Product Support

1510 - Differential Equations in MATLAB

Section 1: What Equations Can MATLAB Handle?

MATLAB provides functions for solving several classes of problems involving differential equations:

1. Initial Value Problems for Ordinary Differential Equations (ODEs)
   This is the most popular type of problems solved using MATLAB ODE solvers. Initial value problems are typically solved with ODE45 for the nonstiff case, and with ODE15S in the stiff case. (For an explanation of Stiffness, refer to the section "What is Stiffness".)
2. Initial Value Problems for Differential Algebraic Equations (DAEs)
   These are frequently encountered in areas where conservation laws dictate a constant relationship between some variables. MATLAB can solve DAE's of index 1 using ODE15S or ODE23T. (For an explanation of index, refer to the section **DAEs and their index**.)
3. Boundary Value Problems (BVPs)
   These consist of differential equations with conditions specified on both sides. While not encountered as frequently as IVP's, these are still a common problem in engineering applications. They can be solved using the function BVP4C.
4. Delay Differential Equations (DDEs)
   These differential equations involve delays in the independent variable. They are frequently encountered in a variety of applications such as biological and chemical modeling, and can be solved using the function DDE23.
5. Partial Differential Equations (PDEs)
   Initial-boundary-value problems for systems of parabolic and elliptic differential equations in one spatial dimension and time can be solved using PDEPE. The PDE Toolbox is available for those interested in solving more general classes of PDEs.

For more information regarding general integration techniques using MATLAB, see **Solution 8314** .
For more information on the algorithms for the various solvers in MATLAB, see the following URLs:

- ODE functionality
- BVP functionality
- DDE functionality
- PDE functionallity

Section 2: Where Can I Find Tutorials or Additional Information?

A series of papers and tutorials available on **MATLAB Central**, our newsgroup and file exchange site, further explain the algorithms and usage of the MATLAB solvers for each type of equations (ODE,DAE,BVP,DDE). The papers and tutorials come with a variety of examples for you to download for each type of equations.

The tutorials are found under the category **Mathematics | Differential Equations**.

Chapter 7 of *Numerical Computing with MATLAB*, by Cleve Moler, discusses ODEs in additional detail and includes many examples and sample problems.

Section 3: What Changes in Syntax Exist for ODE Solvers?

The preferred syntax when using ODE solvers in MATLAB 6.5 (R13) is

```
[t,y] = odesolver(odefun,tspan,y0,options,  parameter1,parameter2, ... ,
                                 parameterN);
```

odesolver is the solver you are using, such as ODE45 or ODE15S. odefun is the function that defines the derivatives, so odefun defines y' as a function of the independent parameter (typically time t) as well as y and other parameters. In MATLAB 6.5 (R13), it is preferred to have odefun in the form of a function handle.

For example, ode45(@xdot,tspan,y0), rather than ode45('xdot',tspan,y0).

See the **documentation** for the benefits of using function handles.

Use function handles to pass any function that defines quantities the MATLAB solver will compute, for example, a mass matrix or Jacobian pattern.

If you prefer using strings to pass your function, the MATLAB ODE solvers are backward compatible.

In older versions of MATLAB, flags were passed to the function to find the state and pass the appropriate computation to the solver. In MATLAB 6.0 (R12) and later releases, this is no longer necessary. This difference is documented **here**.

If you are using an older syntax of the MATLAB ODE solvers, you can see older examples using the various solvers on our FTP site:

ftp://ftp.mathworks.com/pub/doc/papers/dae/

The preceding site contains three directories for BVP, DAE, and DDE examples in the older syntax. You can find examples using ODE45 and ODE23 at the following site:

ftp://ftp.mathworks.com/pub/mathworks/toolbox/matlab/funfun/

You can view the updated versions of theses examples at the **MATLAB Central** file exchange site.

Section 4: How Do I Reduce the Order of an ODE?

The code for a first-order ODE is very straightforward. However, a second- or third-order ODE cannot be directly used. You must first rewrite the higher order ODE as a system of first-order ODEs that can be solved with the MATLAB ODE solvers.

This is an example of how to reduce a second-order differential equation into two first-order equations for use with MATLAB ODE solvers such as ODE45. The following system of equations consists of one first- and one second-order differential equations:

```
x' = -y * exp(-t/5) + y' * exp(-t/5) + 1    Equation (1)
y''= -2*sin(t)                              Equation (2)
```

The first step is to introduce a new variable that equals the first derivative of the free variable in the second order equation:

```
z = y'                                      Equation (3)
```

Taking the derivative of each side yields the following:

```
z' = y''                                    Equation (4)
```

Substituting (4) into (2) produces the following:

```
z' = -2*sin(t)                              Equation (5)
```

Combining (1), (3), and (5) yields three first-order differential equations.

```
x' = -y * exp(-t/5) + y' * exp(-t/5) + 1;   Equation (1)
z  = y'                                     Equation (3)
z' = -2*sin(t)                              Equation (5)
```

Since `z = y'`, substitute `z` for `y'` in equation (1). Also, since MATLAB requires that all derivatives be on the left-hand side, rewrite equation (3). This produces the following set of equations:

```
x' = -y * exp(-t/5) + z * exp(-t/5) + 1     Equation (1a)
y' = z                                      Equation (6a)
z' = -2*sin(t)                              Equation (5a)
```

To evaluate this system of equations using ODE45 or another MATLAB ODE solver, create a function that contains these differential equations. The function requires two inputs, the states and time, and returns the state derivatives.

