix} -\lambda & -1 \\ 1 & -\lambda \end{pmatrix} = 0$$

$$\lambda^2 + 1 = 0$$

$$\lambda_{1,2} = \pm i$$

$$(A - \lambda I)\begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = 0$$

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} - \begin{bmatrix} i & 0 \\ 0 & -i \end{bmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = 0$$

$$\begin{bmatrix} -i & -1 \\ 1 & i \end{bmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = 0$$

$$-iv_1 - v_2 = 0 \quad v_i + iv_2 = 0$$

$$-iv_1 = v_2 \quad v_i = -iv_2$$

let $v_1 = 1$ and v_2= 1 produces:

$$V_1 = \begin{bmatrix} 1 \\ -i \end{bmatrix} \quad V_2 = \begin{bmatrix} -i \\ 1 \end{bmatrix}$$

Which leads to the equation:

$$y(t) = c_1 e^{it} \begin{bmatrix} 1 \\ -i \end{bmatrix} + c_2 e^{-it} \begin{bmatrix} -i \\ 1 \end{bmatrix}$$

Using Euler's Formula:

$$y(t) = c_1(\cos(t) + i\sin(t)) \begin{bmatrix} -i \\ 1 \end{bmatrix} + c_2(\cos(-t) + i\sin(-t)) \begin{bmatrix} -i \\ 1 \end{bmatrix}$$

$$y(t) = c_1 \begin{bmatrix} \cos(t) + i\sin(t) \\ i\cos(t) + \sin(t) \end{bmatrix} + c_2 \begin{bmatrix} -i\cos(-t) + sin(-t) \\ \cos(-t) + i\sin(-t) \end{bmatrix}$$

Removing complex components results in:

$$y(t) = c_1 \begin{bmatrix} \cos(t) \\ \sin(t) \end{bmatrix} + c_2 \begin{bmatrix} \sin(-t) \\ \cos(t) \end{bmatrix}$$

Now we must determine c_1 and c_2 using the inital condition $t_0 = 0$

$$y(0) = c_1 \begin{bmatrix} \cos(0) \\ \sin(0) \end{bmatrix} + c_2 \begin{bmatrix} \sin(0) \\ \cos(0) \end{bmatrix}$$

$$c_1 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + c_2 \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$c_1 = 1 \quad c_2 = 0$$

Therefore we get a general solution of

$$y(t) = \begin{bmatrix} \cos(t) \\ \sin(t) \end{bmatrix}$$

Or alternatively:

$$y_1 = \cos(t) \quad y_2 = sin(t)$$

Now that we have an exact solution, we can begin to develop a method of approximation. In this approach, we'll be using the fixed mid-point rule to estimate the integral. It approximates then next $y(t+1)$ using the differential equations themselves, where the difference between t and $t+1$ is a fixed value h. The formulas used are:

$$y_{n+1} = y_n + h\phi(t_n, y_n)$$

$$\phi(t_n, y_n) = k_2(t_n, y_n) = f(t_n + \frac{1}{2}h, y_n + \frac{1}{2}hk_1)$$

$$k_1(t_n, y_n) = f(t_n, y_n)$$

The first equation will be doing the actual approximation and adjusting the previous y value by some scale h, provided by $\phi(t_n, y_n)$. The equation $f(t_n, y_n)$ refers to the differential equations of our inital problem. In the ϕ/k_2 functions, the inputs are adjusted by$h$. The larger the h, the farther away this formula will approximate $y(t+1)$.

Let's start with the inital equations and conditions.

```
%function handle calculates y'1 and y'2 %
f = @(t, y) [-y(2); y(1)];
%inital conditions%
y0 = [1 ; 0];
t0 = 0;

%Formulas needed for mid-point approx %
k2 = @(t, y, h) f(t+0.5*h, y+.5*h*f(t,y)); % note here that k1 was replaced with f %
```

Next let's make some parameters for our program to make tweaks to how it runs.

```
n = 800;                    % number of steps, will change the value of h
tmax = 20 * pi;             % max time
                % size of our steps based on bounds and number of iterations
d = length(y0);              % for allocating memory below
output = zeros(n+1, 2+d);   % this preallocation is important as it's inneficient to be
                            % constantly updating memory in our loop
```

