Provide feedback about this page

Initial Value Problems for ODEs and DAEs

On this page…

ODE Function Summary
Introduction to Initial Value ODE Problems
Solvers for Explicit and Linearly Implicit ODEs
Examples: Solving Explicit ODE Problems
Solver for Fully Implicit ODEs
Example: Solving a Fully Implicit ODE Problem
Changing ODE Integration Properties
Examples: Applying the ODE Initial Value Problem Solvers
Questions and Answers, and Troubleshooting

ODE Function Summary

ODE Initial Value Problem Solvers

The following table lists the initial value problem solvers, the kind of problem you can solve with each solver, and the method each solver uses.

Solver	Solves These Kinds of Problems	Method
`ode45`	Nonstiff differential equations	Runge-Kutta
`ode23`	Nonstiff differential equations	Runge-Kutta
`ode113`	Nonstiff differential equations	Adams
`ode15s`	Stiff differential equations and DAEs	NDFs (BDFs)
`ode23s`	Stiff differential equations	Rosenbrock
`ode23t`	Moderately stiff differential equations and DAEs	Trapezoidal rule
`ode23tb`	Stiff differential equations	TR-BDF2

ode15i	Fully implicit differential equations	BDFs

ODE Solution Evaluation and Extension

You can use the following functions to evaluate and extend solutions to ODEs.

Function	Description
deval	Evaluate the numerical solution using the output of ODE solvers.
odextend	Extend the solution of an initial value problem for an ODE

ODE Solvers Properties Handling

An options structure contains named properties whose values are passed to ODE solvers, and which affect problem solution. Use these functions to create, alter, or access an options structure.

Function	Description
odeset	Create or alter options structure for input to ODE solver.
odeget	Extract properties from options structure created with `odeset`.

ODE Solver Output Functions

If an output function is specified, the solver calls the specified function after every successful integration step. You can use odeset to specify one of these sample functions as the OutputFcn property, or you can modify them to create your own functions.

Function	Description
odeplot	Time-series plot
odephas2	Two-dimensional phase plane plot
odephas3	Three-dimensional phase plane plot
odeprint	Print to command window

[Back to Top]

Introduction to Initial Value ODE Problems

What Is an Ordinary Differential Equation?

Types of Problems Handled by the ODE Solvers

Using Initial Conditions to Specify the Solution of Interest

Working with Higher Order ODEs

What Is an Ordinary Differential Equation?

The ODE solvers are designed to handle *ordinary differential equations*. An ordinary differential equation contains one or more derivatives of a dependent variable y with respect to a single independent variable t, usually referred to as *time*. The derivative of y with respect to t is denoted as y', the second derivative as y'', and so on. Often $y(t)$ is a vector, having elements $y_1, y_2, \ldots, y_n$.

Types of Problems Handled by the ODE Solvers

The ODE solvers handle the following types of first-order ODEs:

- Explicit ODEs of the form $y' = f(t, y)$
- Linearly implicit ODEs of the form $M(t, y) \cdot y' = f(t, y)$, where $M(t, y)$ is a matrix
- Fully implicit ODEs of the form $f(t, y, y') = 0$ (`ode15i` only)

Using Initial Conditions to Specify the Solution of Interest

Generally there are many functions $y(t)$ that satisfy a given ODE, and additional information is necessary to specify the solution of interest. In an *initial value problem,* the solution of interest satisfies a specific *initial condition*, that is, y is equal to y_0 at a given initial time t_0. An initial value problem for an ODE is then

$$\begin{aligned} y' &= f(t, y) \\ y(t_0) &= y_0 \end{aligned} \quad \textbf{(5-1)}$$

If the function $f(t, y)$ is sufficiently smooth, this problem has one and only one solution. Generally there is no analytic expression for the solution, so it is necessary to approximate $y(t)$ by numerical means, such as using one of the ODE solvers.

Working with Higher Order ODEs

The ODE solvers accept only first-order differential equations. However, ODEs often involve a number of dependent variables, as well as derivatives of order higher than one. To use the

ODE solvers, you must rewrite such equations as an equivalent system of first-order differential equations of the form

$$y' = f(t, y)$$

You can write any ordinary differential equation

$$y^{(n)} = f(t, y, y', \ldots, y^{(n-1)})$$

as a system of first-order equations by making the substitutions

$$y_1 = y, \quad y_2 = y', \quad \ldots, \quad y_n = y^{(n-1)}$$

The result is an equivalent system of n first-order ODEs.