Following is the function named `odetest.m`:

```
function xprime = odetest(t,x)
% Since the states are passed in as a single vector, let
% x(1) = x
% x(2) = y
% x(3) = z


xprime(1) = -x(2) * exp(-t/5) + x(3) * exp(-t/5) + 1;
% x' = -y*exp(-t/5 + z*exp(-t/5) + 1


xprime(2) = x(3);
% y' = z


xprime(3) = -2*sin(t);
% z' = -2*sin(t)


xprime = xprime(:) ;
% This ensures that the vector returned is a column vector
```

To evaluate the system of equations using ODE23 or another MATLAB ODE solver, define the start and stop times and the initial conditions of the state vector. For example,

```
t0 = 5; % Start time
tf = 20; % Stop time
x0 = [1 -1 3] % Initial conditions
[t,s] = ode23(@odetest,[t0,tf],x0);
x = s(:,1);
y = s(:,2);
z = s(:,3);
```

Section 5: How Do I Solve Time-Dependent ODEs?

Following is an example of an ordinary differential equation that has a time-dependent term using a MATLAB ODE solver. The time-dependent term can be defined either by a data set with known sample times or as a simple function. If the time-dependent term is defined by a data set, the data set and its sample times are passed in to the function called by the ODE solver as additional parameters. If the time-dependent term is defined by a function, then that function is called in the derivative function as needed.

The differential equation used in this example is the Damped Wave Equation with a sinusoidal driving term.

```
y''(t) - beta * y'(t) + omega^2 * y(t) =
A * sin(w0 * t - theta)
```

MATLAB requires that the differential equation be expressed as a first-order differential equation using the following form:

```
y'(t) = B * y(t) + f(t)
```

where y is a column vector of states and B is a matrix. The MATLAB definition of this differential equation, using the techniques of the previous section, is as follows:

```
xdot(2) = beta * x(2) - omega^2 * x(1) + ...
A * sin(w0 * t - theta)
xdot(1) = x(2)
```

where

```
xdot = dx/dt
x(1) = y
x(2) = dy/dt
```

In this example beta, omega, A, w0, and theta need to be defined. They are passed as additional parameters for the MATLAB ODE solver.

Example 1: Time-dependent term is a function.
Create the following derivative function:

```
% FUN1.M: Time-dependent ODE example
function xdot = fun1(t,x,beta,omega,A,w0,theta)
% The time-dependent term is A * sin(w0 * t - theta)
xdot(2)= -beta*x(2) + omega^2 * x(1) + ...
A * sin(w0 * t - theta);
xdot(1) = x(2);
xdot=xdot(:);
% To make xdot a column
% End of FUN1.M
```

To call this function in MATLAB, use the following:

```
beta = .1;
omega = 2;
A = .1;
w0 = 1.3;
theta = pi/4;
X0 = [0 1]';
t0 = 0;
tf = 20;
options = [];
[t,y]=ode23(@fun1,[t0,tf],X0,options,beta,omega,A,w0,theta);
plot(t,y)
```

Example 2: The time-dependent term is defined by a data set.
The following example requires the **INTERP1** command. Create the following derivative function:

```
% FUN2.M Time-dependent ODE example with data set
function xdot = fun2(t,x,beta,omega,T,P)
pt=interp1(T,P,t);
xdot(2) = beta*x(2)-omega^2*x(1)+pt;
xdot(1) = x(2);
xdot=xdot(:); % To make xdot a column
% End of FUN2.M
```

To call this function in MATLAB, use the following:

```
beta = .1;
omega = 2;
A = .1;
w0 = 1.3;
theta = pi/4;
X0 = [0 1]';
t0 = 0;
tf = 20;
```

```
T=t0-eps:.1:tf+theta+eps;
P=A.*sin(w0.*T-theta);
[t,y]=ode23(@fun2,[t0,tf],X0,[],beta,omega,T,P);
plot(t,y(:,1))
```

The calls to `INTERP1` might cause the second example to be slower than the first.

Section 6: How Do I Use a Fixed Time Step?

The ordinary differential equation solver functions provided with MATLAB employ a variety of methods. ODE23 is based on the Runge Kutta (2,3)integration method, and ODE45 is based on the Runge Kutta (4,5) integration method. ODE113 is a variable-order Adams-Bashforth-Moulton PECE solver. For a complete listing of the various solvers and their methods, see the **documentation**.

The MATLAB ODE solvers utilize these methods by taking a step, estimating the error at this step, checking to see if the value is greater than or less than the tolerance, and altering the step size accordingly. These integration methods do not lend themselves to a fixed step size. Using an algorithm that uses a fixed step size is dangerous since you can miss points where your signal frequency is greater than the solver frequency. Using a variable step ensures that a large step size is used for low frequencies and a small step size is used for high frequencies. The ODE solvers within MATLAB are optimized for a variable step, run faster with a variable step size, and clearly the results are more accurate. There are now fixed time step solvers available on **our FTP site**. These solvers are

ODE1	A first-order Euler method
ODE2	A second-order Euler method
ODE3	A third-order Runge-Kutta method
ODE4	A fourth-order Runge-Kutta method
ODE5	A fifth-order Runge-Kutta method

These solvers can be used with the following syntax:

```
y = ode4(odefun,tspan,y0);
```

The integration proceeds by steps, taken to the values specified in `tspan`. The time values must be in order, either increasing or decreasing. Note that the step size (the distance between consecutive elements of `tspan`) does not have to be uniform. If the step size is uniform, you might want to use **LINSPACE**.

For example,

```
tspan = linspace(t0,tf,nsteps); % t0 = 0; tf = 10, nsteps = 100;
```

Section 7: How Do I Use Stochastic Differential Equations?

A stochastic differential equation is a differential equation with an element of randomness in the equation. A stochastic differential equation is typically written as

```
dX = lambda*X dt + mu*X dW
```

Where `X` is the variable of interest, `t` is time, and `W` is a random variable or process. `lambda` and `mu` are constant parameters of the problem.

A comprehensive introduction to solving SDEs numerically is found in the paper **"An Algorithmic Introduction to Numerical Simulation of Stochastic Differential Equations"**, by Desmond J Higham (SIAM review, Volume 43, Number 3). This paper also has several links to MATLAB examples which help illustrate the paper's points.

