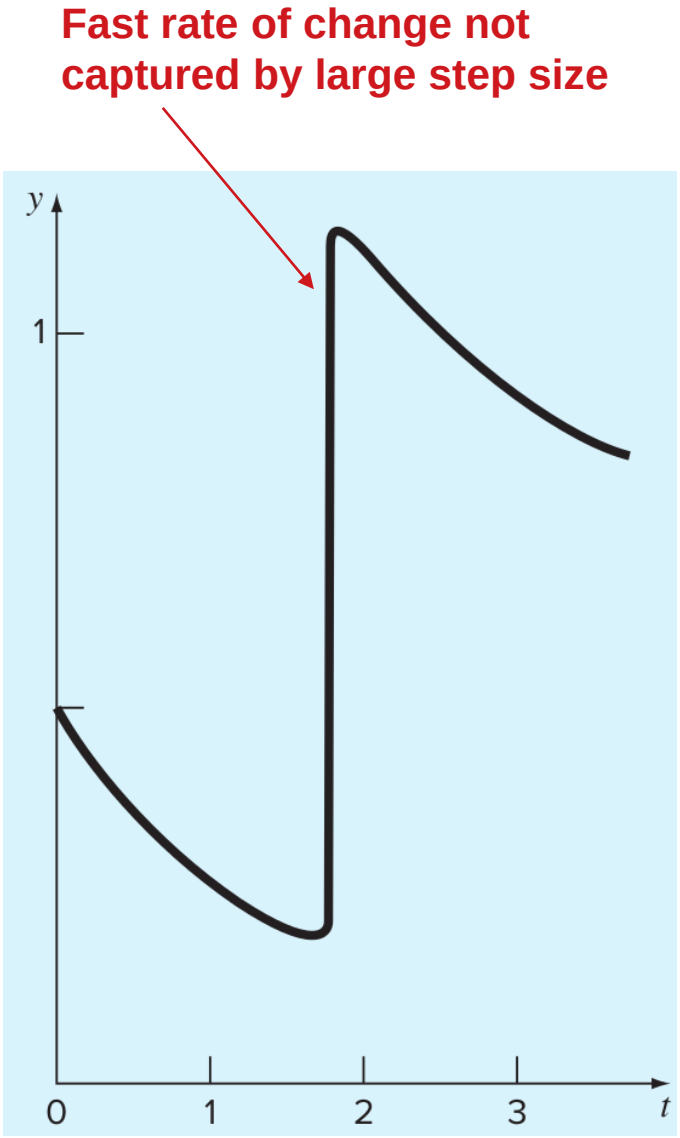
14.1 Adaptive Runge-Kutta

- The derivative of the function shown on the right can be accurately estimated using finite differences with medium step sizes except for close to $x = 2$.

- If we were to take $h = 0.5$, for example, our finite difference equations would indicate that the gradient is always negative and miss the fact that the function rapidly increases at $x = 2$.

- We would have to apply a much smaller step size to capture this quality, however we don't want to waste computation time by employing a small step size over the full domain.



- We could proceed as in previous lectures by estimating the error and halving the step size then employing Richardson extrapolation whenever the estimated error is too high, however other more effective methods have been developed, namely **embedded Runge-Kutta methods**.

- These methods involve combining 2 successive orders of Runge-Kutta (RK) methods in order to adjust the step size.

- Let's begin with an RK method of **order *m***, and another one of **order *m* + 1** for a **system of equations**:

System of ODEs

$$\boldsymbol{y}'(t) = \boldsymbol{F}(t, \boldsymbol{y})$$

$$\begin{bmatrix} y_1'(t) \\ y_2'(t) \\ \vdots \\ y_n'(t) \end{bmatrix} = \begin{bmatrix} f_1(t, y_1, y_2, \ldots, y_n) \\ f_2(t, y_1, y_2, \ldots, y_n) \\ \vdots \\ f_3(t, y_1, y_2, \ldots, y_n) \end{bmatrix}$$

Solution

$$\boldsymbol{y}(t) = \begin{bmatrix} y_1(t) \\ y_2(t) \\ \vdots \\ y_n(t) \end{bmatrix}$$

- Beginning with some initial value of the solution, ***y***(*t*), After one step in time (time $t \rightarrow t + h$) we can use the 2 RK methods to obtain approximations for the next time step as:

$$\boldsymbol{y}_m(t+h), \qquad \boldsymbol{y}_{m+1}(t+h)$$

Estimated solution at next time step using RK order *m*

Estimated solution at next time step using RK order *m* + 1

- We estimate the truncation error similarly to before, by taking the difference of the 2 approximations:

$$\boldsymbol{E}(h) = \boldsymbol{y}_{m+1}(t+h) - \boldsymbol{y}_m(t+h)$$

- Now since we have a choice of RK methods of any order we apply the 4th and 5th order RK methods first used by Erwin Fehlberg (1969), known as **RKF45** (equations are shown on the next page).

$$\boldsymbol{k_1} = h\boldsymbol{F}(t, \boldsymbol{y})$$

$$\boldsymbol{k_{i\neq 1}} = h\boldsymbol{F}\left(t + A_i h, \boldsymbol{y} + \sum_{j=0}^{i-1} B_{ij}\boldsymbol{k_j}\right)$$

$$\boldsymbol{y_5}(t+h) = \boldsymbol{y}(t) + \sum_{i=1}^{6} C_i \boldsymbol{k_i}$$

$$\boldsymbol{y_4}(t+h) = \boldsymbol{y}(t) + \sum_{i=1}^{6} D_i \boldsymbol{k_i}$$

Cash & Karp coefficients

"Butcher Tableau"

i	A_i	B_{ij}					C_i	D_i
1	–	–	–	–	–	–	$\frac{37}{378}$	$\frac{2825}{27\,648}$
2	$\frac{1}{5}$	$\frac{1}{5}$	–	–	–	–	0	0
3	$\frac{3}{10}$	$\frac{3}{40}$	$\frac{9}{40}$	–	–	–	$\frac{250}{621}$	$\frac{18\,575}{48\,384}$
4	$\frac{3}{5}$	$\frac{3}{10}$	$-\frac{9}{10}$	$\frac{6}{5}$	–	–	$\frac{125}{594}$	$\frac{13\,525}{55\,296}$
5	1	$-\frac{11}{54}$	$\frac{5}{2}$	$-\frac{70}{27}$	$\frac{35}{27}$	–	0	$\frac{277}{14\,336}$
6	$\frac{7}{8}$	$\frac{1631}{55296}$	$\frac{175}{512}$	$\frac{575}{13824}$	$\frac{44275}{110592}$	$\frac{253}{4096}$	$\frac{512}{1771}$	$\frac{1}{4}$

- There are many choices we can make for the coefficients in the RK methods, the ones by Cash and Karp (1990) are deemed to be better than the original values proposed by Fehlberg.

- The solution at the next time step is taken from the 5th order RK method so the 4th order one is only used in estimating the truncation error as follows:

$$\boldsymbol{E}(h) = \boldsymbol{y_5}(t+h) - \boldsymbol{y_4}(t+h) = \sum_{i=1}^{6}(C_i - D_i)\boldsymbol{k_i}$$

  which is an error vector giving the estimated error for each of the dependent variables y_i which will be $O(h^5)$ because of the 4th order RK local error.

- We adopt the **root-mean square (RMS) error** as our metric though it is possible to use other metrics such as the maximum error component.

RMS error

$$e(h) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} E_i^2(h)}$$

- Since the local truncation error is determined by the RK4 method, taking 2 step sizes results in:

$$\frac{e(h_1)}{e(h_2)} \approx \left(\frac{h_1}{h_2}\right)^5$$

- Now we select a tolerance for each step, ε, and assuming that our first choice of step size, h_1, results in the error $e(h_1)$, we can obtain the required second step size by setting $\varepsilon = e(h_2)$:

$$h_2 = h_1 \left(\frac{\varepsilon}{e(h_1)}\right)^{\frac{1}{5}}$$

- So if we find that $h_2 < h_1$ we know that the initial step size wasn't small enough to meet the tolerance and we must repeat the RK5 method using the smaller step size.