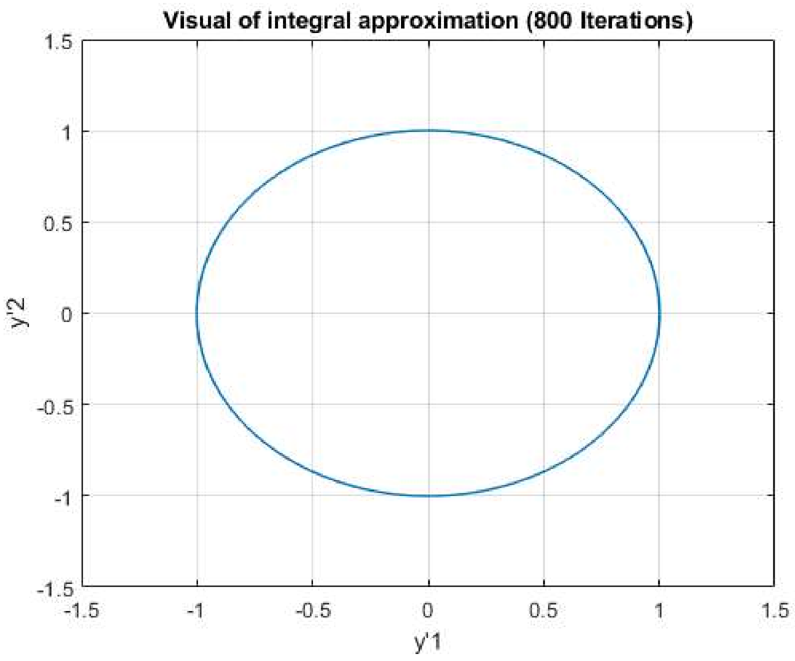
The last variable, *output*, above will store times, h value, and y values. Now we can write the algorithm to approximate the integral.

```
h = (tmax-t0)/n;
i = 2;                          % For indexing %
output(1, :) = [t0, h, y0'];     % Store inital values %
y = y0;
t = t0;

while(t < tmax)
    t = t + h;                      % Take step to new t %
    y = y + h * k2(t, y, h);        % Complete approximation %
    output(i, :) = [t, h, y'];      % Save Step
    i = i + 1;                      % Increment index
end
```

Now we can plot to get a visual of how accurate our approximation is. Given the exact solution we found, we should expect a unit circle.

```
plot(output(:, 3), output(:, 4));
grid on;
xlabel("y'1");
ylabel("y'2");
title('Visual of integral approximation (800 Iterations)')
```



From the graph, it appears that 800 iterations produces a fairly accurate approximation. Lets take a peek at the data to check.

```
VarNames = {'t', 'h', 'yprime1', 'cos t', 'yprime2', 'sin t'};
cos_t = cos(output(:,1));
sin_t = sin(output(:,1));
T = table(output(:,1),output(:,2),output(:,3),cos_t,output(:,4), sin_t,'VariableNames',VarNames);
head(T)
```

```
ans = 8�6 table
```

	t	h	yprime1	cos t	yprime2	sin t
1	0	0.07853981...	1.00000000...	1.00000000...	0	0
2	0.07853981...	0.07853981...	0.99691574...	0.99691733...	0.07853981...	0.07845909...
3	0.15707963...	0.07853981...	0.98767250...	0.98768834...	0.15659515...	0.15643446...
4	0.23561944...	0.07853981...	0.97232732...	0.97236992...	0.23368379...	0.23344536...
5	0.31415926...	0.07853981...	0.95097493...	0.95105651...	0.30932946...	0.30901699...
6	0.39269908...	0.07853981...	0.92374721...	0.92387953...	0.38306481...	0.38268343...
7	0.47123889...	0.07853981...	0.89081230...	0.89100652...	0.45443428...	0.45399049...
8	0.54977871...	0.07853981...	0.85237362...	0.85264016...	0.52299692...	0.52249856...

```
tail(T)
```

```
ans = 8�6 table
```

	t	h	yprime1	cos t	yprime2	sin t
1	62.3606141...	0.07853981...	0.92168935...	0.89100652...	-0.39758057...	-0.453990499740482
2	62.4391539...	0.07853981...	0.95007253...	0.92387953...	-0.32396501...	-0.382683432366062
3	62.5176938...	0.07853981...	0.97258642...	0.95105651...	-0.24834730...	-0.309016994375952
4	62.5962336...	0.07853981...	0.98909187...	0.97236992...	-0.17119458...	-0.233445363856935
5	62.6747734...	0.07853981...	0.99948685...	0.98768834...	-0.09298348...	-0.156434465041280
6	62.7533132...	0.07853981...	1.00370709...	0.99691733...	-0.01419718...	-0.078459095728907
7	62.8318530...	0.07853981...	1.00172645...	1.00000000...	0.06467757...	-0.000000000001068
8	62.9103928...	0.07853981...	0.99355711...	0.99691733...	0.14315350...	0.078459095726777

