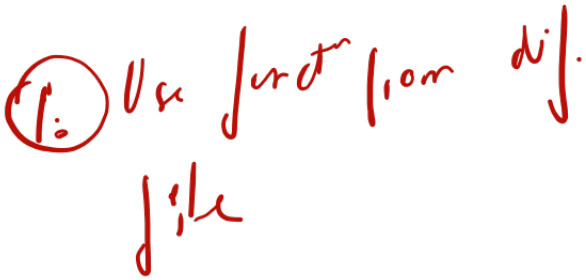
Solving Differential Equation using ODE Solvers in MATLAB



Edited by
Priyanka Shukla

Solving Differential Equations in MATLAB

MATLAB have lots of built-in functionality for solving differential equations. MATLAB includes functions that solve ordinary differential equations (ODE) of the form:

$$\frac{dy}{dt} = f(t, y), \qquad y(t_0) = y_0$$

MATLAB can solve these equations numerically.

Higher order differential equations must be reformulated into a system of first order differential equations.

Note! Different notation is used:

$$\frac{dy}{dt} = y' = \dot{y}$$

Not all differential equations can be solved by the same technique, so MATLAB offers lots of different ODE solvers for solving differential equations, such as **ode45**, **ode23**, **ode113**, etc.

Bacteria Population

In this task we will simulate a simple model of a bacteria population in a jar.

The model is as follows:

$$\text{birth rate} = bx$$

$$\text{death rate} = px^2$$

Then the total rate of change of bacteria population is:

$$\dot{x} = bx - px^2$$

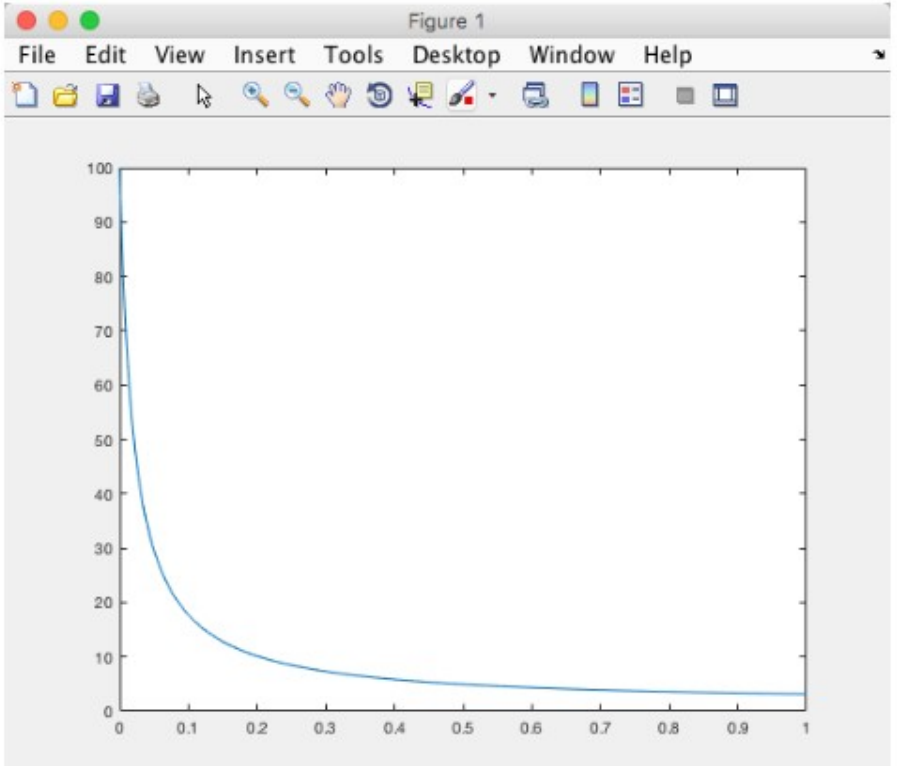
Set ***b=1***/hour and ***p=0.5*** bacteria-hour

→ Simulate the number of bacteria in the jar after **1 hour**, assuming that initially there are **100 bacteria** present.

@ passes a reference handle to the function. So ode45 uses it as a parameter instead of evaluating directly

```
function dx = bacteriadiff(t,x)
% My Simple Differential Equation

b=1;
p=0.5;

dx = b*x - p*x^2;
```

```
clear
clc

tspan=[0 1];
x0=100;

[t,y]=ode45(@bacteriadiff, tspan,x0);
plot(t,y)

[t,y]
```



```
dx = @(t,x) x - 0.5*x.^2 ;
[t,y] = ode45(dx, [0 1], 100) ;
```

Passing Parameters to the model

Given the following system (1.order differential equation):

$$\dot{x} = ax + b$$

where $a = -\frac{1}{T}$,where T is the time constant

In this case we want to pass a and b as parameters, to make it easy to be able to change values for these parameters

We set $b = 1$

We set initial condition $x(0) = 1$ and $T = 5$.