- Conversely if h_2 is larger the next step size can be increased.

EXAMPLE 1 Use the RKF45 method to solve $y' = 4e^{0.8t} - 0.5y$ from $t = 0$ to 8 with an initial step size of $h = 1$ and initial condition $y(0) = 2$. The tolerance at each step should be $\varepsilon = 0.01$.

- The results from applying the RKF45 method with Carp/Kash coefficients are:

y	$\lvert e \rvert$
6.19449134454593	8.0479247803744 2e-05
14.8435240841671	0.000162222930343958
33.6762333750447	0.000350791342526691
75.3368415622294	0.000774488150526054
167.901168029317	0.00171988710764026
373.814028316603	0.00382539376374780
832.025218299412	0.00851218423338196
1851.75882283550	0.0189433736575211

Error for this step size is too big

- Calculate the step size it should be:

$$h_2 = 1\left(\frac{0.01}{0.0189\ldots}\right)^{\frac{1}{5}} = 0.88005\ldots$$

- Re-calculate the step with the new step size of 0.88 from the *y* value 832.025218299412 at *t* = 7:

$$y(7.88) = 1682.268480626394$$

which gives an error of 0.009189635052280 < 0.01.

- Then we compute the final value from this one using the remaining step size of 0.12:

$$y(8) = 1851.780743268326$$

Other Adaptive Runge-Kutta

- Similar methods exist comparing different orders of RK.

- Also various values for the coefficients have been published in computational mathematical journals. For example we have the Bogacki & Shampine (1989) formulas for RK methods of **orders 2 and 3**:

$$y_{i+1} = y_i + \frac{1}{9}(2k_1 + 3k_2 + 4k_3)h$$

$$k_1 = f(t_i, y_i)$$

$$k_2 = f\left(t_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1 h\right)$$

$$k_3 = f\left(t_i + \frac{3}{4}h, y_i + \frac{3}{4}k_2 h\right)$$

- Another famous result is the Dormand & Prince formula for RK methods of **orders 4 and 5**, that give different coefficients to that of Cash & Karp.

14.2 Matlab Built-in Functions

- Matlab implements the 2 methods mentioned previously in the following functions:

ode23 — **Bogacki & Shampine**

ode45 — **Dormand & Prince**

- Matlab also implements a particular multi-step method known as the Adams-Bashforth-Moulton method:

ode113

Adams-Bashforth-Moulton

- The 45 method is the most commonly used one however 113 can be used when the error tolerances must be extremely tight.

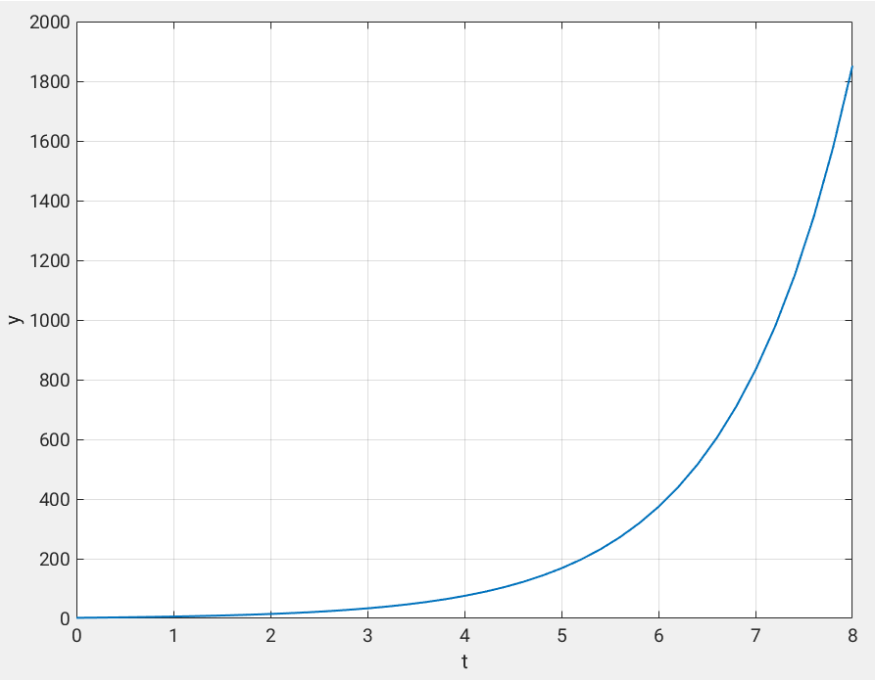
EXAMPLE 2 Repeat **Example 1** using the **ode45** solver.

```
>> dy = @(t,y) 4*exp(0.8*t)-0.5*y;
>> [t,y] = ode45(dy,[0,8],2);
>> plot(t,y,'linewidth',1.5)
>> grid on
>> xlabel('t')
>> ylabel('y')
```

Initial condition

Time interval

- Notice we must define the function with the independent variable first.
- The other solvers have a similar syntax that can be checked using the Matlab documentation.