You can find the examples listed in the above paper, as well as additional examples in the areas of finance, at the following URL:

http://www.maths.strath.ac.uk/~aas96106/algfiles.html

Section 8: Systems of Equations

I. A system of ordinary differential equations contains differential equations that depend on other equations. For example,

$$\frac{d}{dx}u(x) = u'(x) = u(x) + v(x)$$

$$\frac{d}{dx}v(x) = v'(x) = u(x)$$

This example is a rather simplified example that can be solved either analytically or numerically. In general, analytical techniques might not be available for many systems. For a linear system, put these equations in a matrix form. For example,

$$\frac{d}{dx}\begin{pmatrix}u\\v\end{pmatrix}=\begin{pmatrix}1&1\\1&0\end{pmatrix}\otimes\begin{pmatrix}u\\v\end{pmatrix}\quad\text{or}\quad\frac{d\mathbf{y}}{dx}=\mathrm{A}\otimes\mathbf{y},\quad\text{with }\mathbf{y}=\begin{pmatrix}u\\v\end{pmatrix},\quad\text{and }\mathbf{A}=\begin{pmatrix}1&1\\1&0\end{pmatrix}$$

See "Introduction to Linear Algebra and Its Applications" by Gilbert Strang for additional details and considerations.

This technical note focuses on numerical solutions of ordinary differential equations. To solve the above system of equations numerically, create a function that defines the rate of change of the vector `y`.

```
function dy = exampleode(t,y)
% function to be integrated
dy = zeros(2,1);
dy(1) = y(1) + y(2);
dy(2) = y(1); % Alternatively
% A = [1 1; 1 0];
% dy = A*y
```

Call this function using a numerical solver provided in MATLAB. Start with ODE45:

```
xspan = [0 10];
ynot = [1 0];
[X,Y] = ode45(@exampleode,xspan,ynot);
```

This creates a time vector `X` (or whatever `X` represents) and a corresponding `Y` vector, which is simply `Y` at times `X`. In the above example, the first column of `Y` is `u` and the second column is `v`.

II. Consider the second-order system

$$u''(x)+e^{x}v'(x)+3u(x)=e^{2x}$$
$$v''(x)+\cos(x)u'(x)+u(x)=\sin(x)$$

First reduce this system of second-order ODEs to a first-order differential equation by introducing the vector

$$\mathbf{y}=\begin{pmatrix}u\\u'\\v\\v'\end{pmatrix}$$

Next, rewrite the above system of equations as

$$\frac{d}{dx}\begin{pmatrix}u\\u'\\v\\v'\end{pmatrix}=\begin{pmatrix}0&1&0&0\\-3&0&0&-e^{x}\\0&0&0&1\\-1&-\cos(x)&0&0\end{pmatrix}\otimes\begin{pmatrix}u\\u'\\v\\v'\end{pmatrix}+\begin{pmatrix}0\\e^{2x}\\0\\\sin(x)\end{pmatrix}$$

Enter this into MATLAB in the following format:

```
function dy = secondode(x,y)
% function to be integrated
dy = zeros(4,1);


dy(1) = y(2);
dy(2) = -3*y(1) -exp(x)*y(4) + exp(2*x);
dy(3) = y(4);
dy(4) = -y(1) -cos(x)*y(2) + sin(x);
```

Note the change of variable from `x` to `t` (it is simply the independent variable).

Now solve the system using ODE45 and the initial conditions `u(0) = 1, u'(0) = 2, v(0) = 3, v'(0) = 4` over the interval from `x = 0` to `x = 3`. The commands you will need to use are:

```
xspan = [0 3];
y0 = [1; 2; 3; 4];
[x, y] = ode45(@secondode, xspan, y0);
```

The values in the first column of y correspond to the values of u for the x values in x. The values in the second column of y correspond to the values of u', and so on.

III. Consider the following system of equations:

$$\mathbf{A}\otimes\mathbf{y}''+\mathbf{B}\otimes\mathbf{y}'+\mathbf{C}\otimes\mathbf{y}=\mathbf{D} \qquad (1)$$

Where `A`, `B`, `C`, and `D` are matrices and `y` is a vector. For example,

$$\mathbf{y}=\begin{pmatrix} x_1(t) \\ x_2(t) \end{pmatrix}$$

You can reduce the order of this equation and solve it numerically. Begin by defining

$$\mathbf{u}_1(t)=\mathbf{y}(t)$$
$$\mathbf{u}_2(t)=\mathbf{y}'(t)$$

This allows (1) to be rewritten as

$$\mathbf{u}'=\frac{d}{dt}\begin{pmatrix} u_1 \\ u_2 \end{pmatrix}=\begin{pmatrix} 0 & I \\ -A^{-1}\otimes\mathbf{C} & -A^{-1}\otimes\mathbf{B} \end{pmatrix}\otimes\begin{pmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \end{pmatrix}+\begin{pmatrix} \mathbf{0} \\ A^{-1}\otimes\mathbf{D} \end{pmatrix}=\frac{d}{dt}\begin{pmatrix} x_1 \\ x_2 \\ x'_1 \\ x'_2 \end{pmatrix}$$

Note: For implicit solvers, such as ODE15S, ODE23T and ODE23TB, you can use `A` as a mass matrix, which is done frequently with Differential Algebraic Equations (DAE's).

You could write $-A^{-1}\otimes C$ as $-A\backslash C$ etc.