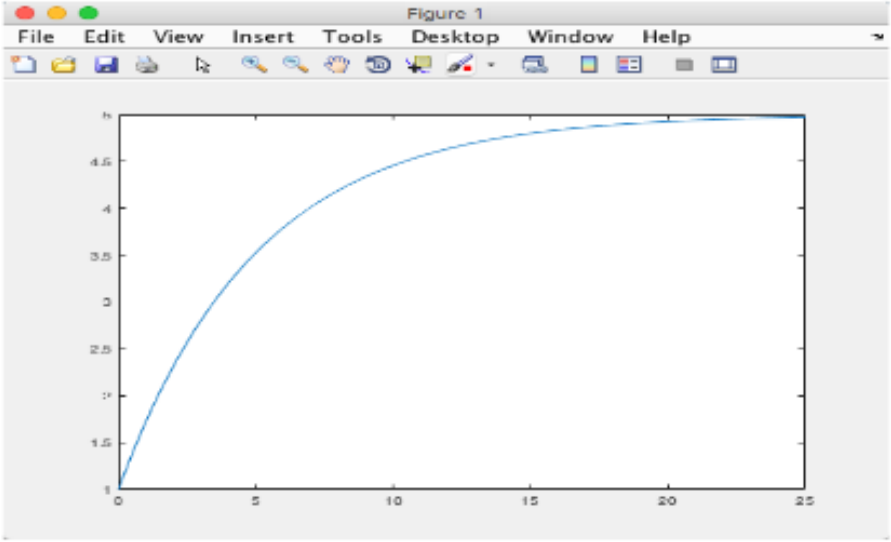
Solve the Equation and Plot the results with MATLAB

```
function dx = mysimplediff(t,x,param)
% My Simple Differential Equation

a=param(1);
b=param(2);

dx=a*x+b;
```

```
tspan=[0 25];
x0=1;
a=-1/5;
b=1;
param=[a b];

[t,y]=ode45(@mysimplediff, tspan,
x0,[], param);
plot(t,y)
```

By doing this, it is very easy to changes values for the parameters a and b.

Note! We need to use the 5. argument in the ODE solver function for this. The 4. argument is for special options and is normally set to "[]", i.e., no options.



Differential Equation

Use the ode23 function to solve and plot the results of the following differential equation in the interval $[t_0, t_f]$:

$$w' + (1.2 + sin10t)w = 0$$

Where:

$$t_0 = 0$$
$$t_f = 5$$
$$w(t_0) = 1$$

F = @(t,y) y

ode45(F, [0:0.1:5], 1)

Differential Equation

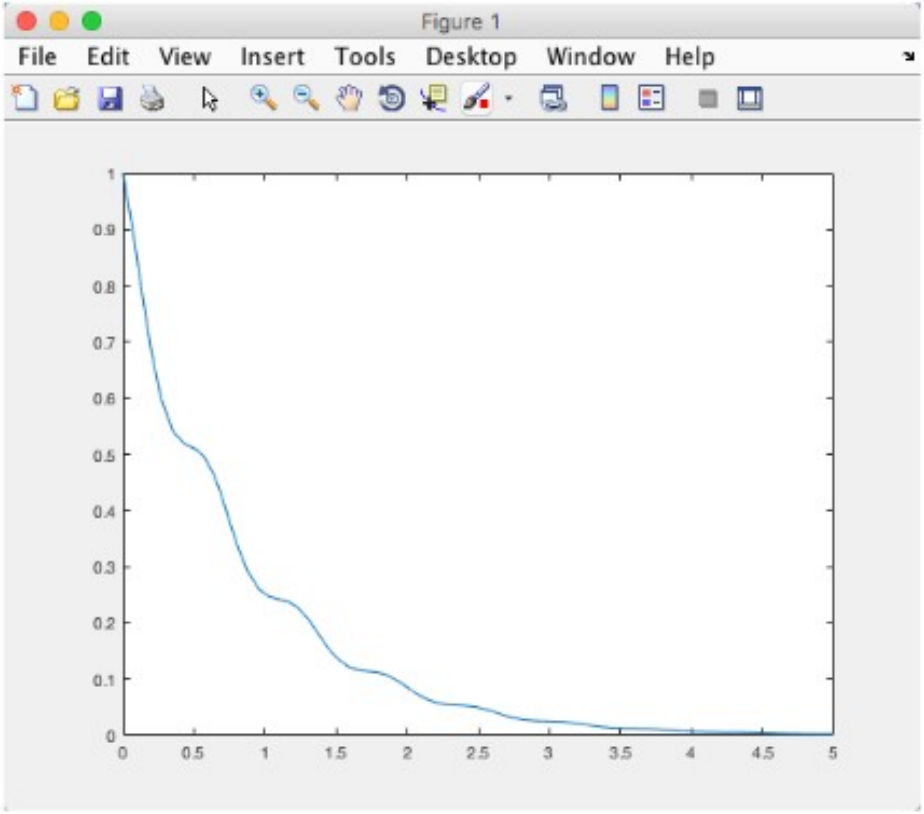
We start by rewriting the differential equation:

$$w' = -(1.2 + sin10t)w$$

Then we can implement it in MATLAB

```
function dw = diff_task3(t,w)

dw = -(1.2 + sin(10*t))*w;
```

```
tspan=[0 5];
w0=1;

[t,w]=ode23(@diff_task3, tspan, w0);
plot(t,w)
```



2.order differential equation

Use the ode23/ode45 function to solve and plot the results of the following differential equation in the interval $[t_0, t_f]$:

$$(1 + t^2)\ddot{w} + 2t\dot{w} + 3w = 2$$

Where; , $t_0 = 0, t_f = 5, w(t_0) = 0, \dot{w}(t_0) = 1$

Note! Higher order differential equations must be reformulated into a system of first order differential equations.

Tip 1: Reformulate the differential equation so $\ddot{w}$ is alone on the left side.

Tip 2: Set:

$$w = x_1$$
$$\dot{w} = x_2$$

2.order differential equation

Tip1: First we rewrite like this:

$$\ddot{w} = \frac{2 - 2t\dot{w} - 3w}{(1 + t^2)}$$

Tip2: In order to solve it using the ode functions in MATLAB it has to be a set with 1.order ode's. So we set:

$$w = x_1 \Rightarrow \dot{x}_1 = \dot{w} = x_2$$
$$\dot{w} = x_2$$

This gives 2 first order differential equations:

$$\dot{x}_2 = \frac{2 - 2x_2 - 3x_1}{(1 + t^2)}$$

$$\dot{x}_1 = x_2$$

$$\dot{x}_2 = \frac{2 - 2tx_2 - 3x_1}{(1 + t^2)}$$

ode23 updates both state variables x(1) & x(2)

x(1) = position ; x(2) = velocity

x(:,1) = soln for position ; x(:,2) = soln for velocity.