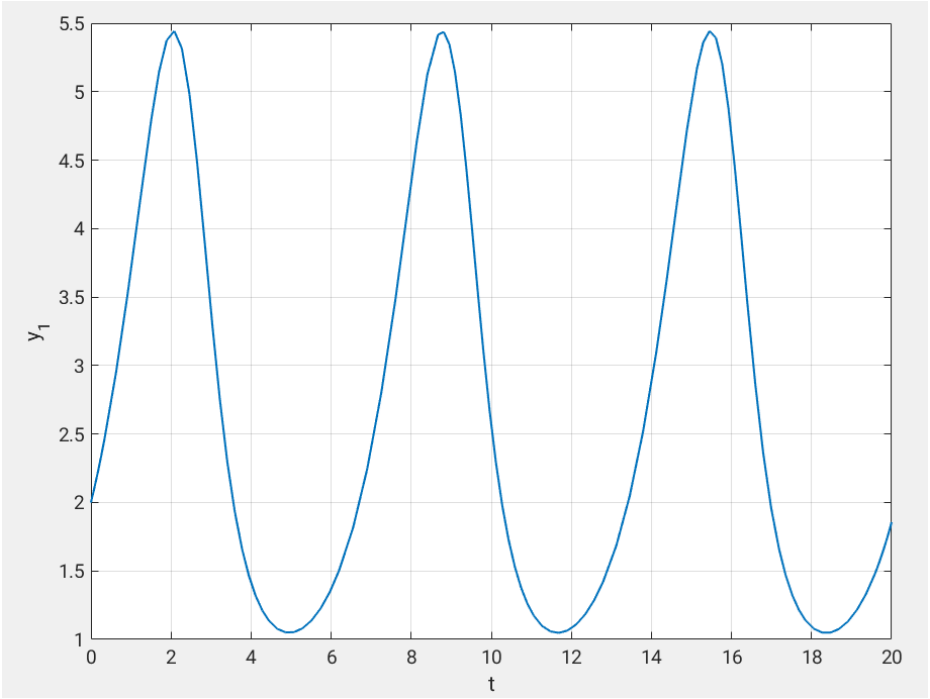


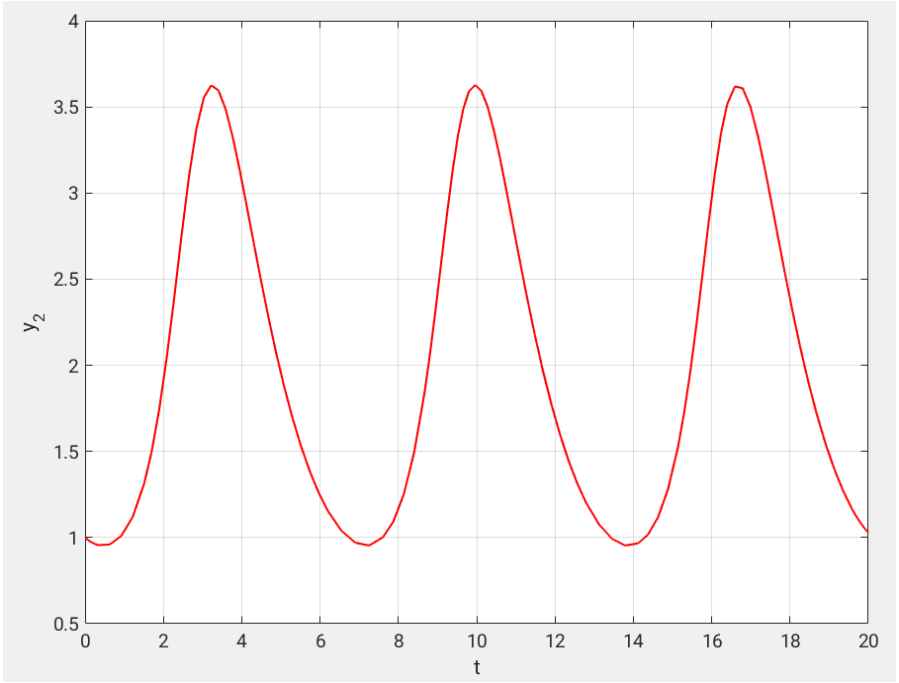
EXAMPLE 3 Solve the following system of ODEs with **ode45** from $t = 0$ to 20 and an initial condition of $\boldsymbol{y}(0) = (y_1(0), y_2(0)) = (2,1)$. The equations represents a predator-prey system.

$$\frac{dy_1}{dt} = 1.2y_1 - 0.6y_1\,y_2 \qquad \frac{dy_2}{dt} = -0.8y_2 + 0.3y_1\,y_2$$

```
%% Function file to store Predator-Prey
equations
function dy = pred_prey(t,y)
dy = [1.2*y(1)-0.6*y(1)*y(2);
    -0.8*y(2)+0.3*y(1)*y(2)];
```

```
>> [t,y] = ode45(@pred_prey,[0,20],[2,1]);
>> plot(t,y(:,1),'linewidth',1.5)
>> grid on
>> figure
>> plot(t,y(:,2),'r','linewidth',1.5)
>> grid on
```



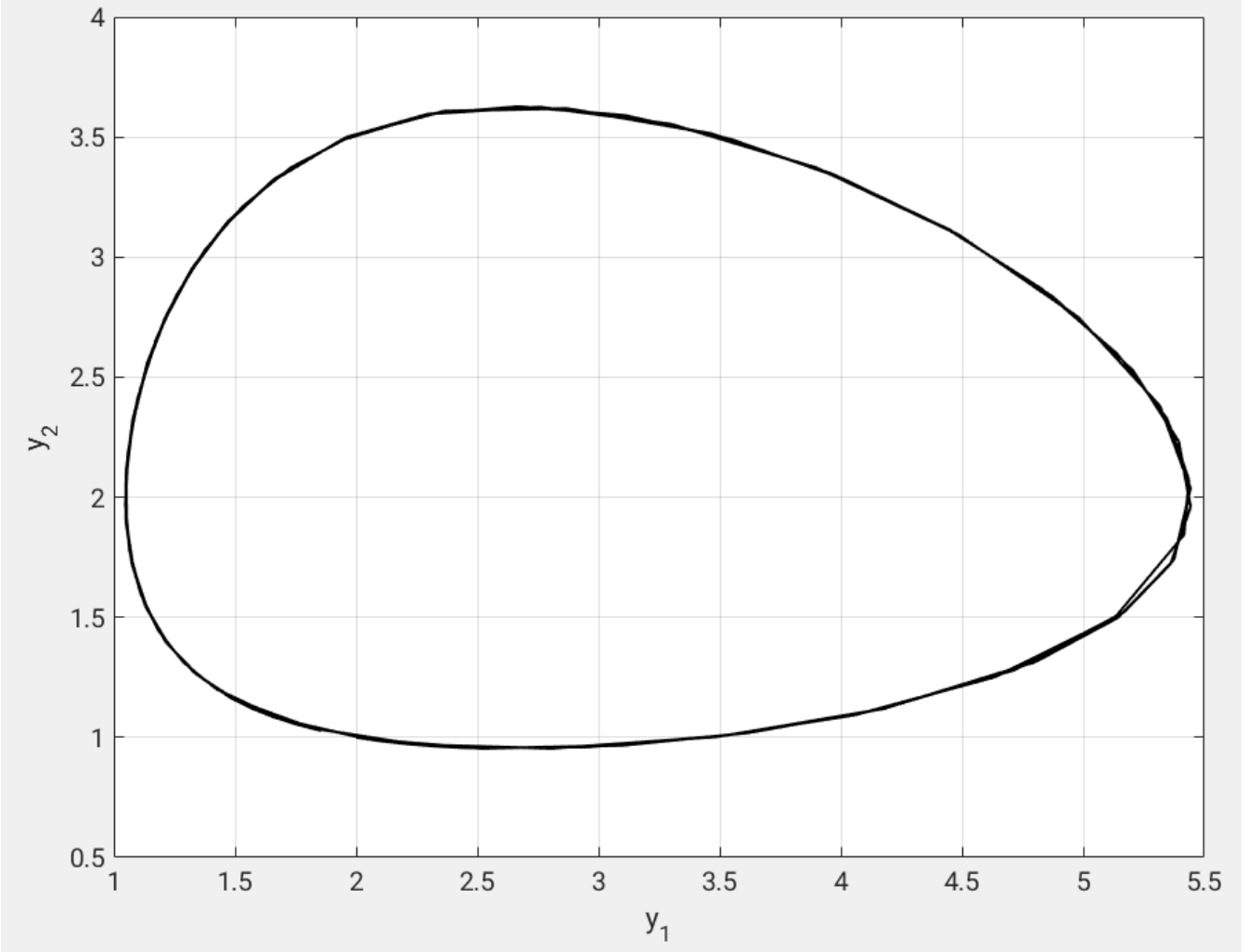


- The above plots show the progression of each of the dependent variables over time.

- In the analysis of differential equations it is common to plot the dependent variables against each other in what is known as a **phase diagram**. The solution paths are known as **trajectories**.

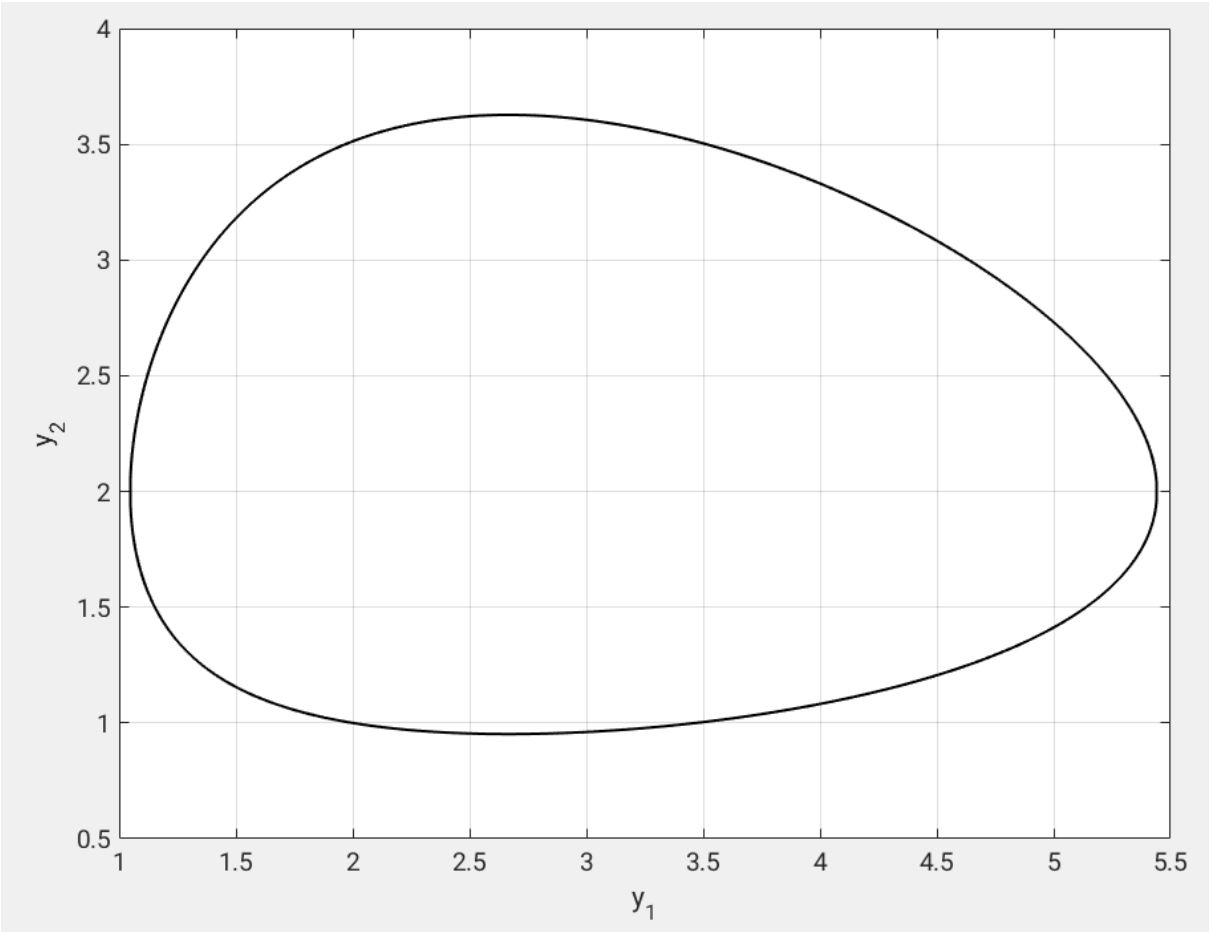
```
>> plot(y(:,1),y(:,2),'k','linewidth',1.5)
>> grid on
>> xlabel('y_1')
>> ylabel('y_2')
```