You can then use this in conjunction with ODE45 or another ODE solver to obtain a numerical solution. For example, with the matrices `A`, `B`, `C`, and `D` as

$$\mathbf{A}=\begin{pmatrix} 0 & 2 \\ -1 & 1 \end{pmatrix}, \qquad B=\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}, \qquad C=\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \qquad D=\begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

You can use ODE45 to solve this system as

```
function dy = matrixode(t,y)
% function to be integrated
dy = zeros(4,1);
dy(1) = y(3);
dy(2) = y(4);
dy(3) = -0.5*y(1) - y(1) + 0.5*y(3) + y(4);
dy(4) = -0.5*y(1) + 0.5*y(3) + 1;
```

With initial conditions x1(0) = 9, x2(0) = 7, x1'(0) = 5, x2'(0) = 3 and a time span of t = 0 to t = 5, the commands to solve this system are:

```
tspan = [0 5];
x_init = [9; 7; 5; 3];
[t, x] = ode45(@matrixode, tspan, x_init);
```

To plot the values of `x2(t)` in red and `x2'(t)` in green versus time, use this command:

```
plot(t, x(:,2), 'r-', t, x(:,4), 'g-')
```

Section 9: Boundary Value Problem (BVP): Channel Flow

I. The Navier-Stokes equation for an incompressible fluid is

$$\frac{\;}{\partial t} + v \bullet \nabla v = -\frac{\;}{\rho}\nabla p + \mu\nabla^2 v$$

where v is velocity, þ is the density, and μ is the kinematic viscosity. The left-hand side is the total derivative or material derivative. Assuming a steady state, the left-hand side becomes zero, and you are left with a much simpler equation:

$$\frac{1}{\rho}\nabla p = \mu\nabla^2 v$$

Assuming that the flow is fully developed in the x-direction because of a pressure difference between inlet and outlet, you now have the equation

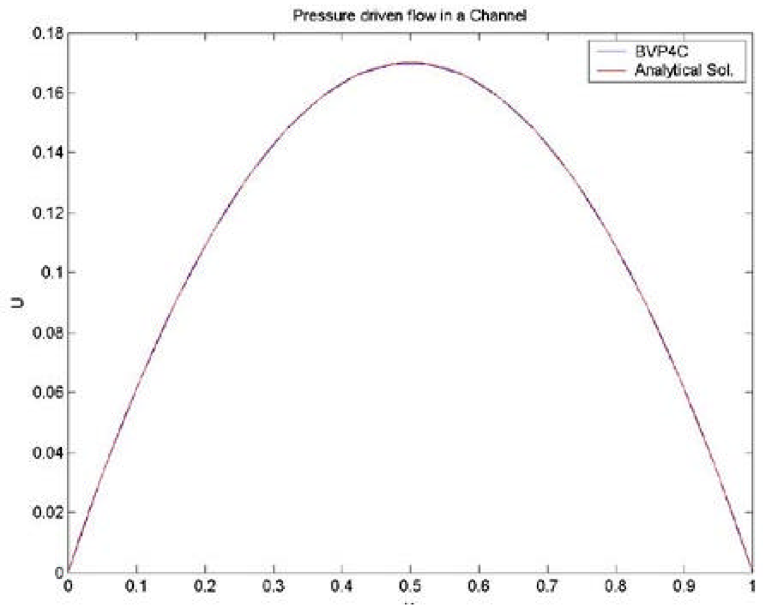
$$\frac{\partial^2 v}{\partial y^2} = \frac{1}{\eta L}\left(p_1 - p_2\right)$$

where η is the absolute viscosity and L is the length between the pressures þ$_1$ and þ$_2$. This equation can be solved using the function BVP4C, shown in the following code:

```
function NavStokes(n)
%  Illustration of an application of BVP4C
%  This example takes the case of a steady-state fluid, with only a
%  pressure gradient and flow in the x-direction, therefore the
%  Navier-Stokes equation, which is


%  dv/dt + v * grad(v) = -grad(P) + laplacian(v)


%  becomes, with u as the x-fluid speed


%  d^2u/dy^2 = (p1-p2)/L


%  The BCs are u(0)=u(1)=0, no-slip condition.
%  Where L is the length between the pressure gradients


% n is the number of points used in the mesh


if nargin~=1 || ~isnumeric(n)
  solinit = bvpinit(linspace(0,1,50),@ex1init);
  % initializing the mesh
else
  solinit = bvpinit(linspace(0,1,n),@ex1init);
  % initializing the mesh
end


options = bvpset('Stats','on','RelTol',1e-5);
sol = bvp4c(@f,@ex1bc,solinit,options);


% The solution at the mesh points
x = sol.x;
y = sol.y;


figure;
plot(x,y(1,:)')
hold on
a = linspace(0,1,50);
analytical = (2-1)/0.735*0.5*a.*(1-a);
% The analytical solution to the problem is (p1-p2)/L*0.5*y*(1-y)
plot(a,analytical,'-r')
```

```
legend('BVP4C','Analytical Sol.');
xlabel('y');
ylabel('U');>
title('Pressure-driven flow in a Channel');
%----------------------------------------------------------------


function dudy = f(y,u)
dudy = zeros(2,1);
p1 = 2; p2 = 1; L = 0.735;


press_grad = (p1-p2)/L;


dudy(1) = u(2);
dudy(2) = -press_grad;


%----------------------------------------------------------------


function res = ex1bc(ua,ub)
res = [ua(1) - 0; ub(1) - 0];


%----------------------------------------------------------------


function v = ex1init(u)
v = [0 0];
```

A plot comparing the computed velocity profile with the analytical velocity profile is below. You can adapt this code as needed for a different problem.



Section 10: What Is Stiffness?

A stiff ODE is an ordinary differential equation that has a transient region whose behavior is on a different scale from that outside this transient region. A physical example of a stiff system involves chemical reaction rates, where typically the convergence to a final solution can be quite rapid.