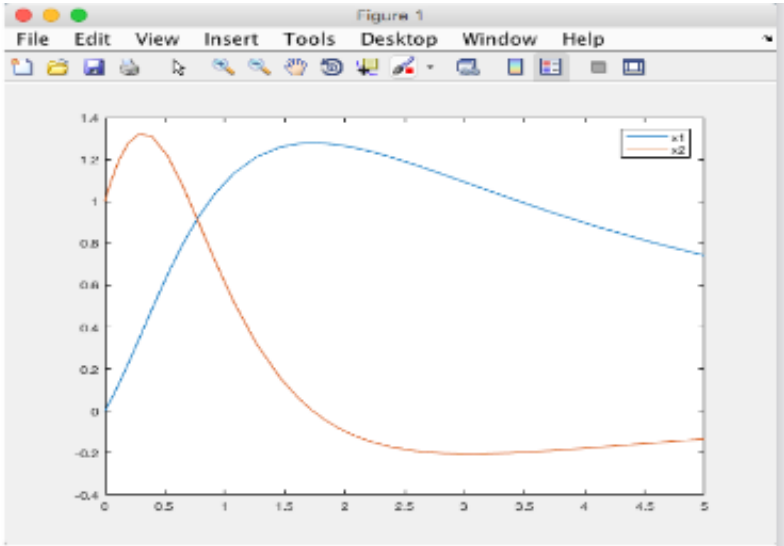
```
function dx = diff_secondorder(t,x)

[m,n] = size(x);
dx = zeros(m,n)

dx(1) = x(2);
dx(2) = (2-2*t*x(2)-3*x(1))/(1+t^2);
```

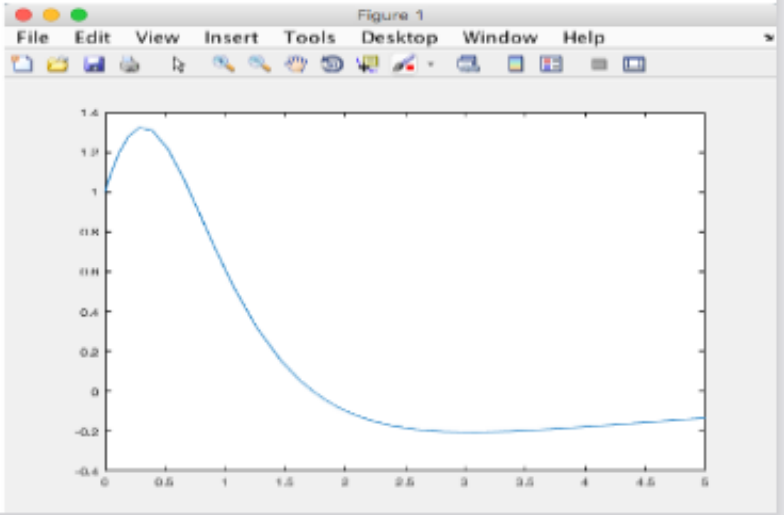
```
tspan=[0 5];
x0=[0; 1];

[t,x]=ode23(@diff_secondorder, tspan, x0);
plot(t,x)
legend('x1','x2')
```

Then x(1) & x(2) are plotted

first is x(1)

```
tspan=[0 5];
x0=[0; 1];

[t,x]=ode23(@diff_secondorder, tspan, x0);
plot(t, x(:,2))
```





System of Coupled ODE

$$\begin{cases} x' = -ye^{(-t/5)} + y'e^{(-t/5)} + 1 \\ y'' = -2\sin(t) \end{cases}$$

- Step 1: Introduce a new variable that equals the first derivative of the free variable in the second order equation: $z = y'$
- Step 2: Taking the derivative of each side yields the following: $z' = y''$
- Step 3: Substituting the second order ODE with z and z':

$$\begin{cases} x' = -ye^{(-t/5)} + ze^{(-t/5)} + 1 \\ z' = -2\sin(t) \\ y' = z \end{cases}$$

Methods:

- Step 4: Write a MATLAB function ho_ode.m to define the ODE:

```
function dy = high_order_ode_example(t,x)
% x(1) = x
% x(2) = y
% x(3) = z
dy = [-x(2) * exp(-t/5) + ...
          x(3) * exp(-t/5) + 1;
          x(3);
          -2*sin(t)]
end
```

Methods:

- Step 5: evaluate the system of equations using ODE45:

```
t0 = 5; % Start time
tf = 20; % Stop time
x0 = [1 -1 3] % Initial conditions
[t,s] = ode45(@ho_ode,[t0,tf],x0);
x = s(:,1);
y = s(:,2);
z = s(:,3);
plot(t,s);
```

Stiff ODE

The phenomenon of *stiffness* is not precisely defined in the literature. Some attempts at describing a stiff problem are:

- A problem is stiff if it contains widely varying time scales, i.e., some components of the solution decay much more rapidly than others.
- A problem is stiff if the stepsize is dictated by stability requirements rather than by accuracy requirements.
- A problem is stiff if explicit methods don't work, or work only extremely slowly.
- A linear problem is stiff if all of its eigenvalues have negative real part, and the *stiffness ratio* (the ratio of the magnitudes of the real parts of the largest and smallest eigenvalues) is large.
- More generally, a problem is stiff if the eigenvalues of the Jacobian of $\boldsymbol{f}$ differ greatly in magnitude.