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= y_3 \\ &\vdots \\ y_n' &= f(t, y_1, y_2, \ldots, y_n) \end{aligned}$$

Example: Solving an IVP ODE (van der Pol Equation, Nonstiff) rewrites the second-order van der Pol equation

$$y_1'' - \mu(1 - y_1^2)y'_1 + y_1 = 0$$

as a system of first-order ODEs.

Back to Top

Solvers for Explicit and Linearly Implicit ODEs

Solvers for Nonstiff Problems

Solvers for Stiff Problems

Basic ODE Solver Syntax

This section describes the ODE solver functions for explicit or linearly implicit ODEs, as described in Types of Problems Handled by the ODE Solvers. The solver functions implement numerical integration methods for solving initial value problems for ODEs. Beginning at the initial time with initial conditions, they step through the time interval, computing a solution at each time step. If the solution for a time step satisfies the solver's error tolerance criteria, it is a successful step. Otherwise, it is a failed attempt; the solver shrinks the step size and tries again.

Mass Matrix and DAE Properties, in the reference page for `odeset`, explains how to set options to solve more general linearly implicit problems.

The function `ode15i`, which solves implicit ODEs, is described in Solver for Fully Implicit ODEs.

Solvers for Nonstiff Problems

There are three solvers designed for nonstiff problems:

ode45	Based on an explicit Runge-Kutta (4,5) formula, the Dormand-Prince pair. It is a *one-step* solver - in computing $y(t_n)$, it needs only the solution at the immediately preceding time point, $y(t_{n-1})$. In general, ode45 is the best function to apply as a "first try" for most problems.
ode23	Based on an explicit Runge-Kutta (2,3) pair of Bogacki and Shampine. It may be more efficient than ode45 at crude tolerances and in the presence of mild stiffness. Like ode45, ode23 is a one-step solver.
ode113	Variable order Adams-Bashforth-Moulton PECE solver. It may be more efficient than ode45 at stringent tolerances and when the ODE function is particularly expensive to evaluate. ode113 is a *multistep* solver—it normally needs the solutions at several preceding time points to compute the current solution.

Solvers for Stiff Problems

Not all difficult problems are stiff, but all stiff problems are difficult for solvers not specifically designed for them. Solvers for stiff problems can be used exactly like the other solvers. However, you can often significantly improve the efficiency of these solvers by providing them with additional information about the problem. (See Changing ODE Integration Properties.)

There are four solvers designed for stiff problems:

ode15s	Variable-order solver based on the numerical differentiation formulas (NDFs). Optionally it uses the backward differentiation formulas, BDFs (also known as Gear's method). Like ode113, ode15s is a multistep solver. If you suspect that a problem is stiff or if ode45 failed or was very inefficient, try ode15s.
ode23s	Based on a modified Rosenbrock formula of order 2. Because it is a one-step solver, it may be more efficient than ode15s at crude tolerances. It can solve some kinds of stiff problems for which ode15s is not effective.
ode23t	An implementation of the trapezoidal rule using a "free" interpolant. Use this solver if the problem is only moderately stiff and you need a solution without numerical damping.
ode23tb	An implementation of TR-BDF2, an implicit Runge-Kutta formula with a first stage that is a trapezoidal rule step and a second stage that is a backward differentiation formula of order 2. Like ode23s, this solver may be more efficient than ode15s at crude tolerances.

Basic ODE Solver Syntax

All of the ODE solver functions, except for `ode15i`, share a syntax that makes it easy to try any of the different numerical methods, if it is not apparent which is the most appropriate. To apply a different method to the same problem, simply change the ODE solver function name. The simplest syntax, common to all the solver functions, is

```
[t,y] = solver(odefun,tspan,y0,options)
```

where *solver* is one of the ODE solver functions listed previously.

The basic input arguments are

`odefun`	Handle to a function that evaluates the system of ODEs. The function has the form `dydt = odefun(t,y)` where `t` is a scalar, and `dydt` and `y` are column vectors. See Function Handles in the MATLAB Programming documentation for more information.
`tspan`	Vector specifying the interval of integration. The solver imposes the initial conditions at `tspan(1)`, and integrates from `tspan(1)` to `tspan(end)`.
`y0`	Vector of initial conditions for the problem See also Introduction to Initial Value ODE Problems.
`options`	Structure of optional parameters that change the default integration properties. Changing ODE Integration Properties tells you how to create the structure and describes the properties you can specify.