An important characteristic of a stiff system is that the equations are always stable, meaning that they converge to a solution. The following example clarifies this characteristic:

```
y'(x) = J(x)y(x) + p'(x)
```

The analytical solution is

```
y(x) = (A - p(0))*exp(J(x)) + p(x)
```

For any negative |J(x)|, the solutions y and p will eventually converge, demonstrating that the equation is stable. If |J(x)| is large, the convergence will be quite rapid. During this transient period, a small step size might be necessary for numerical stability. Using the following stiff system,

$\begin{pmatrix} u'(x) \\ v'(x) \end{pmatrix} = \begin{pmatrix} 998 & 1998 \\ -999 & -1999 \end{pmatrix} \otimes \begin{pmatrix} u(x) \\ v(x) \end{pmatrix}$ with the initial condition $\begin{pmatrix} u(0) \\ v(0) \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$,

the analytical solution to this system is

$$u(x) = 2e^{-x} - e^{-1000x}$$
$$v(x) = -e^{-x} + e^{-1000x}$$

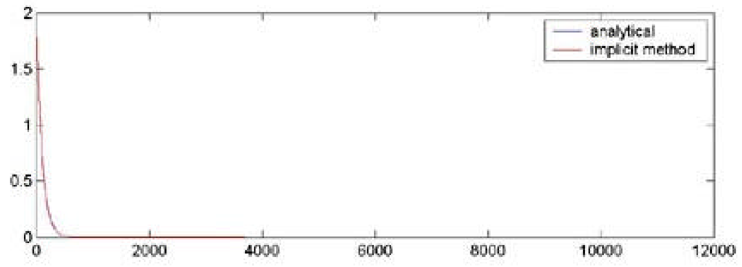
The solutions u and v have a second term that has a negligible effect on the solution for x greater than zero but can restrict the step size of a numerical solver. You must usually reduce this step size to limit the numerical instability.

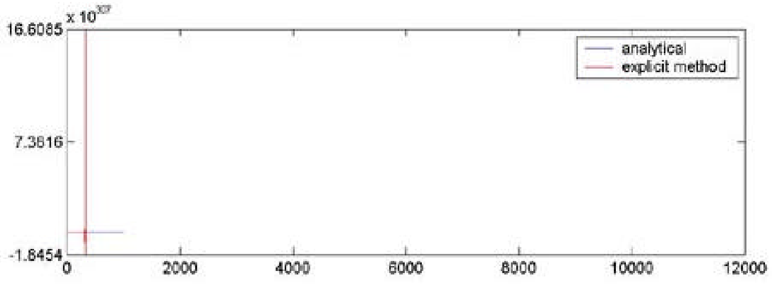
Section 11: Implicit vs. Explicit Methods

The following code shows an implicit numerical approach and an explicit numerical approach. Implicit methods are frequently called backward step solvers, whereas an explicit method is regarded as a forward step solver.

```
% Example of stiff system solved by first-order implicit system
C = [ 998, 1998; -999, -1999]; % y' = C*y
% implicitly as Y(n+1) = Y(n) + h*Y'(n+1)
ynot = [1; 0.0];
h = 10/max(abs(eig(C)));
% solving the implicit system Y(n+1) = Y(n) + h*C*Y(n+1)
% gives, Y(n+1)(1-h*C) = Y(n), or Y(n+1) = inv(eye(2)-h*C)*Y(n)
% defining Delta = inv(eye(2)-h*C)
Delta = inv(eye(2)-h*C);
Y(1:2,1) = ynot;
% to find Y over T = [0 10],
% tic
for i = 1:10000, Y(1:2,i+1)=Delta*Y(1:2,i); end
% toc


% Next, use an explicit approach
% y(n+1) = y(n) + h*y'(n)
% giving y(n+1) = y(n) + h*C*y(n), or y(n+1) = (1+h*C)*y(n)
Delta2 = (eye(2)+h*C);
Ye(1:2,1) =ynot;
for i = 1:10000, Ye(1:2,i+1)=Delta2*Ye(1:2,i);
end


% Giving the explicit solution


% Next, obtain the analytical solution
t = linspace(0,10,1000);
u = 2*exp(-t)-exp(-1000*t);
v = -exp(-t) + exp(-1000*t);
Ya = [u;v];


% comparing
subplot(2,1,1)
plot(Ya(1,:),'-b');
hold on
plot(Y(1,:),'-r');
legend('analytical','implicit method');
subplot(2,1,2);
plot(Ya(1,:),'-b');
```

```
hold on
plot(Ye(1,:),'-r');
legend('analytical','explicit method');
```





You can see that the explicit method requires a smaller step size for numerical stability as compared to the implicit method. In fact, the first five values of the explicit method are 1, 10.98, -79.0398, 730.9406, and -6.5591e+003.

If you edit the above code so that the step size, `h`, is 0.001 (reducing `h` by a factor of 10) then both the explicit method and implicit methods will agree with the analytical solution. This is a simplified example, but does illustrate the basic idea of a stiff solver, using an inherently numerically stable method.

Looking at the analytical solution, you should be more concerned about using a larger step size while retaining numerical stability for an efficient integration. A nonstiff solver, by contrast, is primarily concerned with accuracy. When encountering a stiff problem, a nonstiff solver will reduce its step size accordingly (making it much more inefficient). Remember that a stiff system is stable (it converges to a solution), justifying numerical stability as the highest priority rather than pure numerical accuracy.

Identifying a stiff system is one of the more important steps in the process of numerical integration. As noted above, a nonstiff solver is much less efficient than a stiff solver. While not rules, the following tips might help you in identifying a stiff system.

1. If the eigenvalues are obtainable, or available, a measure of stiffness can be calculated. This stiffness ratio is the ratio of the eigenvalue with the largest magnitude to the eigenvalue with the smallest magnitude. In the stiff system above, the eigenvalues are -1 and -1000, giving a stiffness ratio of 1000.
2. If the region of integration is on a region with no transient, the equation is not stiff. A stiff equation must have a transient. In the system above, this transient is close to the origin. This transient dies out quickly, giving two time scales over the time of integration.
3. Understanding what you are modeling is a great advantage when you are choosing a solver, esepcially in this instance. If you expect behaviors on different scales, you might want to choose a stiff solver.
4. Finally, and somewhat unfortunately, you might want to choose a stiff solver if you have tried a nonstiff solver and found it to be very inefficient and time-consuming.