In mathematics, a **stiff equation** is a differential equation for which certain numerical methods for solving the equation are numerically unstable, unless the step size is taken to be extremely small. It has proven difficult to formulate a precise definition of stiffness, but the main idea is that the equation includes some terms that can lead to rapid variation in the solution.

$$y'(t) = -15y(t), \quad t \geq 0, y(0) = 1.$$

The exact solution (shown in cyan) is

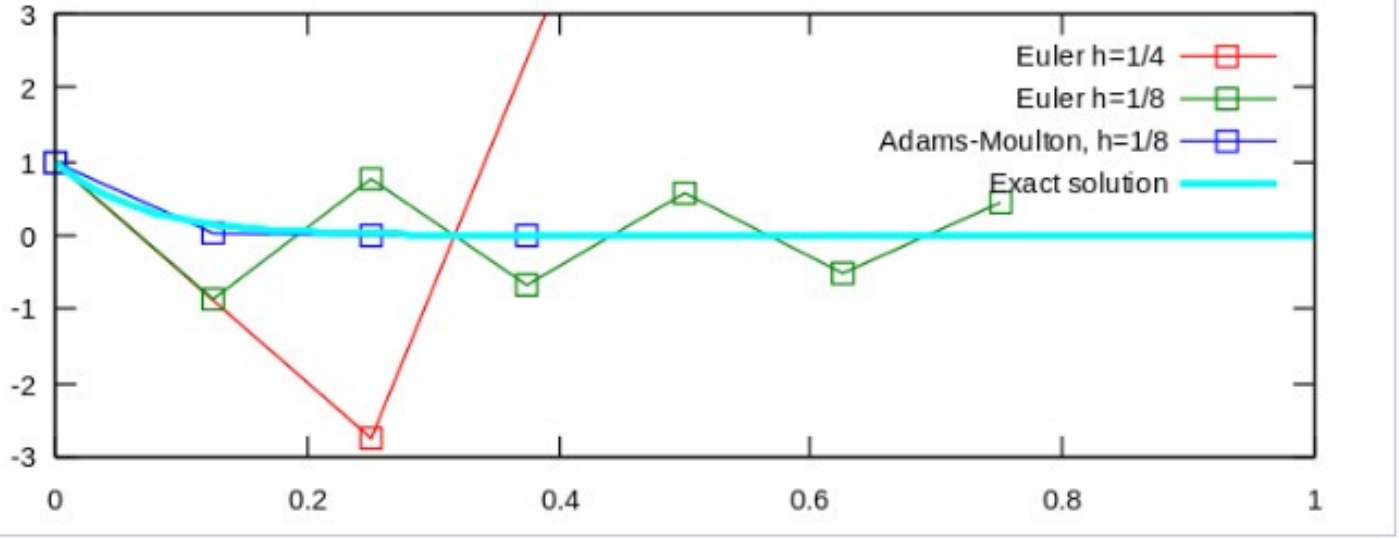
$$y(t) = e^{-15t} \text{ with } y(t) \to 0 \text{ as } t \to \infty.$$

We seek a numerical solution that exhibits the same behavior.

1. Euler's method with a step size of $h = 1/4$ oscillates wildly and quickly exits the range of the graph (shown in red).
2. Euler's method with half the step size, $h = 1/8$, produces a solution within the graph boundaries, but oscillates about zero (shown in green).
3. The trapezoidal method (that is, the two-stage Adams–Moulton method) is given by

$$y_{n+1} = y_n + \frac{1}{2}h\left(f(t_n, y_n) + f(t_{n+1}, y_{n+1})\right),$$

where $y' = f(t, y)$. Applying this method instead of Euler's method gives a much better result (blue). The numerical results decrease monotonically to zero, just as the exact solution does.



Explicit numerical methods exhibiting instability when integrating a stiff ordinary differential equation

The van der Pol equation is a second order ODE

$$y_1'' - \mu\left(1 - y_1^2\right) y_1' + y_1 = 0,$$

where $\mu > 0$ is a scalar parameter. When $\mu = 1$, the resulting system of ODEs is nonstiff and easily solved using ode45. However, if you increas μ to 1000, then the solution changes dramatically and exhibits oscillation on a much longer time scale. Approximating the solution of the initial value problem becomes more difficult. Because this particular problem is stiff, a solver intended for nonstiff problems, such as ode45, is too inefficient to be practical. Use a stiff solver such as ode15s for this problem instead.