- Notice the trajectory is not smooth indicating that perhaps we should have taken a smaller time step then Matlab's default algorithm.



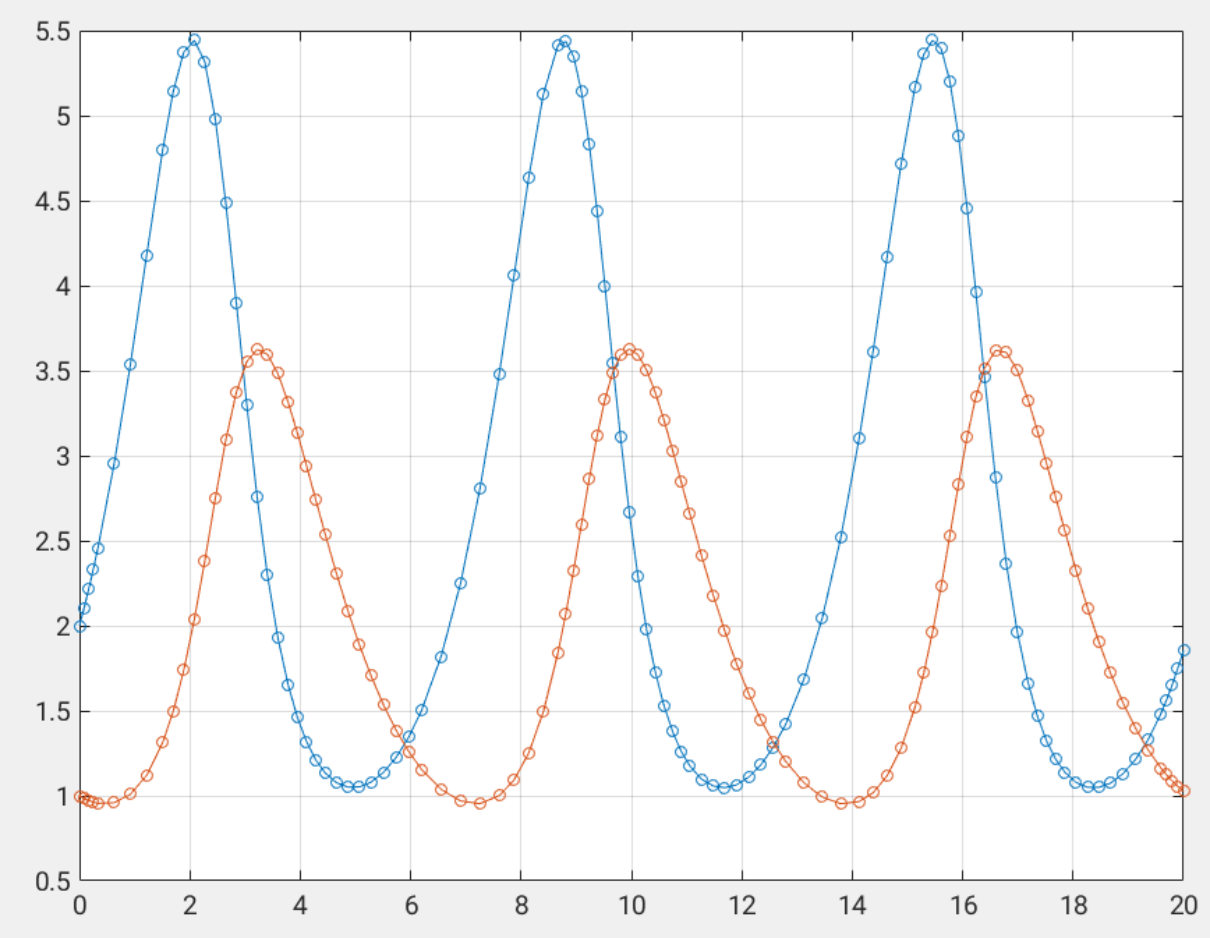
```
>> options = odeset('MaxStep',1e-3);
>> [t,y] = ode45(@pred_prey,[0,20],[2,1],options);
>> figure
>> plot(y(:,1),y(:,2),'k','linewidth',1.5)
>> grid on
```

- We use the **odeset** function to specify parameters of the solver algorithm.
- Now the trajectory is smooth.
- Other options exist too, and we can also pass parameters into the pred_prey function after the options argument.



- Matlab automatically generates a plot with markers on the time steps that it has evaluated when an ODE solver is called without specifying an output argument.

```
>> ode45(@pred_prey,[0,20],[2,1]);
>> grid on
```



- Some options that can be set in Matlab's ODE solvers:

`'RelTol'`	Allows you to adjust the relative tolerance.
`'AbsTol'`	Allows you to adjust the absolute tolerance.
`'InitialStep'`	The solver automatically determines the initial step. This option allows you to set your own.
`'MaxStep'`	The maximum step defaults to one-tenth of the `tspan` interval. This option allows you to override this default.

- Pass parameters into the ODE function file:

```
[t, y] = ode45(odefun, tspan, y0, options, p1, p2,...)
```

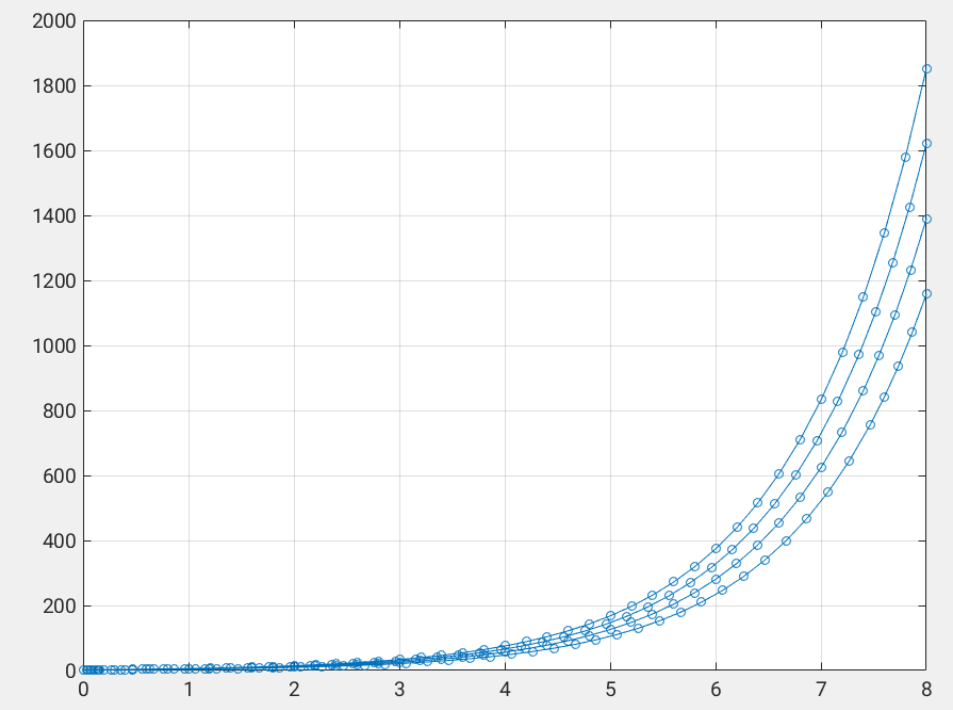
These get passed into the function to be solved

- If you want to pass parameters into the function with no special options then put empty square brackets **[]** where *options* is.