The output arguments contain the solution approximated at discrete points:

`t`	Column vector of time points
`y`	Solution array. Each row in `y` corresponds to the solution at a time returned in the corresponding row of `t`.

See the reference page for the ODE solvers for more information about these arguments.

> **Note** See Evaluating the Solution at Specific Points for more information about solver syntax where a continuous solution is returned.

Back to Top

Examples: Solving Explicit ODE Problems

This section uses the van der Pol equation

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

to describe the process for solving initial value ODE problems using the ODE solvers.

- Example: Solving an IVP ODE (van der Pol Equation, Nonstiff) describes each step of the process. Because the van der Pol equation is a second-order equation, the example must first rewrite it as a system of first order equations.
- Example: The van der Pol Equation, μ = 1000 (Stiff) demonstrates the solution of a stiff problem.
- Evaluating the Solution at Specific Points tells you how to evaluate the solution at specific points.

Note See Basic ODE Solver Syntax for more information.

Example: Solving an IVP ODE (van der Pol Equation, Nonstiff)

This example explains and illustrates the steps you need to solve an initial value ODE problem:

1. **Rewrite the problem as a system of first-order ODEs.** Rewrite the van der Pol equation (second-order)

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

   where $\mu > 0$ is a scalar parameter, by making the substitution $y'_1 = y_2$. The resulting system of first-order ODEs is

   $$y'_1 = y_2$$

   $$y'_2 = \mu(1 - y_1^2)y_2 - y_1$$

   See Working with Higher Order ODEs for more information.
2. **Code the system of first-order ODEs.** Once you represent the equation as a system of first-order ODEs, you can code it as a function that an ODE solver can use. The function must be of the form

   ```
   dydt = odefun(t,y)
   ```

   Although `t` and `y` must be the function's two arguments, the function does not need to use them. The output `dydt`, a column vector, is the derivative of `y`.

   The code below represents the van der Pol system in the function, `vdp1`. The `vdp1` function assumes that $\mu = 1$. The variables y_1 and y_2 are the entries `y(1)` and `y(2)` of a two-element vector.

   ```
   function dydt = vdp1(t,y)
   dydt = [y(2); (1-y(1)^2)*y(2)-y(1)];
   ```

   Note that, although `vdp1` must accept the arguments `t` and `y`, it does not use `t` in its computations.
3. **Apply a solver to the problem.**

Decide which solver you want to use to solve the problem. Then call the solver and pass it the function you created to describe the first-order system of ODEs, the time interval on which you want to solve the problem, and an initial condition vector. See Examples: Solving Explicit ODE Problems and the @ for descriptions of the ODE solvers.

For the van der Pol system, you can use `ode45` on time interval `[0 20]` with initial values `y(1) = 2` and `y(2) = 0`.

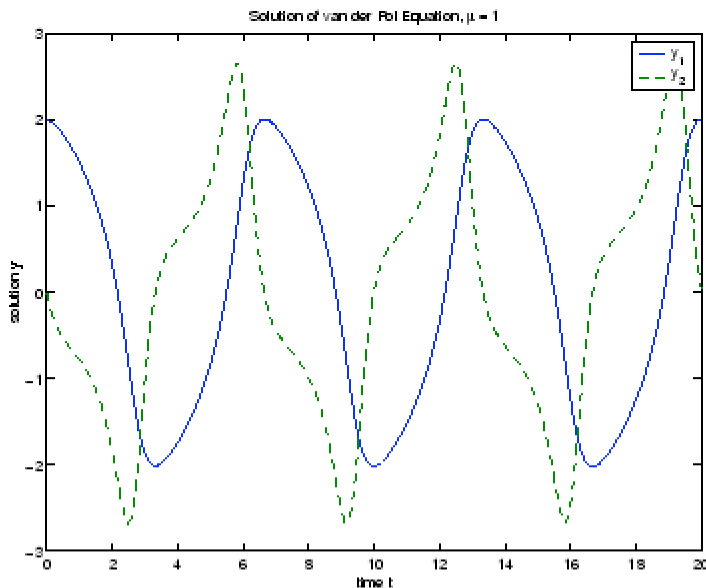
```
[t,y] = ode45(@vdp1,[0 20],[2; 0]);
```

This example uses @ to pass `vdp1` as a function handle to `ode45`. The resulting output is a column vector of time points `t` and a solution array `y`. Each row in `y` corresponds to a time returned in the corresponding row of `t`. The first column of `y` corresponds to y_1, and the second column to y_2.