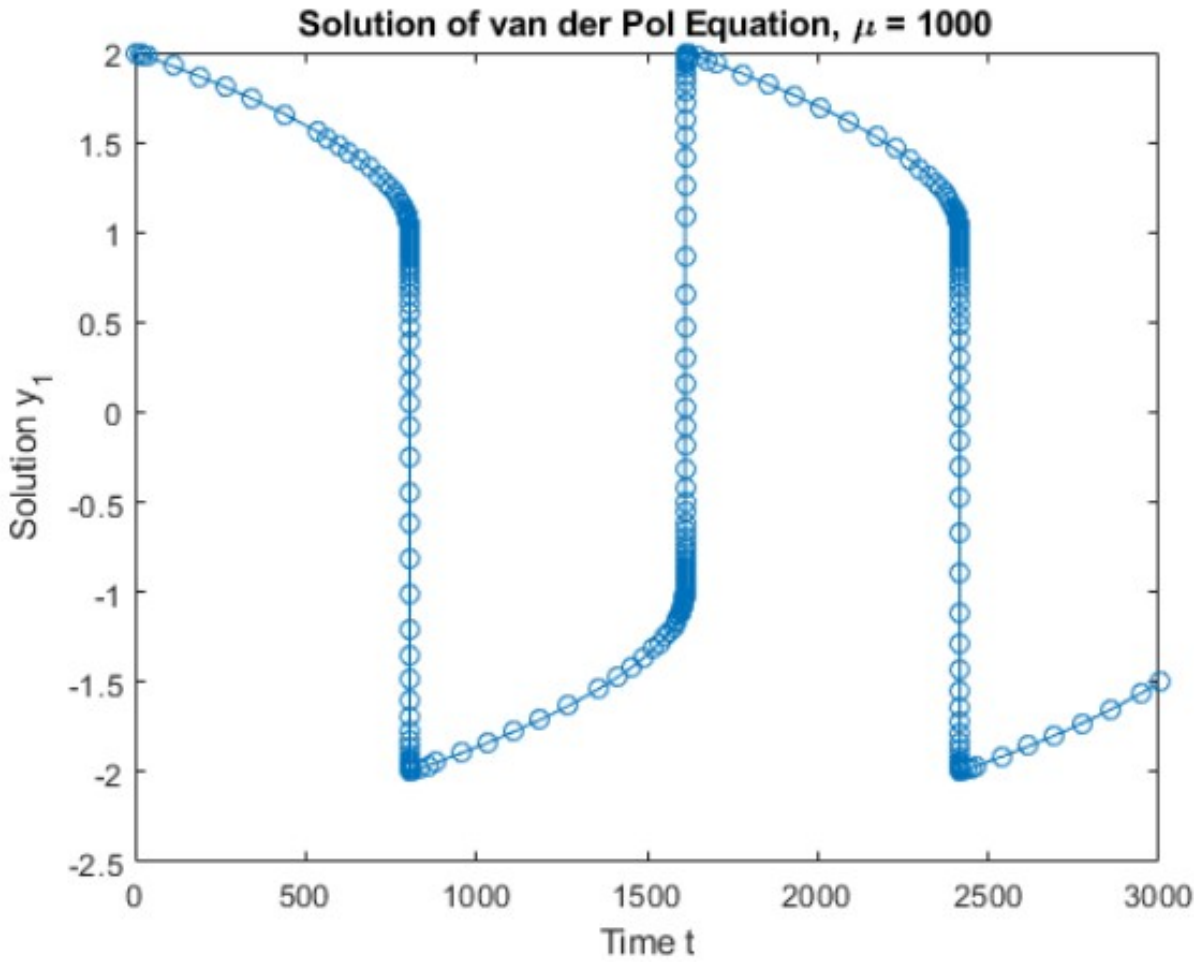
Rewrite the van der Pol equation as a system of first-order ODEs by making the substitution $y_1' = y_2$. The resulting system of first-order ODEs is

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= \mu(1 - y_1^2)y_2 - y_1. \end{aligned}$$

The vdp1000 function evaluates the van der Pol equation using $\mu = 1000$.

```
function dydt = vdp1000(t,y)
%VDP1000  Evaluate the van der Pol ODEs for mu = 1000.
%
%   See also ODE15S, ODE23S, ODE23T, ODE23TB.

%   Jacek Kierzenka and Lawrence F. Shampine
%   Copyright 1984-2014 The MathWorks, Inc.

dydt = [y(2); 1000*(1-y(1)^2)*y(2)-y(1)];
```

```
[t,y] = ode15s(@vdp1000,[0 3000],[2; 0]);
plot(t,y(:,1),'-o');
title('Solution of van der Pol Equation, \mu = 1000');
xlabel('Time t');
ylabel('Solution y_1');
```



Solve initial value problems for ordinary differential equations (ODEs)

Syntax

- `[T,Y] = solver(odefun,tspan,y0)`
- `[T,Y] = solver(odefun,tspan,y0,options)`
- `[T,Y,TE,YE,IE] = solver(odefun,tspan,y0,options)`
- `sol = solver(odefun,[t0 tf],y0...)`
-

where `solver` is one of `ode45`, `ode23`, `ode113`, `ode15s`, `ode23s`, `ode23t`, or `ode23tb`.

Solver	Problem Type	Order of Accuracy	When to Use
`ode45`	Nonstiff	Medium	Most of the time. This should be the first solver you try.
`ode23`	Nonstiff	Low	For problems with crude error tolerances or for solving moderately stiff problems.
`ode113`	Nonstiff	Low to high	For problems with stringent error tolerances or for solving computationally intensive problems.
`ode15s`	Stiff	Low to medium	If `ode45` is slow because the problem is stiff.
`ode23s`	Stiff	Low	If using crude error tolerances to solve stiff systems and the mass matrix is constant.
`ode23t`	Moderately Stiff	Low	For moderately stiff problems if you need a solution without numerical damping.
`ode23tb`	Stiff	Low	If using crude error tolerances to solve stiff systems

Ode45

ode45 is a sophisticated built-in MATLAB function that gives very accurate solutions. Ode45 is based on a simultaneous implementation of an explicit fourth and fifth order Runge-Kutta formula called the Dormand-Prince pair. It is a one-step solver. This is the first solver to be tried for most problems. It is designed for non-stiff problems. Ode45 can use long step size and so the default is to compute solution values at four points equally spaced within the span of each natural step.

The Dormand-Prince pair is an explicit method and a member of the Runge-Kutta family of solvers. The method employs function evaluations to calculate fourth and fifth order accurate solutions. The difference between these solutions is then taken to be the error of the fourth order solution.

Ode23

This solver is designed for solving non-stiff problems. It's method is based on the 2^{nd} and 3^{rd} Order Runge-Kutta pair called the Bogacki-Shampine method. Ode23 is less expensive than ode45 in that it requires less computation steps than ode45. But it is of a lower order, although it may be more efficient at crude tolerances and in the presence of mild stiffness. Ode23 is a one-step solver.

The Bogacki-Shampine method is a Runge-Kutta method of order 3 with four stages proposed by Przemyslaw Bogacki and Lawrence F. Shampine in 1989. It uses three function evaluations per step. It has embedded second order method which is used to implement adaptive step size for the method.

Ode113

Ode113 is a multi-step variable order method which uses Adams–Bashforth–Moulton predictor-correctors of order 1 to 13. It may be more efficient than ode45 at stringent tolerances and when the ODE problem is particularly expensive to evaluate. It is designed for non-stiff problems.