EXAMPLE 4 Repeat Example 1 but try different values of the coefficient of the exponential term by passing an argument to **ode45**.

```
>> dy = @(t,y,c) c*exp(0.8*t)-0.5*y;
>> ode45(dy,[0,8],2,[],4);
>> grid on
>> hold on
>> ode45(dy,[0,8],2,[],3.5);
>> ode45(dy,[0,8],2,[],3);
>> ode45(dy,[0,8],2,[],2.5);
```

Values of *c* = 4, 3.5, 3 and 2.5



14.3 Events

- Sometimes we don't know the time we wish to solve our differential equation up to.

- Instead we might want to know at what time a particular value of the solution is reached.

- We do this by using the **events option** in the ode solvers.

EXAMPLE 5 The following ODE system represents the position and velocity of a free-falling object.

$$\frac{dx}{dt} = v$$

$$\frac{dv}{dt} = g - \frac{c_d}{m} v|v|$$

x = distance (m), t = time (s), v = velocity (m/s) where positive velocity is in the downward direction, g = the acceleration of gravity (= 9.81 m/s^2), c_d = drag coefficient (kg/m), and m = mass (kg)

- Distance and velocity are positive in the downward direction. Ground level is defined as zero distance. The initial position of the jumper 200 m above the ground and the initial velocity is 20 m/s in the upward direction:

$$x(0) = -200,\ v(0) = -20$$

- Find the time when the jumper reaches ground level.

```
function dydt = freefall(t,y,cd,m)
% y(1) = x and y(2) = v
grav = 9.81;
dydt = [y(2);grav-cd/m*y(2)*abs(y(2))];
```

Function file for the ODE system

- We now write an event function that we pass to **ode45** that has 3 output arguments.

- The first tells Matlab what value of the dependent variable to look for, the second tells Matlab to stop solving, the third specifies the direction.

```
function [detect,stopint,direction] = endevent(t,y,varargin)
% Locate the time when height passes through zero
% and stop integration.
detect = y(1);
% Detect height = 0
stopint = 1;
% Stop the integration
direction = 0;
% Direction does not matter
```

Function file for the event

- The detect value must be set to 0.
- This means if we want to find when the jumper reaches a height 12m above the ground we would write:

```
detect = y(1) - 12;
```

- If we wanted to know when the jumper reaches a height in only the negative direction we would use direction = – 1 (+1 for positive direction).

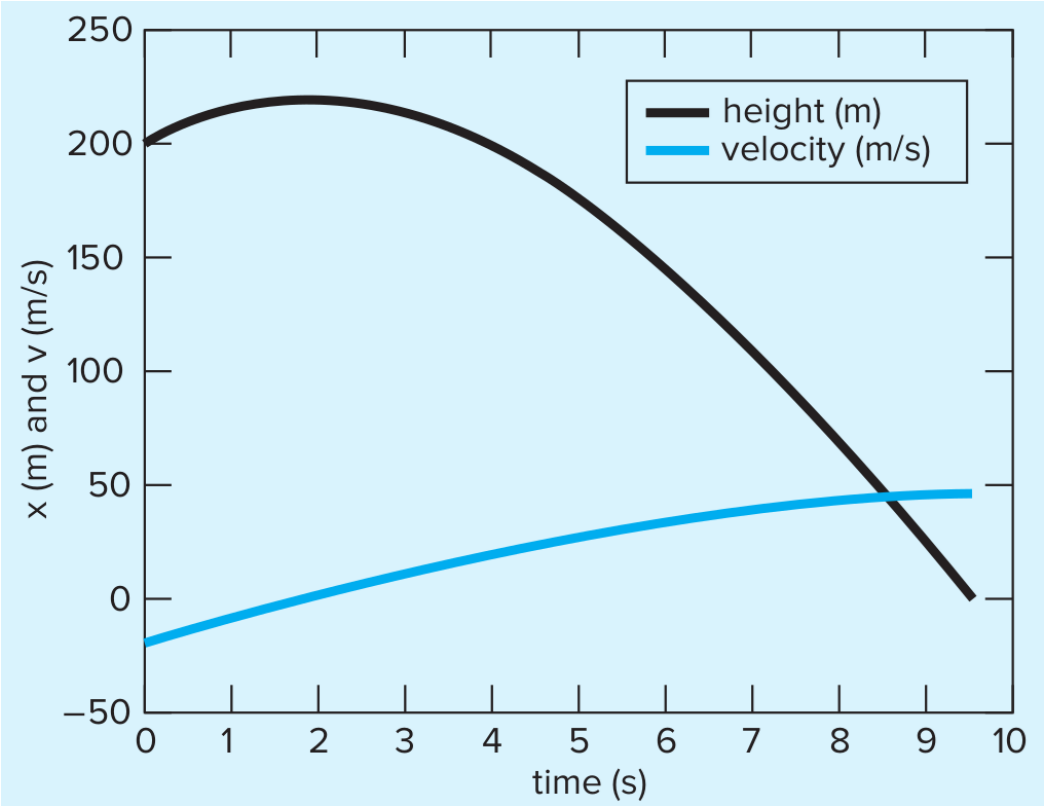
```
>> opts = odeset('events',@endevent);
>> y0 = [-200,-20];
>> [t,y,te,ye] = ode45(@freefall,[0,inf],y0,opts,0.25,68.1);
>> plot(t,-y(:,1),'-',t,y(:,2),'--','LineWidth',2)
>> legend('Height (m)','Velocity (m/s)')
>> xlabel('time (s)');
>> ylabel('x (m) and v (m/s)')
>> te, ye

te =
            9.5475
ye =
            0.0000          46.2454
```

Integrate with upper limit of infinity

Drag coefficient and mass

te and ye are the time and values (height & velocity) of the event

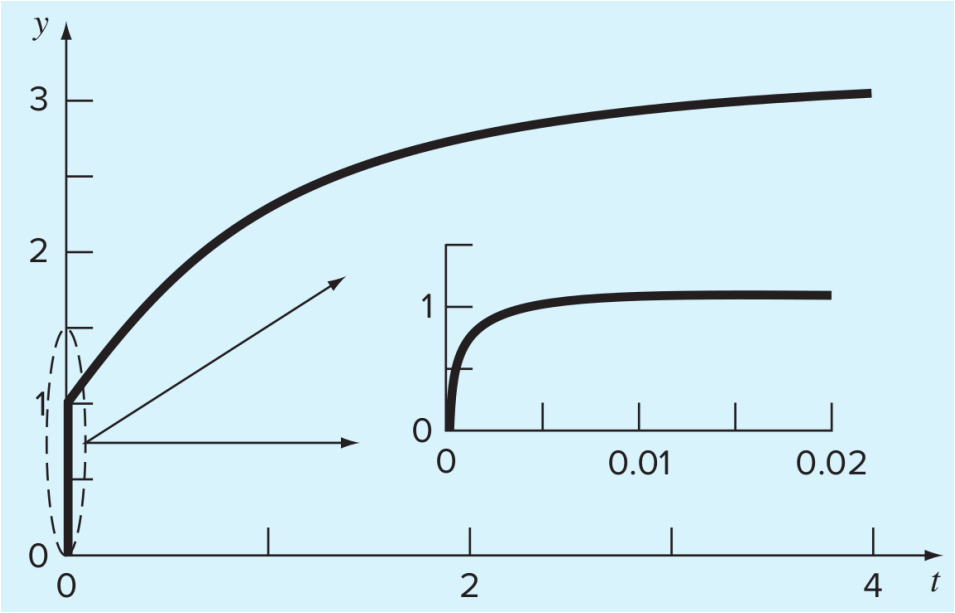


14.4 Stiffness

- ODE systems that have components some of which change quickly, and some of which change slowly are called **stiff systems**.