> **Note** For information on function handles, see the `function_handle` (@), `func2str`, and `str2func` reference pages, and the Function Handles section of in the MATLAB documentation.

4. **View the solver output.** You can simply use the `plot` command to view the solver output.

```
plot(t,y(:,1),'-',t,y(:,2),'--')
title('Solution of van der Pol Equation, \mu = 1');
xlabel('time t');
ylabel('solution y');
legend('y_1','y_2')
```



As an alternative, you can use a solver output function to process the output. The solver calls the function specified in the integration property `OutputFcn` after each successful time step. Use `odeset` to set `OutputFcn` to the desired function. See Solver Output Properties, in the reference page for `odeset`, for more information about `OutputFcn`.

Example: The van der Pol Equation, μ = 1000 (Stiff)

This example presents a stiff problem. For a stiff problem, solutions can change on a time scale that is very short compared to the interval of integration, but the solution of interest changes on a much longer time scale. Methods not designed for stiff problems are ineffective on intervals where the solution changes slowly because they use time steps small enough to resolve the fastest possible change.

When μ is increased to 1000, the solution to the van der Pol equation changes dramatically and exhibits oscillation on a much longer time scale. Approximating the solution of the initial value problem becomes a more difficult task. Because this particular problem is stiff, a solver intended for nonstiff problems, such as `ode45`, is too inefficient to be practical. A solver such as `ode15s` is intended for such stiff problems.

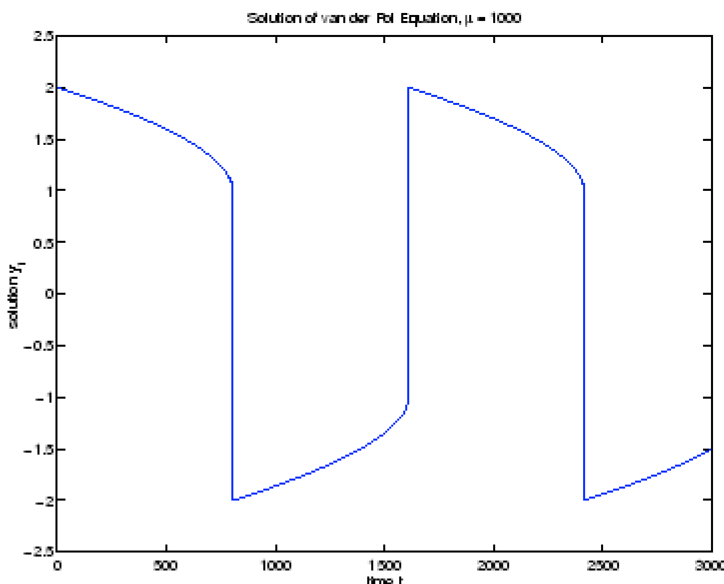
The `vdp1000` function evaluates the van der Pol system from the previous example, but with μ = 1000.

```
function dydt = vdp1000(t,y)
dydt = [y(2); 1000*(1-y(1)^2)*y(2)-y(1)];
```

> **Note** This example hardcodes μ in the ODE function. The `vdpode` example solves the same problem, but passes a user-specified μ as a parameter to the ODE function.

Now use the `ode15s` function to solve the problem with the initial condition vector of `[2; 0]`, but a time interval of `[0 3000]`. For scaling reasons, plot just the first component of `y(t)`.

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



> **Note** For detailed instructions for solving an initial value ODE problem, see Example: Solving an IVP ODE (van der Pol Equation, Nonstiff).

Parameterizing an ODE Function

The preceding sections showed how to solve the van der Pol equation for two different values of the parameter μ. In those examples, the values μ = 1 and μ=1000 are hard-coded in the ODE functions. If you are solving an ODE for several different parameter values, it might be more convenient to include the parameter in the ODE function and assign a value to the parameter each time you run the ODE solver. This section explains how to do this for the van der Pol equation.

One way to provide parameter values to the ODE function is to write an M-file that

- Accepts the parameters as inputs.
- Contains ODE function as a nested function, internally using the input parameters.
- Calls the ODE solver.