Ode 15s

Ode15s a variable order solver whose algorithm is based on the numerical differentiation formulas (NDFs) and optionally along with the backward differentiation formulas (BDFs) which is also called Gear's method. This is a multi-step solver that is designed for stiff problems. This is the next recommended solver if ode45 fails or is too slow.

A Summary of MATLAB ODE Solvers

Solver	Kind of Problem	Base Algorithm
Ode45	Non-stiff differential equations	Runge-Kutta
Ode23	Non-stiff differential equations	Runge-Kutta
Ode113	Non-stiff differential equations	Adams-Bashfort-Moulton
Ode15s	Stiff differential equations	Numerical Differentiation Formulas (Backward Differentiation Formulas)
Ode23s	Stiff differential equations	Rosenbrock
Ode23t	Moderately stiff differential equations	Trapezoidal Rule
Ode23tb	Stiff differential equations	TR-BDF2
Ode15i	Fully implicit differential equations	BDFs

Choosing the partition

- the solver ode45 selects a certain partition of the tspan interval, and MATLAB returns a value of y at each point in this partition.
- Sometimes, we would like to specify the partition of values on which MATLAB returns an approximation.
- For example, we may only want to approximate y(0.1),y(0.2),…,y(0.5). We can specify this by entering the vector of values [0,0.1,0.2,0.3,0.4,0.5] as the domain in ode45.
- tvalues=0:0.1:5; [x,y]=ode45(f,tvalues,1)
- It is important to point out here that MATLAB continues to use roughly the same partition of values that it originally chose.
- The only thing that has changed is the values at which it is printing a solution. In this way, no accuracy is lost.

Pass Extra Parameters to ODE Function

ode45 only works with functions that use two input arguments, t and y.

However, we can pass in extra parameters by defining them outside the function and passing them in when you specify the function handle or calling solver.

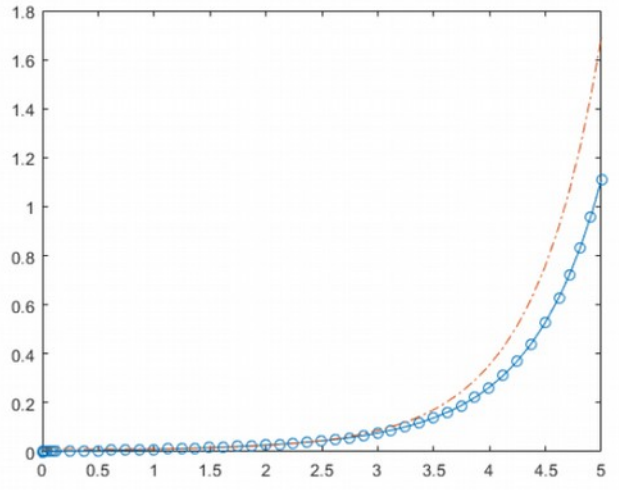
Solve the ODE

$$y'' = \frac{A}{B} t y.$$

Rewriting the equation as a first-order system yields

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= \tfrac{A}{B} t y_1. \end{aligned}$$

```
function dydt = odefcn(t,y,A,B)
dydt = zeros(2,1);
dydt(1) = y(2);
dydt(2) = (A/B)*t.*y(1);
```



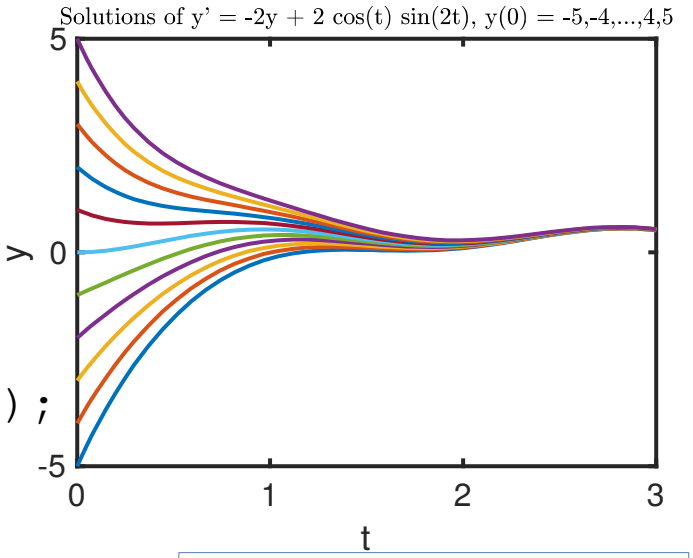
In command line

```
A = 1; B = 2;
tspan = [0 5];
y0 = [0 0.01];
[t,y] = ode45(@(t,y) odefcn(t,y,A,B), tspan, y0);
plot(t,y(:,1),'-o',t,y(:,2),'-.')
```

```
[t,y] = ode45(@odefcn, tspan, y0,[],A,B);
```

Solving single ODE with multiple initial conditions.

For example: Let us create an anonymous function to rep $f(t, y) = -2y + 2\cos(t)\sin(2t)$.




```
yprime = @(t,y) -2*y + 2*cos(t).*sin(2*t);
tspan = [0 3];
y0 = -5:5;
[t,y] = ode45(yprime,tspan,-5:5);
h=plot(t,y);
grid on;
set(h,'linewidth',3)
set(gca,'fontsize',24, 'linewidth',3)
 xlabel('t')
 ylabel('y')

title('{Solutions of y'' = -2y + 2 cos(t) sin(2t)}, {y(0) = -
5,4,...,4,5}','interpreter','latex','fontsize',18)
```

solving for multiple initial conditions at the same time is generally faster than solving the equations separately using a for-loop.