It appears that we lose quite a bit of accuracy as *t* increases, especially for y_2'. Additionally our approximation seems to start off by underestimating the true data, and then overestimating also while t gets large.

But we can further analyze the accuracy by comparing the infinity norms of the exact solution to our expected.

```
forward_error_y1 = norm(output(:,3), "inf") - norm(cos_t, "inf")
```

```
forward_error_y1 =
   0.003707095653615
```

```
forward_error_y2 = norm(output(:,4), "inf") - norm(sin_t, "inf")
```

```
forward_error_y2 =
   0.003587435646275
```

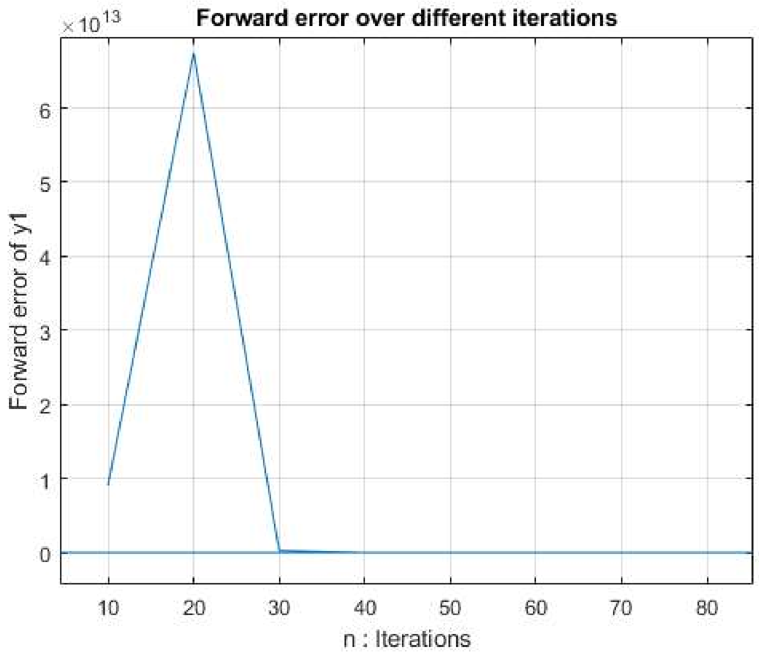
From this, we can confirm that our approximation is second-order accurate. Presumably, if we chose a larger h (based on the iterations and bounds), we likely would increase our accuracy, and vice versa for a smaller h. We can look at the condition number the coefficient matrix to determine by how much this might change based on the h.

```
cond([0, -1; 1, 0])
```

```
ans =
     1
```

Since the condition number is very low, we can assume that it will take large changes in h for us to loose accuracy.

```
plot(errors(:, 3), errors(:, 1));
xlabel("n : Iterations");
ylabel("Forward error of y1");
title("Forward error over different iterations");
xlim([4.4 85.4])
ylim([-4300541189964 69605982209543])
grid on;
```



By running the program over iterations from 10-1000 in incrememnts of 10, we can clearly see by the graph that after 30 iterations changes in accuracy will become significantly smaller. For example the difference in error between 15 and 20 iterations will be significant, whereas 55 to 65 will not see the same amount of fluctuation.

Regardless, this method can be improved in the event the problem we are presented with is not as forgiving as this one.