The following code illustrates this:

```
function [t,y] = solve_vdp(mu)
tspan = [0 max(20, 3*mu)];
y0 = [2; 0];

% Call the ODE solver ode15s.
[t,y] = ode15s(@vdp,tspan,y0);

    % Define the ODE function as nested function,
    % using the parameter mu.
    function dydt = vdp(t,y)
    dydt = [y(2); mu*(1-y(1)^2)*y(2)-y(1)];
    end
end
```

Because the ODE function `vdp` is a nested function, the value of the parameter `mu` is available to it.

To run the M-file for `mu = 1`, as in Example: Solving an IVP ODE (van der Pol Equation, Nonstiff), enter

```
[t,y] = solve_vdp(1);
```

To run the code for μ = 1000, as in Example: The van der Pol Equation, μ = 1000 (Stiff), enter

```
[t,y] = solve_vdp(1000);
```

See the vdpode code for a complete example based on these functions.

Evaluating the Solution at Specific Points

The numerical methods implemented in the ODE solvers produce a continuous solution over the interval of integration $[a, b]$. You can evaluate the approximate solution, $S(x)$, at any point in $[a, b]$ using the function `deval` and the structure `sol` returned by the solver. For example, if you solve the problem described in Example: Solving an IVP ODE (van der Pol Equation, Nonstiff) by calling `ode45` with a single output argument `sol`,

```
sol = ode45(@vdp1,[0 20],[2; 0]);
```

`ode45` returns the solution as a structure. You can then evaluate the approximate solution at points in the vector `xint = 1:5` as follows:

```
xint = 1:5;
Sxint = deval(sol,xint)

Sxint =

    1.5081    0.3235   -1.8686   -1.7407   -0.8344
   -0.7803   -1.8320   -1.0220    0.6260    1.3095
```

The `deval` function is vectorized. For a vector `xint`, the `i`th column of `Sxint` approximates the solution $y(\text{xint(i)})$.

Back to Top

Solver for Fully Implicit ODEs

The solver `ode15i` solves fully implicit differential equations of the form

$$f(t, y, y') = 0$$

using the variable order BDF method. The basic syntax for `ode15i` is

```
[t,y] = ode15i(odefun,tspan,y0,yp0,options)
```

The input arguments are

`odefun`	A function that evaluates the left side of the differential equation of the form $f(t, y, y') = 0$.
`tspan`	A vector specifying the interval of integration, `[t0,tf]`. To obtain solutions at specific times (all increasing or all decreasing), use `tspan = [t0,t1,...,tf]`.
`y0`, `yp0`	Vectors of initial conditions for $y(t_0)$ and $y'(t_0)$, respectively. The specified values must be consistent; that is, they must satisfy `f(t0,y0,yp0) = 0`. Example: Solving a Fully Implicit ODE Problem shows how to use the function `decic` to compute consistent initial conditions.

options	Optional integration argument created using the odeset function. See the odeset reference page for details.

The output arguments contain the solution approximated at discrete points:

t	Column vector of time points
y	Solution array. Each row in y corresponds to the solution at a time returned in the corresponding row of t.

See the ode15i reference page for more information about these arguments.

> **Note** See Evaluating the Solution at Specific Points for more information about solver syntax where a continuous solution is returned.

Back to Top

Example: Solving a Fully Implicit ODE Problem

The following example shows how to use the function ode15i to solve the implicit ODE problem defined by Weissinger's equation

$$ty^2(y')^3 - y^3(y')^2 + t(t^2+1)y' - t^2y = 0$$

with the initial value $y(1) = \sqrt{3/2}$. The exact solution of the ODE is

$$y(t) = \sqrt{t^2 + 0.5}$$

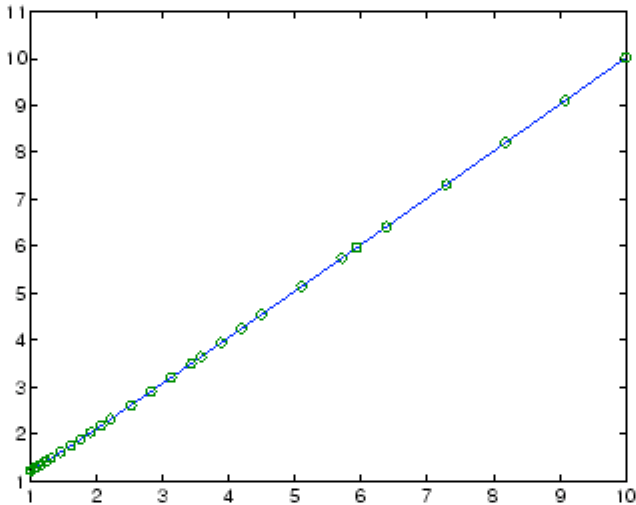
The example uses the function weissinger, which is provided with MATLAB, to compute the left-hand side of the equation.