Solving Boundary value problem using Solvers in MATLAB

Edited by
Priyanka Shukla

bvp4c

R2021b

Solve boundary value problem — fourth-order method

collapse all in page

Syntax

```
sol = bvp4c(odefun,bcfun,solinit)
sol = bvp4c(odefun,bcfun,solinit,options)
```

Description

example

`sol = bvp4c(odefun,bcfun,solinit)` integrates a system of differential equations of the form $y' = f(x,y)$ specified by `odefun`, subject to the boundary conditions described by `bcfun` and the initial solution guess `solinit`. Use the `bvpinit` function to create the initial guess `solinit`, which also defines the points at which the boundary conditions in `bcfun` are enforced.

example

`sol = bvp4c(odefun,bcfun,solinit,options)` also uses the integration settings defined by `options`, which is an argument created using the `bvpset` function. For example, use the `AbsTol` and `RelTol` options to specify absolute and relative error tolerances, or the `FJacobian` option to provide the analytical partial derivatives of `odefun`.

bvpinit

Form initial guess for boundary value problem solver

Syntax

```
solinit = bvpinit(x,yinit)
solinit = bvpinit(sol,[anew bnew])
solinit = bvpinit(__,parameters)
```

Description

`solinit = bvpinit(x,yinit)` uses the initial mesh `x` and initial solution guess `yinit` to form an initial guess of the solution for a boundary value problem. You then can use the initial guess `solinit` as one of the inputs to `bvp4c` or `bvp5c` to solve the boundary value problem.

Consider the differential equation

$$y'' = -y.$$

The equation is subject to the boundary conditions

$$y(0) = y(\pi) = 0.$$

The function that encodes the equation as a first-order system i

```
function dydx = bvpfun(x,y)
dydx = [y(2)
        -y(1)];
end
```

Similarly, the function that encodes the boundary conditions is

```
function res = bcfun(ya,yb)
res = [ya(1)
       yb(1)];
end
```

You either can include the required functions as local functions at the end of a file (as done here), or you can save them as separate, named files in a directory on the MATLAB path

Initial Guess with Function Handle

You reasonably can expect the solution to the equation to be oscillatory, so sine and cosine functions are a good initial guess of the behavior of the solution and its derivative between the fixed boundary points.

```
function y = guess(x)
y = [sin(x)
     cos(x)];
end
```

Create a solution structure using 10 equally spaced mesh points in the domain $[0, \pi]$ and the initial guess function.

```
xmesh = linspace(0,pi,10);
solinit = bvpinit(xmesh,@guess);
```

Solve BVP

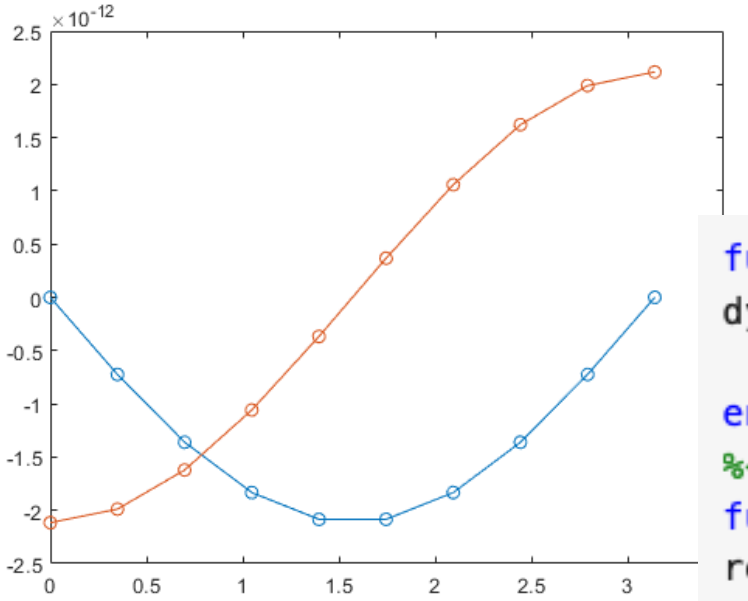
Call bvp4c with the ode function, boundary conditions, and solution guess. Plot the result.

```
sol = bvp4c(@bvpfun, @bcfun, solinit);
plot(sol.x,sol.y,'-o')
```

Solve BVP

Call bvp4c with the ode function, boundary conditions, and solution guess. Plot the result.

```
sol = bvp4c(@bvpfun, @bcfun, solinit);
plot(sol.x,sol.y,'-o')
```




```
function dydx = bvpfun(x,y) % equation being solved
dydx = [y(2)
        -y(1)];
end
%-------------------------------------------
function res = bcfun(ya,yb) % boundary conditions
res = [ya(1)
       yb(1)];
end
%-------------------------------------------
function y = guess(x) % guess at solution behavior
y = [sin(x)
     cos(x)];
end
%-------------------------------------------
```

Finite Difference Methods for the Solution of BVPs

Edited by
Priyanka Shukla

The solution of BVPs by a one dimensional finite difference (FD) method can be accomplished by the following steps (the equilibrium or replacement method):

a. Discretize the solution domain into a one dimensional finite difference grid,
b. Approximate the derivatives in the ODE by rational algebraic approximations,
c. Substitute the approximations into the ODE,
d. Solve the resulting system of algebraic FD Equations.

In this process, a system of coupled FD equations must be solved simultaneously.

Forward difference:

$$f_i' = (f_{i+1} - f_i)/h$$

Backward difference:

$$f_i' = (f_i - f_{i-1})/h$$

Central difference:

$$f_i' = (f_{i+1} - f_{i-1})/2h$$

Boundary condition, starting point derivative for central difference:

$$\frac{dy(0)}{dx} \rightarrow \frac{y_1 - y_{-1}}{2h}$$

Boundary condition, ending point derivative for central difference:

$$\frac{dy}{dx}(n) \rightarrow \frac{y_{n+1} - y_{n-1}}{2h}$$

$$f_i'' = \frac{f_{i+1} - 2f_i + f_{i-1}}{h^2}$$

Use finite difference method to solve the linear, autonomous BVP,

$$y'' + 1 = 0, \quad y(0) = y(1) = 0$$

for $h = 0.2$ and $h = 0.02$. Plot the solution curve for $h = 0.02$.