Consider the following ODE IVP

$$y' = -\frac{1}{2}y^3, \quad y(t_0) = y(1) = 1, \text{ with exact solution: } y(t) = t^{-1/2}$$

One way that our approximation can be improved when integrating this equation is by dyanmically changing the step size h as the program runs based on the estimated error. Plainly speaking, if the error of our current iteration is expected to be small, increase the step size. If it's expected to be large, maintain a smaller step size. The formulas we will use are going to be releatively the same, but with a replacement of the fixed step-size h with h_n

$$y_{n+1} = y_n + h_n\phi(t_n, y_n; h_n)$$

$$\phi(t_n, y_n; h_n) = k_2(t_n, y_n; h_n) = f(t_n + \frac{1}{2}h_n, y_n + \frac{1}{2}h_n k_1)$$

$$k_1(t_n, y_n) = f(t_n, y_n)$$

To determine if a change in step size is needed, we will define delta to be:

$$\delta = \max_{1 \le i \le d} \delta_i \quad \delta_i = h_n\|k_{2,i} - k_{1,i}\| / \|3y_i\|$$

Delta will be compared to a defined tolerance, when $\delta_i \le tol$, $h_{n+1} = h_n \min(1.5, \sqrt{tol/1.2\delta})$. If the error is too big we shall restart the iteration with a smaller h_n, defined as $h_n \min(0.1, \sqrt{tol/(1.2\delta)})$. Additionally to prevent issues with underflow, $\overline{y_i} = \max(\|y_i\|, 0.001)$.

Similarly as before, let's first define the problem and it's initial conditions.

```
f = @(t, y) -0.5*y^3; % Differential equation
k2 = @(t, y, h) f(t+0.5*h, y+.5*h*f(t,y)); % Define K2

% Initial Conditions
t = 1;
y = 1;
```

Then we can begin writing the program by establishing parameters.

```
tol = 2*10^(-4);                % Tolerance for adjusting step-size
MAXTIME = 50;                   % Max time
h = 0.025;                      % Step size
nsteps = 0;                     % Step count
d = length(y);                  % Number of ys in system
Nguess = 112;                 % Guess Number of steps to meet final time
output=zeros(Nguess+1, 2+d);    % Storage for t, h, and y
q = 1; output(q,:) = [t h y];   % Store inital values

while(t < MAXTIME)
    nsteps0 = nsteps;
    % Prevent y from getting too small
    if abs(y) < 0.001
        y = 0.001;
    end

    s = h * abs(k2(t, y, h) - f(t, y))/abs(3*y);   %calculate inital delta

    % Compare delta to tolerance
    while s > tol   % Recalculate if too large
        h = min(0.1, sqrt(tol/(1.2*s)))
        s = h * abs(K2(t, y) - f(t, y))/abs(3*y);
    end

    % Calculate next step size and step
    h = h * min(1.5, sqrt(tol/(1.2*s)));
    t = t + h;
    %complete approximation
    y = y + h * k2(t, y, h);

    q = q+1;
    if(q > Nguess + 1)
        err = 'Too many steps';
        pause
    end
    output(q, :) = [t h y]; %save data

    % 'Land' on max time
    if(MAXTIME - t < h)
        h = MAXTIME-t;
    end
end
```

Let's calculate the exact solution and compare

```
for i = 1:110
    output(i,4) = output(i, 1)^(-1/2);
end


VarNames = {'t', 'h', 'yprime', 'exact', 'error'};
T = table(output(:,1),output(:,2),output(:,3),output(:,4), abs(output(:,4)-output(:,3)),'VariableNames',VarNames);
head(T)
```

```
ans = 8�5 table
```

	t	h	yprime	exact	error
1	1.00000000...	0.02500000...	1.00000000...	1.00000000...	0
2	1.03662924...	0.03662924...	0.98218392...	0.98217363...	1.028811244352390e-05
3	1.07464860...	0.03801936...	0.96466320...	0.96464331...	1.989058614093597e-05
4	1.11406178...	0.03941317...	0.94745494...	0.94742613...	2.880722944620384e-05
5	1.15491965...	0.04085787...	0.93055365...	0.93051657...	3.707718737722132e-05
6	1.19727517...	0.04235552...	0.91395385...	0.91390912...	4.473764997903196e-05
7	1.24118324...	0.04390806...	0.89765017...	0.89759835...	5.182379410462179e-05
8	1.28670076...	0.04551752...	0.88163733...	0.88157896...	5.836888978538735e-05

```
tail(T)
```

```
ans = 8�5 table
```

	t	h	yprime	exact	error
1	43.8362569...	1.55002854...	0.15108085...	0.15103697...	4.388548500960932e-05
2	45.4431020...	1.60684502...	0.14838578...	0.14834264...	4.313821642501492e-05
3	47.1088461...	1.66574412...	0.14573878...	0.14569638...	4.240240730632450e-05
4	48.8356482...	1.72680217...	0.14313900...	0.14309732...	4.167795157544729e-05
5	50.5821758...	1.74652755...	0.14064606...	0.14060515...	4.091190149640989e-05
6	0	0	0	0	0
7	0	0	0	0	0
8	0	0	0	0	0