- The allowable step size is determined by the rapidly changing component in order to fully capture the qualities of the system.

- In the figure the solution near $t = 0$ looks to be 1, however zooming in we see it is actually a curve starting at the origin. We must take small step sizes to account for this.



EXAMPLE 6 The following stiff system requires very small step sizes to capture the true dynamics.

$$\frac{dy}{dt} = -1000y + 3000 - 2000e^{-t} \qquad y(0) = 0$$

- Using analytic techniques the exact solution is:

$$y = 3 - 0.998e^{-1000t} - 2.002e^{-t}$$

- For the first very small interval of time both exponential terms are comparable and must be included, however afterwards the 1st exponential term will be dominated by the 2nd.

- As we saw before, a general ODE $\frac{dy}{dt} = -ay \quad y(0) = y_0$ has exponential solution:

$$y = y_0 e^{-at}$$

which approaches 0 as $t \to \infty$.

- Approximating with Euler's method, $y_{i+1} = y_i + \frac{dy_i}{dt}h$, the ODE formula is:

$$y_{i+1} = y_i - ay_ih = y_i(1 - ah)$$

which is only stable (approaches the solution) when $|1 - ah| < 1$.

- From the 1st exponential term in our equation we require,

$$h < \frac{2}{1000} = 0.002$$

which is small compared with the required step size for the 2nd exponential term.

- We can address this issue by formulating a slightly different version of Euler's method known as the **Implicit Euler Method**.

- The implicit Euler method evaluates the derivative at the current time step:

$$y_{i+1} = y_i + \frac{dy_{i+1}}{dt} h$$

- Applying this to our ODE results in:

$$y_{i+1} = y_i - a y_{i+1} h \longrightarrow y_{i+1} = \frac{y_i}{1 + a h}$$

which is stable for any step size.

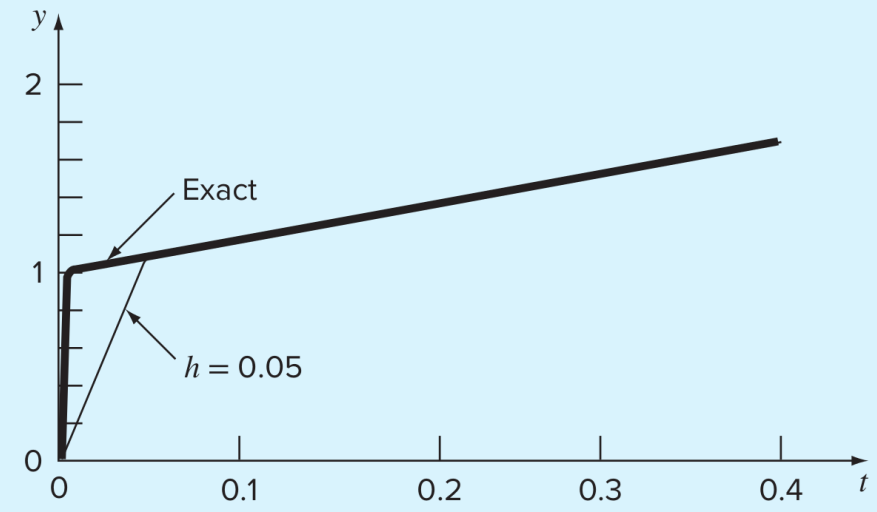
$$y_{i+1} = y_i + (-1000 y_{i+1} + 3000 - 2000 e^{-t_{i+1}}) h$$

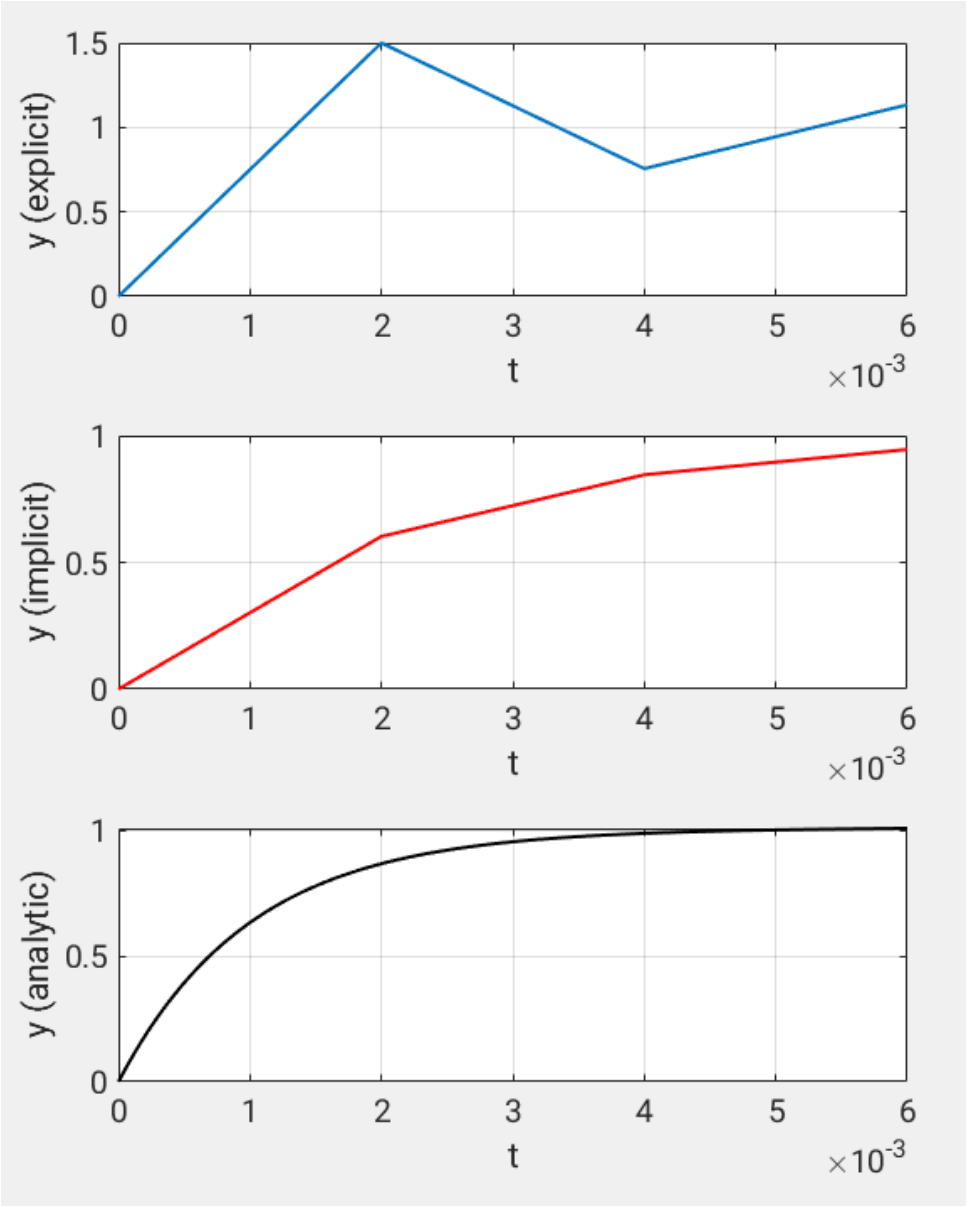
Implicit Euler formula

$$y_{i+1} = \frac{y_i + 3000h - 2000he^{-t_{i+1}}}{1 + 1000h}$$

- The graph on the right compares the explicit and implicit Euler method alongside the exact solution for a step size of $h = 0.0015$.