For $h = 0.2$,

$$\frac{y_{i+1} + y_{i-1} - 2y_i}{h^2} + 1 = 0$$

$$y_{i+1} - 2y_i + y_{i-1} = -h^2$$

$$i = 1, \quad y_0 - 2y_1 + y_2 = -h^2$$

$$i = 2, \quad y_1 - 2y_2 + y_3 = -h^2$$

$$i = 3, \quad y_2 - 2y_3 + y_4 = -h^2$$

$$i = 4, \quad y_3 - 2y_4 + y_5 = -h^2$$

$$y_0 = y_5 = 0$$

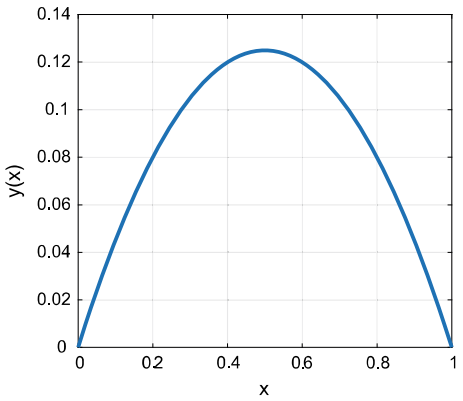
Let,

$$\boldsymbol{Ay} = \boldsymbol{b}$$

$$\begin{bmatrix} -2 & 1 & 0 & 0 \\ 1 & -2 & 1 & 0 \\ 0 & 1 & -2 & 1 \\ 0 & 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} -h^2 \\ -h^2 \\ -h^2 \\ -h^2 \end{bmatrix}$$

Solution of this equation gives

$$y_1 = 0.08, \quad y_2 = 0.12, \quad y_3 = 0.12, \quad y_4 = 0.08$$



```
%general case
h=0.02; H=-h*h; N=1/h;
A=zeros(N-1);
for i=1:N-2
    A(i,i+1)=1; B=A';
end
A=A+B;
for i=1:N-1
    A(i,i)=-2;
end
A; b=H*ones(N-1,1); y=A\b ; Y=zeros(N+1,1);
i=2:N;
Y(i)=y(i-1)
x=linspace(0,1,N+1);
plot(x,Y,'linewidth',2);grid;xlabel('x');ylabel('y(x)');


% dsolve_fin_diff2.m   Solves the  BVP y"+1=0 y(0)=0,y(1)=0
clc;clear; syms  y(x)
bc='y(0) ==0,y(1)==0';
S=dsolve(diff(diff(y))+1 == 0,bc,x)
x = linspace(0,1,50); z = eval(vectorize(S));
plot(x,z,'linewidth',2);grid;xlabel('x'); ylabel('y(x)');
```

Solve the autonomous BVP, $y'' + y = 0, \quad y(0) = 0, \quad y(1) = 1,$

$$\frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} + y_i = 0$$

$$y_{i-1} + \left(h^2 - 2\right)y_i + y_{i+1} = 0$$

$$h^2 - 2 = (0.2)^2 - 2 = -1.96$$

$$i = 1 \quad y_0 - 1.96y_1 + y_2 = 0$$

$$i = 2 \quad y_1 - 1.96y_2 + y_3 = 0$$

$$i = 3 \quad y_2 - 1.96y_3 + y_4 = 0$$

$$i = 4 \quad y_3 - 1.96y_4 + y_5 = 0$$

$$y(0) = y_0 = 0, \quad y(5) = y_5 = 1$$

$$-1.96y_1 + y_2 = 0$$

$$y_1 - 1.96y_2 + y_3 = 0$$

$$y_2 - 1.96y_3 + y_4 = 0$$

$$y_3 - 1.96y_4 + 1 = 0$$

$$\boldsymbol{Ay} = \boldsymbol{b}$$

$$\begin{bmatrix} -1.96 & 1 & 0 & 0 \\ 1 & -1.96 & 1 & 0 \\ 0 & 1 & -1.96 & 1 \\ 0 & 0 & 1 & -1.96 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ -1 \end{bmatrix}$$

Solution of this equation yields

$$y_1 = 0.2362, \quad y_2 = 0.463, \quad y_3 = 0.6713, \quad y_4 = 0.8527.$$

```
%fin_diff_bvp1.m
clc;clear;close;

% A=[-1.96 1 0 0;
% 1 -1.96 1 0;
% 0 1 -1.96 1;
% 0 0 1 -1.96];
% b=[0 0 0 -1]';
% y=A\b

%general case
h=0.02; N=1/h;
A=zeros(N-1);
for i=1:N-2
    A(i,i+1)=1; B=A';
end
A=A+B;
for i=1:N-1;A(i,i)=h^2-2;end
A;
b=zeros(N-1,1); b(N-1,1)=-1;
y=A\b;
Y=zeros(N+1,1); Y(1)=0;Y(N+1)=1;
i=2:N;
Y(i)=y(i-1);
x=linspace(0,1,N+1);
plot(x,Y,'linewidth',2);grid;xlabel('x');ylabel('y(x)');

% dsolve_fin_diff1.m   SOLVE  BVP y"+y=0 y(0)=0,y(1)=1
clc;clear; syms  y(x)
bc='y(0) ==0,y(1)==1';
S=dsolve(diff(diff(y))+y == 0, bc)
x = linspace(0,1,50);z = eval(vectorize(S));
plot(x,z,'k','linewidth',2);grid; xlabel('x'); ylabel('y(x)');
```