With a tolerance of 2×10^{-4}, the approximation appears to be of third-order accuracy.

```
forward_error = abs(norm(output(:,3), "inf") - norm(output(:,4), "inf"))
```

```
forward_error = 
     0
```

Unfortunately it appears that the forward error doesn't provide much insight, so lets see what happens if we modify the tolerance.

```
t = 1;
y = 1;
tol = 5*10^(-4);                % Tolerance for adjusting step-size
MAXTIME = 50;                   % Max time
h = 0.025;                      % Step size
nsteps = 0;                     % Step count
d = length(y);                  % Number of ys in system
Nguess = 70;                 % Guess Number of steps to meet final time
output=zeros(Nguess+1, 2+d);    % Storage for t, h, and y
q = 1; output(q,:) = [t h y];   % Store inital values

while(t < MAXTIME)
    nsteps0 = nsteps;
    % Prevent y from getting too small
    if abs(y) < 0.001
        y = 0.001;
    end
    s = h * abs(k2(t, y, h) - f(t, y))/abs(3*y);   %calculate inital delta

    % Compare delta to tolerance
    while s > tol   % Recalculate if too large
        h = min(0.1, sqrt(tol/(1.2*s)))
        s = h * abs(K2(t, y) - f(t, y))/abs(3*y);
    end

    % Calculate next step size and step
    h = h * min(1.5, sqrt(tol/(1.2*s)));
    t = t + h;
    %complete approximation
    y = y + h * k2(t, y, h);

    q = q+1;
    if(q > Nguess + 1)
        err = 'Too many steps';
        pause
    end
    output(q, :) = [t h y]; %save data

    % 'Land' on max time
    if(MAXTIME - t < h)
        h = MAXTIME-t;
    end
end

for i = 1:71
    output(i,4) = output(i, 1)^(-1/2);
end


VarNames = {'t', 'h', 'yprime', 'exact', 'error'};
T = table(output(:,1),output(:,2),output(:,3),output(:,4), abs(output(:,4)-output(:,3)),'VariableNames',VarNames);
head(T)
```

```
ans = 8�5 table
```

	t	h	yprime	exact	error
1	1.000000000000000	0.02500000...	1.00000000...	1.00000000...	0
2	1.037500000000000	0.03750000...	0.98177241...	0.98176138...	1.102800423580241e-05
3	1.093750000000000	0.05625000...	0.95622515...	0.95618288...	4.227142296497899e-05
4	1.157300235455537	0.06355023...	0.92963617...	0.92955904...	7.713109807705454e-05
5	1.224567177686818	0.06726694...	0.90377570...	0.90366756...	1.081398086218988e-04
6	1.295738925827323	0.07117174...	0.87863454...	0.87849895...	1.355923513783885e-04
7	1.371041949627652	0.07530302...	0.85419275...	0.85403295...	1.598063860005494e-04
8	1.450716054064648	0.07967410...	0.83043089...	0.83024982...	1.810732143885385e-04

```
tail(T)
```

```
ans = 8�5 table
```

	t	h	yprime	exact	error
1	34.2210483...	1.87752003...	0.17106840...	0.17094379...	1.246073998632524e-04
2	36.2075519...	1.98650359...	0.16630963...	0.16618828...	1.213472852369668e-04
3	38.3093652...	2.10181327...	0.16168324...	0.16156508...	1.18161125311849 7e-04
4	40.5331814...	2.22381627...	0.15718555...	0.15707050...	1.150482210126036e-04
5	42.8860826...	2.35290113...	0.15281298...	0.15270097...	1.120077824764698e-04
6	45.3755615...	2.48947892...	0.14856204...	0.14845300...	1.090389405263847e-04
7	48.0095461...	2.63398458...	0.14442935...	0.14432321...	1.061407570985273e-04
8	50.7917332...	2.78218709...	0.14041809...	0.14031480...	1.032972190182224e-04