Before calling ode15i, the example uses a helper function decic to compute a consistent initial value for $y'(t_0)$. In the following call, the given initial value $y(1) = \sqrt{3/2}$ is held fixed and a guess of 0 is specified for $y'(1)$. See the reference page for decic for more information.

```
t0 = 1;
y0 = sqrt(3/2);
yp0 = 0;
[y0,yp0] = decic(@weissinger,t0,y0,1,yp0,0);
```

You can now call ode15i to solve the ODE and then plot the numerical solution against the analytical solution with the following commands.

```
[t,y] = ode15i(@weissinger,[1 10],y0,yp0);
ytrue = sqrt(t.^2 + 0.5);
plot(t,y,t,ytrue,'o');
```



Back to Top

Changing ODE Integration Properties

The default integration properties in the ODE solvers are selected to handle common problems. In some cases, you can improve ODE solver performance by overriding these defaults. You do this by supplying the solvers with an `options` structure that specifies one or more property values.

For example, to change the value of the relative error tolerance of the solver from the default value of `1e-3` to `1e-4`,

1. Create an options structure using the function `odeset` by entering

```
options = odeset('RelTol', 1e-4);
```

2. Pass the `options` structure to the solver as follows:
   - For all solvers except `ode15i`, use the syntax

   ```
   [t,y] = solver(odefun,tspan,y0,options)
   ```

   - For `ode15i`, use the syntax

   ```
   [t,y] = ode15i(odefun,tspan,y0,yp0,options)
   ```

For an example that uses the `options` structure, see Example: Stiff Problem (van der Pol Equation). For a complete description of the available options, see the reference page for `odeset`.

Back to Top

Examples: Applying the ODE Initial Value Problem Solvers

Running the Examples

Example: Simple Nonstiff Problem

Example: Stiff Problem (van der Pol Equation)

Example: Finite Element Discretization

Example: Large, Stiff, Sparse Problem

Example: Simple Event Location

Example: Advanced Event Location

Example: Differential-Algebraic Problem

Example: Computing Nonnegative Solutions

Summary of Code Examples

Running the Examples

This section contains several examples that illustrate the kinds of problems you can solve. For each example, there is a corresponding M-file, included in MATLAB. You can

- View the M-file code in an editor by entering `edit` followed by the name of the M-file at the MATLAB prompt. For example, to view the code for the simple nonstiff problem example, enter

  ```
  edit rigidode
  ```

  Alternatively, if you are reading this in the MATLAB Help Browser, you can click the name of the M-file in the list below.
- Run the example by entering the name of the M-file at the MATLAB prompt.

Example: Simple Nonstiff Problem

`rigidode` illustrates the solution of a standard test problem proposed by Krogh for solvers intended for nonstiff problems [8].

The ODEs are the Euler equations of a rigid body without external forces.

$$y_1' = y_2 y_3$$

$$y_2' = -y_1 y_3$$

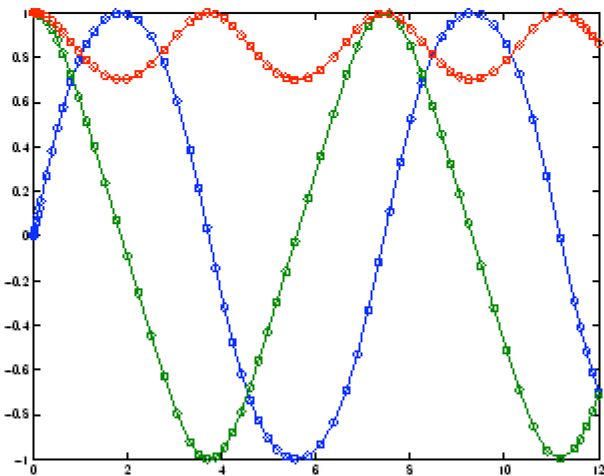
$$y_3' = -0.51 y_1 y_2$$

For your convenience, the entire problem is defined and solved in a single M-file. The differential equations are coded as a subfunction `f`. Because the example calls the `ode45` solver without output arguments, the solver uses the default output function `odeplot` to

plot the solution components.