Section 12: Examples

Following isd an example showing the differences in efficiency of a nonstiff solver and a stiff solver. The above system is solved using the solvers ODE45 and ODE23S. The system can be defined in a function, such as

```
function dy = stiffode(t,y)


dy = zeros(2,1);
dy(1) = 998*y(1) + 1998*y(2);
dy(2) = -999*y(1) -1999*y(2);
```

Upon execution with the following interval and initial condition,

```
ynot = [1 0];
tspan = [0 20];
[T, Y] = ode45(@stiffode,tspan,ynot);
length(Y) % Executed to find the number of steps.
```

```
ans=        24149
```

```
[T, Y] = ode15s(@stiffode,tspan,ynot);
length(Y)
% This shows a comparison the number of steps using the stiff solver.
```

```
ans=        91
```

The following example shows the differences using a nonstiff solver and a stiff solver on a stiff problem. The equation to solve is quite simple:

$$y'(t) = y^2 \times (1 - y)$$

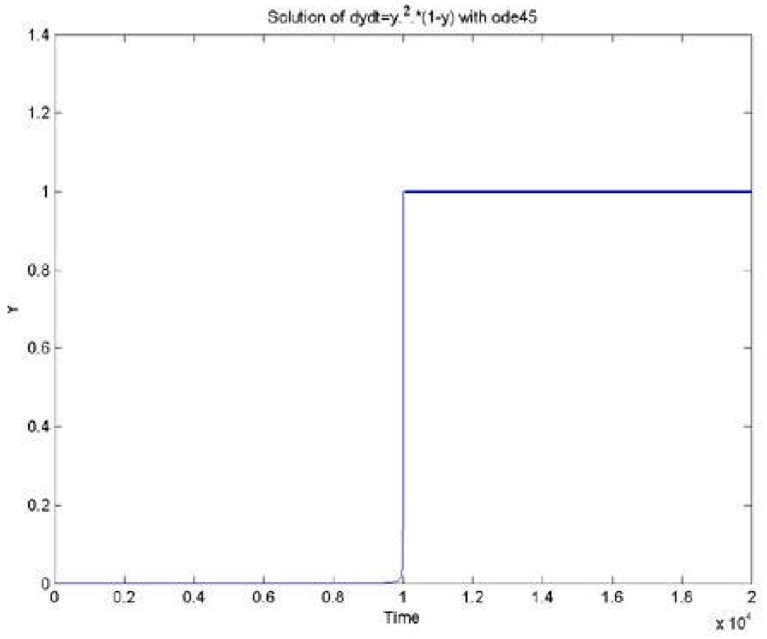
The MATLAB odefun is then

```
function dydt = f(t,y)
dydt = y.^2.*(1-y);
```

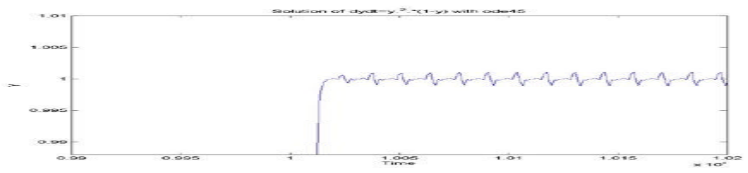
First, solve this system using ODE45, a nonstiff solver.

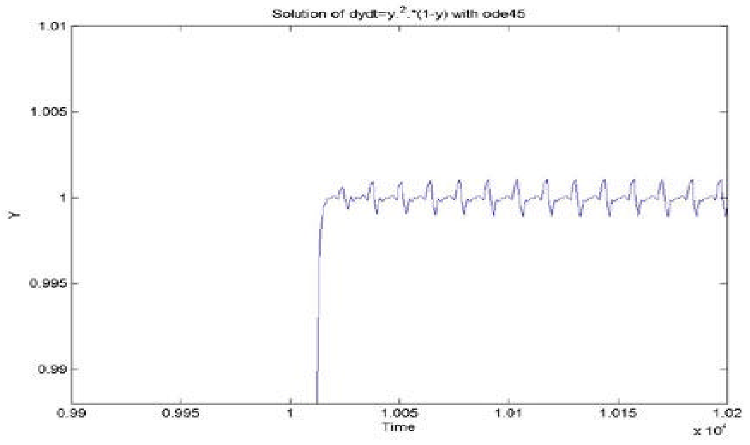
```
tspan = [0,20000];
y0 = 1e-4;
options = [];
[t,y]=ode45(@f,tspan,y0,options);
plot(t,y)
title('Solution of dydt=y.^2.*(1-y) with ode45')
xlabel('Time')
ylabel('Y')
```

The preceding code produces the following figure:



If you zoom in on the region where time is equal to 1 and the solution is equal to 1, you see the following picture:



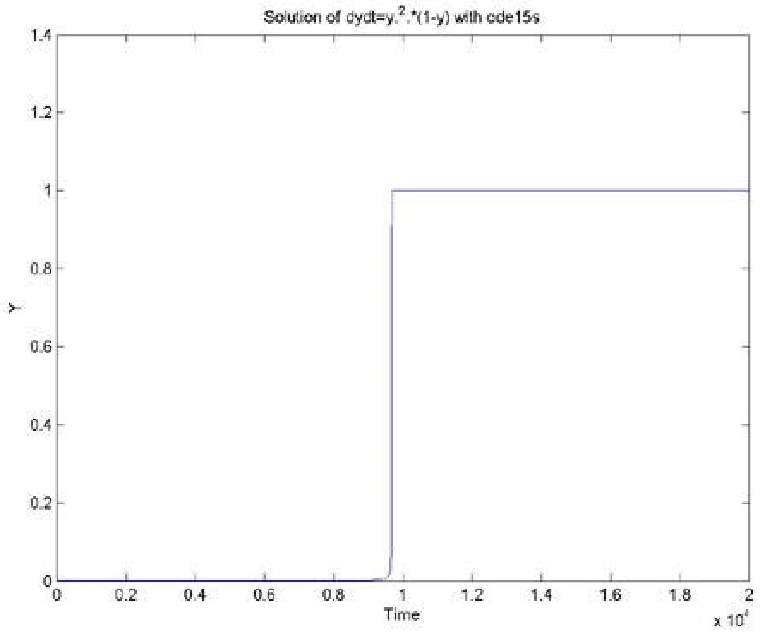


Notice that ODE45 took a large number of steps to solve this rather simple equation (you can verify this by taking the length of `y` or `t`).

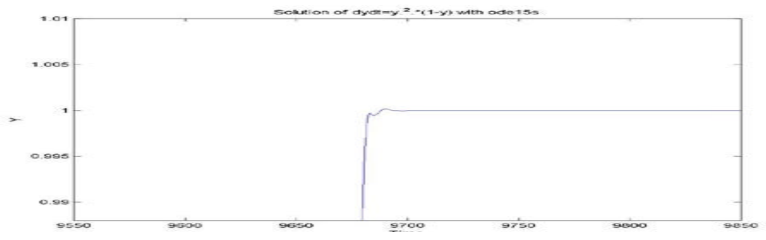
If you now use the solver ODE15S, in the following manner,

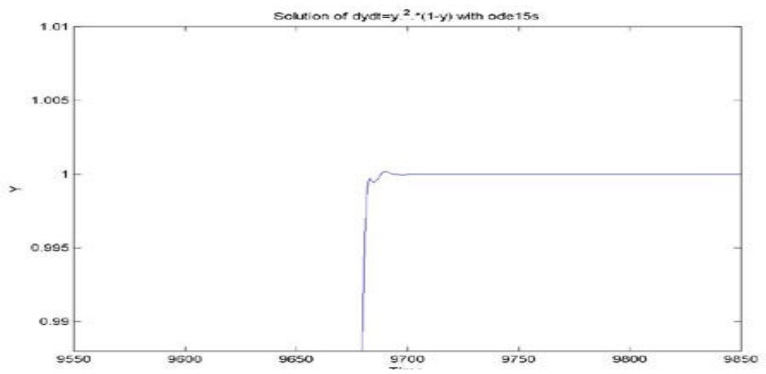
```
[t,y]=ode15s(@f,tspan,y0,options);
% using the same f,tspan,y0 and options structure



plot(t,y)
title('Solution of dydt=y.^2.*(1-y) with ode15s')
xlabel('Time')
ylabel('Y')
```

You obtain the following figure:



If you zoom in on the same region where you saw some instabilities with ODE45, you obtain the following picture:





Notice that ODE15S took a much smaller number of steps. This shows the advantage of using the appropriate solver for a particular problem.

The following equations model diffusion in a chemical reaction. This is another example of a potentially large stiff problem:

$$u_i' = 1 + u_i^2 v_i - 4u_i + \alpha(N+1)^2(u_{i-1} - 2u_i + u_{i+1})$$
$$v_i' = 3u_i - u_i^2 v_i + \alpha(N+1)^2(v_{i-1} - 2v_i + v_{i+1})$$

These are solved on the interval `[ 0, 10 ]` with α equal to `0.02` and

$$\left.\begin{array}{l} u_i(0) = 1 + \sin(2\pi x_i) \\ v_i(0) = 3 \end{array}\right\} \quad x_i = i/(N+1) \quad \text{for } i = 1, 2, \ldots\ldots N$$

Create initial conditions and set options to use a Jacobian pattern:

```
tspan = [0; 10];
y0 = [1+sin((2*pi/(N+1))*(1:N));
repmat(3,1,N)];
options = odeset('Vectorized','on','Jpattern',jpattern(N));
```

Now, create functions defining the differential form and providing the pattern of the Jacobian:

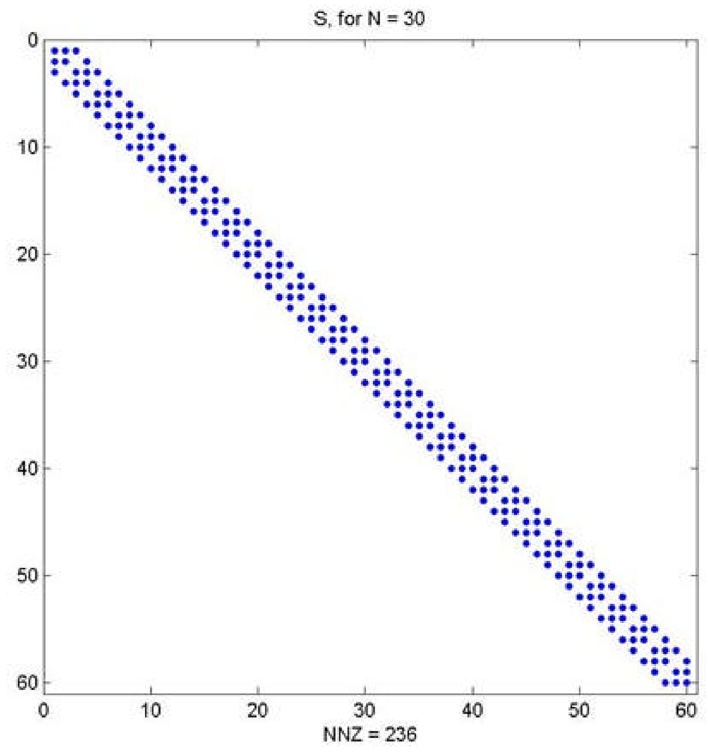
```
function dydt = f(t,y,N)
c = 0.02 * (N+1)^2;
dydt = zeros(2*N,size(y,2));      % preallocate dy/dt
% Evaluate the two components of the function at one edge of
% the grid (with edge conditions).
i = 1;

dydt(i,:) = 1 + y(i+1,:).*y(i,:).^2 - 4*y(i,:) + ...
            c*(1-2*y(i,:)+y(i+2,:));
dydt(i+1,:) = 3*y(i,:) - y(i+1,:).*y(i,:).^2 + ...
              c*(3-2*y(i+1,:)+y(i+3,:));
% Evaluate the two components of the function at all interior
% grid points.
i = 3:2:2*N-3;
dydt(i,:) = 1 + y(i+1,:).*y(i,:).^2 - 4*y(i,:) + ...
            c*(y(i-2,:)-2*y(i,:)+y(i+2,:));
dydt(i+1,:) = 3*y(i,:) - y(i+1,:).*y(i,:).^2 + ...
              c*(y(i-1,:)-2*y(i+1,:)+y(i+3,:));
% Evaluate the two components of the function at the other edge
% of the grid (with edge conditions).
i = 2*N-1;
dydt(i,:) = 1 + y(i+1,:).*y(i,:).^2 - 4*y(i,:) + ...
            c*(y(i-2,:)-2*y(i,:)+1);
dydt(i+1,:) = 3*y(i,:) - y(i+1,:).*y(i,:).^2 + ...
              c*(y(i-1,:)-2*y(i+1,:)+3);
```

For the Jacobian pattern, define the following function:

```
function S = jpattern(N)
B = ones(2*N,5);
```

```
B(2:2:2*N,2) = zeros(N,1);
B(1:2:2*N-1,4) = zeros(N,1);
S = spdiags(B,-2:2,2*N,2*N);
```

The following plot shows the Jacobian pattern for N equal to 30, using the function SPY:



You can then use MATLAB to solve the system as follows:

```
[t,y] = ode15s(@f,tspan,y0,options,N);


 surf((1:N)/(N+1),t,y);
set(gca,'ZLim',[0 1]);
view(142.5,30);
title(['Finite element problem with time-dependent mass ' ...
'matrix, solved by ode15S']);
xlabel('space ( x/\pi )');
ylabel('time');
zlabel('solution');
```

See **the documentation** for other examples of stiff differential equations and how to solve them.

Section 13: How Can I Change Options When Solving a Differential Equation??

You can change various options when solving a differential equation. For more information on creating an options structure, see the documentation on the **ODESET** function.

Section 14: What Option Parameters Can Be Modified?

Of particular interest for day-to-day applications are the parameters `RelTol`, `AbsTol` and `NormControl`, which are related to the error control of the ODE solver. Changing these parameters can increase or decrease the accuracy of the solver.

Section 15: How Can Options Be Used as Functions?

In various applications, you need to set an option as a function, for example, the mass matrix calculation for DAEs. It is strongly suggested that your look at the tutorials provided in **MATLAB Central**, where examples and an explanation are given for various equations.

There is a tutorial for each category of differential equations (BVP, DDE, DAE and ODE) as well as examples. Additional examples can be found **here**.

The **ballode** example, shipped with MATLAB, uses an event function and is also a good reference.

Section 16: How Can Differential Algebraic Equations Systems Be Solved in MATLAB?

The simplest system of differential algebraic equations (DAEs) has the folowing semiexplicit form:

```
u' = f(t,u,v)                    (1a)
0 = g(t,u,v)                     (1b)
```

In this notation, t is the independent variable (time); u stands for the differential variables, and v stands for the algebraic variables.

One idea for solving (1) could be to solve (1a) as an ODE. Evaluating the derivative u' for a given u would require solving the algebraic equation (1b) for the corresponding value of v. This is easy enough in principle, but can lead to initial value problems for which the evaluation of f is rather expensive.

MATLAB solvers use a different approach. To solve DAEs in MATLAB, first you need to combine the differential and algebraic part. Note that any semiexplicit system (1) can be written as

```
M * y' = F(t,y)
```

where

```
M = [I 0; 0 0]
```

and

```
y = [u;v]
```

For differential algebraic equations, the mass matrix M is singular, but such systems can still be solved with ODE15S and ODE23T. In fact, the solvers can handle more general systems, with time- and state-dependent singular mass matrices

```
M(t,y) *y' = F(t,y)             (2)
```

The only restriction on DAEs solved in MATLAB is that they must be of index 1. Semi explicit DAEs are of index 1 when their matrix of partial derivatives dg/dv is nonsingular. For more general, linearly implicit DAEs (2), the index 1 condition is satisfied when the matrix (M+lambda*dF/dy) is nonsingular.

MATLAB does not currently provide solvers for DAEs of index higher than 1.

For more detailed discussion of solving DAEs, see the paper

Shampine,Lawrence F., Reichelt,Mark W., and Kierzenka, Jacek A., Solving Index-1 DAEs in MATLAB and Simulink, SIAM Review, Vol. 41 (1997), No. 3, pp. 538-552. (Available at **MATLAB Central**)

and the following books:

Brenan, K.E., Campbell, S.L., and Petzold, L.R., Numerical Solution of Initial Value Problems in Differential-Algebraic Equations, SIAM, Philadelphia, 1996.

Hairer, E., Lubich, C., and Roche, M., The Numerical Solution of Differential-Algebraic Systems By Runge-Kutta Methods, Springer, Berlin, 1989.

Language Options: English Français Deutsch Español Italiano 한국어

© 1994-2004 The MathWorks, Inc. - Trademarks - Privacy Policy