- Even for a much larger step size of $h = 0.05$ the implicit Euler method converges to the exact solution:





Implicit Euler for Systems of ODEs

- The system,

$$\frac{dy_1}{dt} = -5y_1 + 3y_2 \qquad y_1(0) = 52.29$$

$$\frac{dy_2}{dt} = 100y_1 - 301y_2 \qquad y_2(0) = 83.82$$

is stiff since we can see from it's exact solution the difference in magnitude of various terms:

$$y_1 = 52.96e^{-3.9899t} - 0.67e^{-302.0101t}$$

$$y_2 = 17.83e^{-3.9899t} + 65.99e^{-302.0101t}$$

- We apply the implicit Euler method:

$$y_{1,i+1} = y_{1,i} + (-5y_{1,i+1} + 3y_{2,i+1})h$$

$$y_{2,i+1} = y_{2,i} + (100y_{1,i+1} - 301y_{2,i+1})h$$

- Re-arranging the previous equations gives us the implicit Euler iterative formula:

$$(1+5h)y_{1,i+1} - 3hy_{2,i+1} = y_{1,i}$$

$$(1+301h)y_{2,i+1} - 100hy_{1,i+1} = y_{2,i}$$

- We have to solve simultaneous equations at each time step.

- For highly nonlinear systems we would have to employ a root finding method in conjunction with this in order to solve each time step which may be more costly then simply taking smaller time steps using the explicit Euler method.

- Therefore for systems, the advantage of the implicit Euler method may be counteracted by having to solve simultaneous equations.

14.5 Matlab Stiff System Solvers

- Matlab has the following built-in functions to solve stiff systems:

  ode15s **ode23s** **ode23t** **ode23tb**

- Their use is similar to the previous solvers however have special ways of dealing with stiff systems embedded into their algorithms.

- Each can be useful depending on the system you are trying to solve so it is a matter of trying to find one that will work for the application at hand.

- If your system is large and very stiff investigate the time taken to compute small scale solutions with each solver to see which one is more effective. Choose that one for your large scale simulation.

EXAMPLE 7 Solve the following stiff system (Van der Pol equations), which models an electrical circuit, for μ = 1 (non-stiff) from t = 1 to 20 and μ = 1000 (stiff) from t = 1 to 6000.

$$\frac{d^2y_1}{dt^2} - \mu\left(1 - y_1^2\right)\frac{dy_1}{dt} + y_1 = 0 \qquad y_1(0) = 1,\ y_2(0) = 1$$

Convert to system of ODEs

$$\frac{dy_1}{dt} = y_2$$

$$\frac{dy_2}{dt} = \mu\left(1 - y_1^2\right)y_2 - y_1 = 0$$

Function file to store the system

```
%% Function for Van der Pol
Oscillator
function dy = vanderpol(t,y,mu)
dy = [y(2);mu*(1-y(1)^2)*y(2)-y(1)];
```

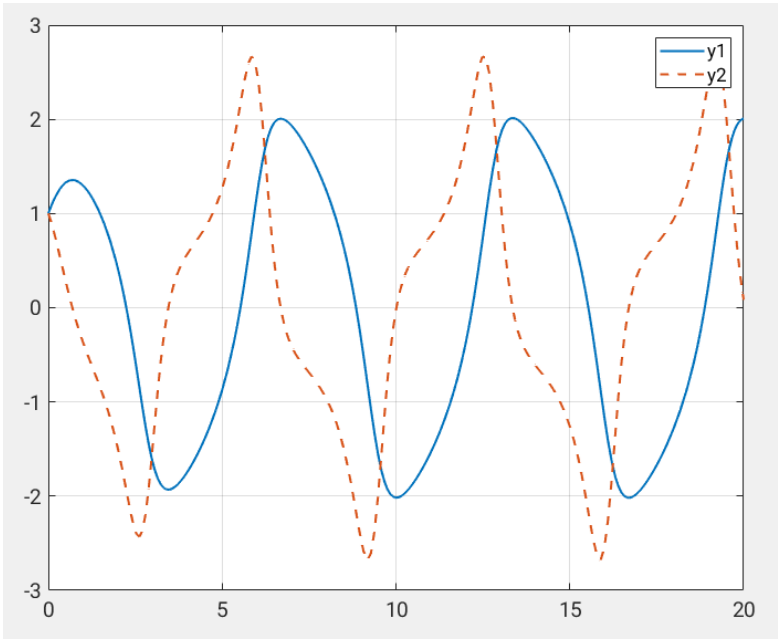
Solve non-stiff system

```
>> [t,y] = ode45(@vanderpol,[0,20],[1,1],[],1);
>> plot(t,y(:,1),'-',t,y(:,2),'--')
>> legend('y1','y2');
>> grid on
```

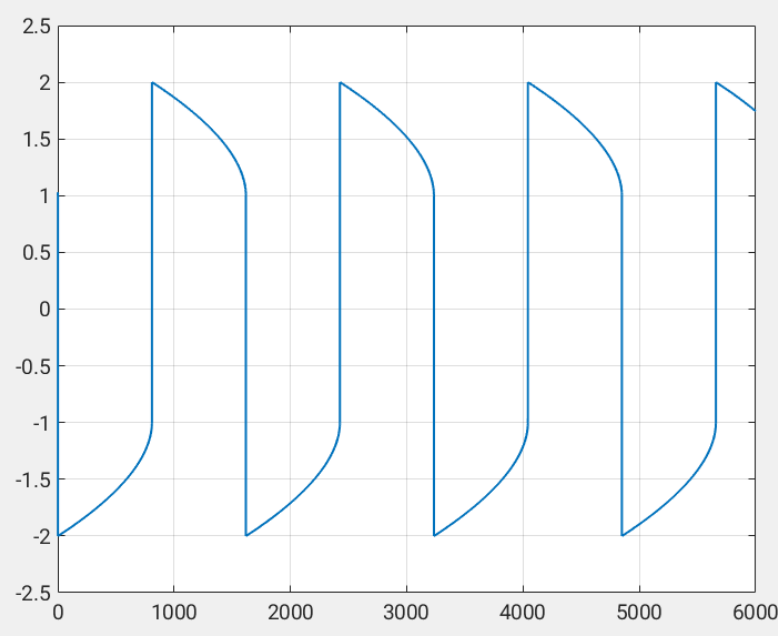
- Solving with ode45 results in a smooth curve since the system is non-stiff.

- However if we try to solve the case with μ = 1000 ode45 takes an extremely long time to try to solve and we must resort to ode23s.

```
>> [t,y] = ode23s(@ vanderpol,[0,6000],[1 1],[],1000);
>> plot(t,y(:,1),'-')
>> grid on
```



y_1 for stiff case →



EXAMPLE 8 Compare the different stiff solvers for the following equation with initial condition $y(0) = 1$ for $t = 0$ to 0.5.

$$y' = -(1e9)y$$

- The **tic toc** function times how long a computation takes.

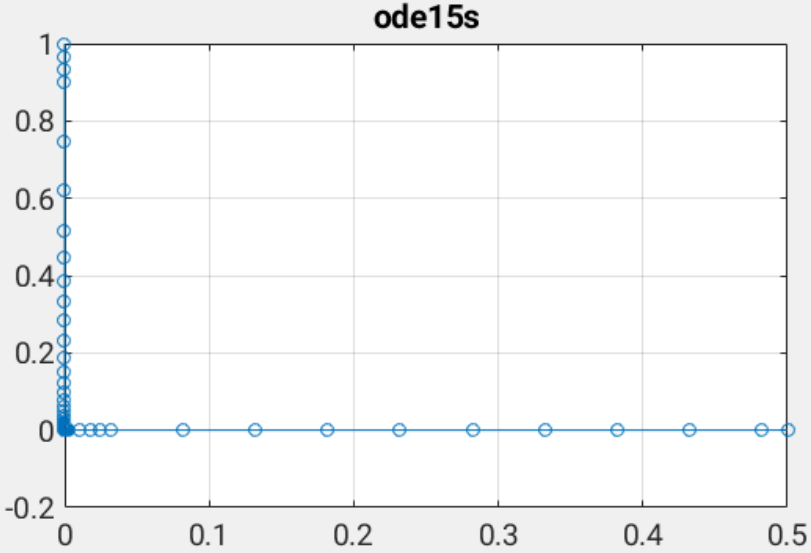
```
y0 = 1;
tspan = [0 0.5];
options = odeset('Stats','on');
subplot(2,2,1)
tic
ode15s(@(t,y) -1e9*y, tspan, y0, options)
toc
title('ode15s')
grid on
```