To run this example, click on the example name, or type rigidode at the command line.

```
function rigidode
%RIGIDODE  Euler equations of a rigid body without external forces
tspan = [0 12];
y0 = [0; 1; 1];

% Solve the problem using ode45
ode45(@f,tspan,y0);
% ------------------------------------------------------------
function dydt = f(t,y)
dydt = [ y(2)*y(3)
        -y(1)*y(3)
        -0.51*y(1)*y(2) ];
```



Example: Stiff Problem (van der Pol Equation)

vdpode illustrates the solution of the van der Pol problem described in Example: The van der Pol Equation, μ = 1000 (Stiff). The differential equations

$$y'_1 = y_2$$

$$y'_2 = \mu(1 - y_1^2)y_2 - y_1$$

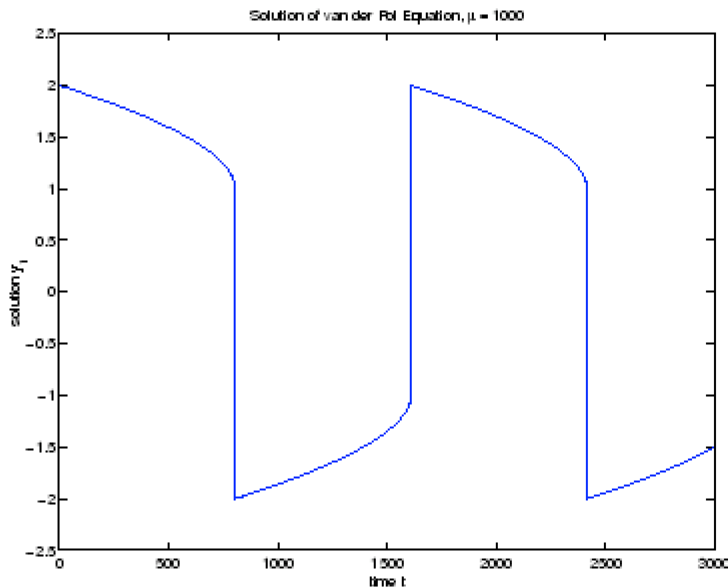
involve a constant parameter μ.

As μ increases, the problem becomes more stiff, and the period of oscillation becomes larger. When μ is 1000 the equation is in relaxation oscillation and the problem is very stiff. The limit cycle has portions where the solution components change slowly and the problem is quite stiff, alternating with regions of very sharp change where it is not stiff (quasi-discontinuities).

By default, the solvers in the ODE suite that are intended for stiff problems approximate Jacobian matrices numerically. However, this example provides a nested function J(t,y) to evaluate the Jacobian matrix $\partial f/\partial y$ analytically at (t,y) for μ = MU. The use of an analytic Jacobian can improve the reliability and efficiency of integration.

To run this example, click on the example name, or type vdpode at the command line. From the command line, you can specify a value of μ as an argument to vdpode. The default is mu μ = 1000.

```
function vdpode(MU)
%VDPODE  Parameterizable van der Pol equation (stiff for large MU)
if nargin < 1
  MU = 1000;     % default
end

tspan = [0; max(20,3*MU)];                 % Several periods
y0 = [2; 0];
options = odeset('Jacobian',@J);

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

plot(t,y(:,1));
title(['Solution of van der Pol Equation, \mu = ' num2str(MU)]);
xlabel('time t');
ylabel('solution y_1');

axis([tspan(1) tspan(end) -2.5 2.5]);
---------------------------------------------------------------
function dydt = f(t,y)
dydt = [            y(2)
         MU*(1-y(1)^2)*y(2)-y(1) ];
end   % End nested function f
---------------------------------------------------------------
function dfdy = J(t,y)
dfdy = [         0                  1
         -2*MU*y(1)*y(2)-1    MU*(1-y(1)^2) ];
end   % End nested function J
end
```



Example: Finite Element Discretization

`fem1ode` illustrates the solution of ODEs that result from a finite element discretization of a partial differential equation. The value of `N` in the call `fem1ode(N)` controls the discretization, and the resulting system consists of `N` equations. By default, `N` is `19`.