Increasing the tolerance has an affect of converging to the max time much quicker after only 71 iterations but appears to have much of the same amount of error as the previous selected tolerance.

```
forward_error = abs(norm(output(:,3), "inf") - norm(output(:,4), "inf"))
forward_error =
     0
```

Similarly as before, it's also producing a zero forward error. Let's try changing the max time to 200 and see how it affects the program.

```
t = 1;
y = 1;
tol = 5*10^(-4);                % Tolerance for adjusting step-size
MAXTIME = 200;                   % Max time
h = 0.025;                      % Step size
nsteps = 0;                     % Step count
d = length(y);                  % Number of ys in system
Nguess = 95;                 % Guess Number of steps to meet final time
output=zeros(Nguess+1, 2+d);    % Storage for t, h, and y
q = 1; output(q,:) = [t h y];   % Store inital values

while(t < MAXTIME)
    nsteps0 = nsteps;
    % Prevent y from getting too small
    if abs(y) < 0.001
        y = 0.001;
    end

    s = h * abs(k2(t, y, h) - f(t, y))/abs(3*y);   %calculate inital delta

    % Compare delta to tolerance
    while s > tol   % Recalculate if too large
        h = min(0.1, sqrt(tol/(1.2*s)))
        s = h * abs(K2(t, y) - f(t, y))/abs(3*y);
    end

    % Calculate next step size and step
    h = h * min(1.5, sqrt(tol/(1.2*s)));
    t = t + h;
    %complete approximation
    y = y + h * k2(t, y, h);

    q = q+1;
    if(q > Nguess + 1)
        err = 'Too many steps';
        pause
    end
    output(q, :) = [t h y]; %save data

    % 'Land' on max time
    if(MAXTIME - t < h)
        h = MAXTIME-t;
    end
end

for i = 1:96
    output(i,4) = output(i, 1)^(-1/2);
end


VarNames = {'t', 'h', 'yprime', 'exact', 'error'};
T = table(output(:,1),output(:,2),output(:,3),output(:,4), abs(output(:,4)-output(:,3)),'VariableNames',VarNames);
head(T)
```

```
ans = 8�5 table
```

	t	h	yprime	exact	error
1	1.00000000...	0.02500000...	1.00000000...	1.00000000...	0
2	1.03750000...	0.03750000...	0.98177241...	0.98176138...	1.102800423580241e-05
3	1.09375000...	0.05625000...	0.95622515...	0.95618288...	4.227142296497899e-05
4	1.15730023...	0.06355023...	0.92963617...	0.92955904...	7.713109807705454e-05
5	1.22456717...	0.06726694...	0.90377570...	0.90366756...	1.081398086218988e-04

	t	h	yprime	exact	error
6	1.29573892…	0.07117174…	0.87863454…	0.87849895…	1.355923513783885e-04
7	1.37104194…	0.07530302…	0.85419275…	0.85403295…	1.598063860005494e-04
8	1.45071605…	0.07967410…	0.83043089…	0.83024982…	1.810732143885385e-04

```
tail(T)
```

```
ans = 8�5 table
```

	t	h	yprime	exact	error
1	1.40257964…	7.69491720…	0.08450066…	0.08443766…	6.299458041679473e-05
2	1.48399544…	8.14158058…	0.08215003…	0.08208876…	6.126705347749006e-05
3	1.57013716…	8.61417124…	0.07986478…	0.07980520…	5.958556595860032e-05
4	1.66127910…	9.11419415…	0.07764311…	0.07758516…	5.794900081140808e-05
5	1.75771151…	9.64324166…	0.07548324…	0.07542688…	5.635625877821460e-05
6	1.85974150…	10.2029985…	0.07338345…	0.07332865…	5.480625897544278e-05
7	1.96769397…	10.7952474…	0.07134208…	0.07128878…	5.329793937300376e-05
8	2.01615301…	4.84590309…	0.07047846…	0.07042684…	5.161388487233409e-05

With the same tolerance, increasing the max time increases the number of iterations needed and also appears to increase the error towards the end of the program, but still with little fluctuation. In comparison to the previous problem, the accuracy has increased a small amount. Both of these methods can be effective, but dynamically changing the step size will produce more accurate results when estimating the solution to a differential equation.