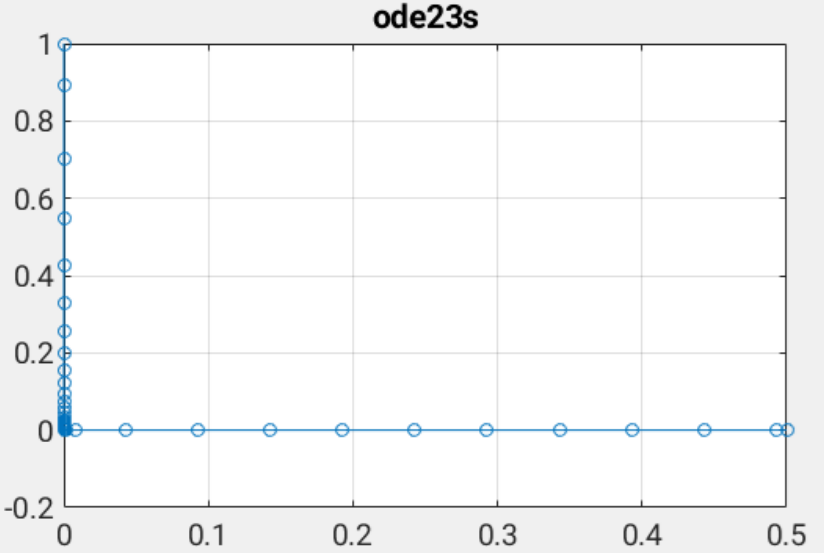
We run these command and the following in a script so as to not affect the time it takes to type in the tic toc function

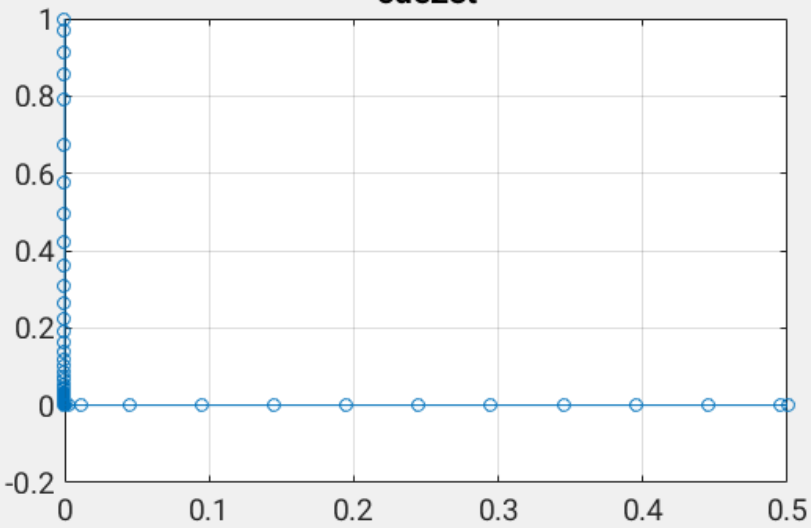
```
subplot(2,2,2)
tic
ode23s(@(t,y) -1e9*y, tspan, y0, options)
toc
title('ode23s')
grid on
```

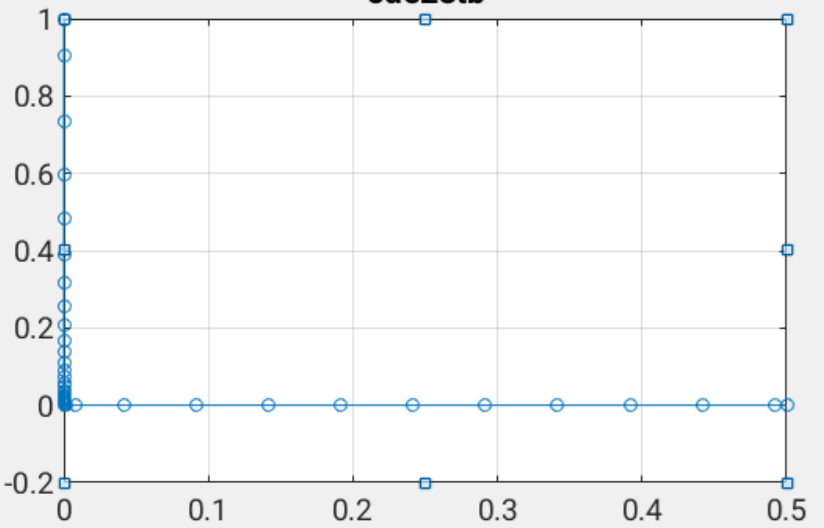
```
subplot(2,2,3)
tic
ode23t(@(t,y) -1e9*y, tspan, y0,
options)
toc
title('ode23t')
grid on
```

```
subplot(2,2,4)
tic
ode23tb(@(t,y) -1e9*y, tspan, y0,
options)
toc
title('ode23tb')
grid on
```


ode15s
1
0.8
0.6
0.4
0.2
0
-0.2
0
0.1
0.2
0.3
0.4
0.5


ode23s
1
0.8
0.6
0.4
0.2
0
-0.2
0
0.1
0.2
0.3
0.4
0.5


ode23t
1
0.8
0.6
0.4
0.2
0
-0.2
0
0.1
0.2
0.3
0.4
0.5


ode23tb
1
0.8
0.6
0.4
0.2
0
-0.2
0
0.1
0.2
0.3
0.4
0.5

ode15s

104 successful steps
1 failed attempts
213 function evaluations
1 partial derivatives
21 LU decompositions
210 solutions of linear systems
Elapsed time is 0.551393 seconds.

ode23s

63 successful steps
0 failed attempts
254 function evaluations
63 partial derivatives
63 LU decompositions
189 solutions of linear systems
Elapsed time is 0.486404 seconds.

ode23t

95 successful steps
0 failed attempts
126 function evaluations
1 partial derivatives
28 LU decompositions
123 solutions of linear systems
Elapsed time is 0.561366 seconds.

ode23tb

71 successful steps
0 failed attempts
168 function evaluations
1 partial derivatives
23 LU decompositions
236 solutions of linear systems
Elapsed time is 0.508973 seconds.

- For this system ode23s gave the best results and would be suitable for large-scale simulation.

14.6 Boundary Value Problems (BVPs)

- Some ODE systems don't specify an initial time value, instead they specify values of the dependent variable or it's derivatives at various times/positions.

- Instead of initial conditions we call them **boundary conditions** and solve them using Matlab's built-in function **bvp4c** when the system is 2D.

- To solve BVPs we must provide points on the domain (known as a mesh) at which to evaluate the derivative function. We must also supply initial guesses since the method of solution involves solving systems of simultaneous equations.

- The method is a type of collocation method which essentially approximates the solution using a polynomial.

EXAMPLE 9 Solve the following boundary value problem.

$$\frac{d^2y}{dx^2} + y = 1 \qquad y(0) = 1 \qquad y(\pi/2) = 0$$

Convert into a system of ODEs

$$\frac{dy}{dx} = z$$

$$\frac{dz}{dx} = 1 - y$$

Function to hold system

```
%% Function for BVP
function dy = bvp_eg(x,y)
dy = [y(2); 1 - y(1)];
```

Function to hold BCs

```
%% Function for Boundary Conditions
function r = bcs(ya,yb)
r = [ya(1) - 1; yb(1)];
```

Notice the boundary conditions must be set equal to 0!!

Script to solve

```
solinit = bvpinit (linspace(0,pi/2,10),[1,-1]);
sol = bvp4c(@bvp_eg,@bcs,solinit);
x = linspace(0,pi/2);
y = deval(sol,x);
plot(x,y(1,:))
```

Mesh & initial guess

Evaluates BVP solution at each point *x*