This example involves a mass matrix. The system of ODEs comes from a method of lines solution of the partial differential equation

$$e^{-t}\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}$$

with initial condition $u(0, x) = \sin(x)$ and boundary conditions $u(t, 0) = u(t, \pi) = 0$. An integer N is chosen, h is defined as $\pi/(N+1)$, and the solution of the partial differential equation is approximated at $x_k = kh$ for *k* = 0, 1, ..., *N*+1 by

$$u(t,x_k) = \sum_{k=1}^{N} c_k(t)\phi_k(x)$$

Here $\phi_k(x)$ is a piecewise linear function that is 1 at x_k and 0 at all the other x_j. A Galerkin discretization leads to the system of ODEs

$$M(t)c' = Jc \quad \text{where} \quad c(t) = \begin{bmatrix} c_1(t) \\ c_N(t) \end{bmatrix}$$

and the tridiagonal matrices $M(t)$ and J are given by

$$M_{ij} = \begin{cases} 2h/3\exp(-t) & \text{if } i = j \\ h/6\exp(-t) & \text{if } i = j \pm 1 \\ 0 & \text{otherwise} \end{cases}$$

and

$$J_{ij} = \begin{cases} -2/h & \text{if } i = j \\ 1/h & \text{if } i = j \pm 1 \\ 0 & \text{otherwise} \end{cases}$$

The initial values $c(0)$ are taken from the initial condition for the partial differential equation. The problem is solved on the time interval $[0, \pi]$.

In the `fem1ode` example, the properties

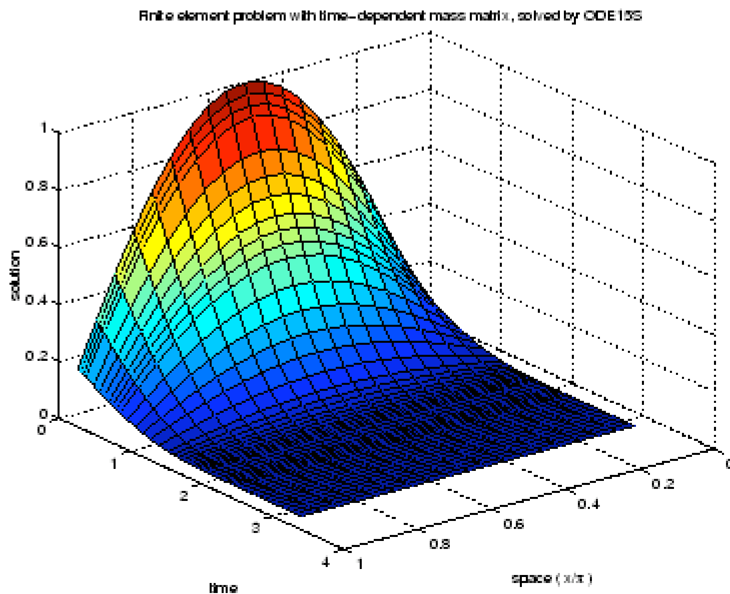
```
options = odeset('Mass',@mass,'MStateDep','none','Jacobian',J)
```

indicate that the problem is of the form $M(t)y' = Jy$. The nested function `mass(t)` evaluates the time-dependent mass matrix $M(t)$ and `J` is the constant Jacobian.

To run this example, click on the example name, or type fem1ode at the command line. From the command line, you can specify a value of N as an argument to `fem1ode`. The default is N = 19.

```
function fem1ode(N)
%FEM1ODE  Stiff problem with a time-dependent mass matrix

if nargin < 1
  N = 19;
end
h = pi/(N+1);
y0 = sin(h*(1:N)');
tspan = [0; pi];

% The Jacobian is constant.
e = repmat(1/h,N,1);     %  e=[(1/h) ... (1/h)];
d = repmat(-2/h,N,1);    %  d=[(-2/h) ... (-2/h)];
% J is shared with the derivative function.
J = spdiags([e d e], -1:1, N, N);

d = repmat(h/6,N,1);
% M is shared with the mass matrix function.
M = spdiags([d 4*d d], -1:1, N, N);

options = odeset('Mass',@mass,'MStateDep','none', ...
                 'Jacobian',J);

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

figure;
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');
%------------------------------------------------------------------
-
function yp = f(t,y)
% Derivative function.
   yp = J*y;     % Constant Jacobian is provided by outer function
end              % End nested function f
%------------------------------------------------------------------
-
function Mt = mass(t)
% Mass matrix function.
   Mt = exp(-t)*M;     % M is provided by outer function
end                    % End nested function mass
%------------------------------------------------------------------
-
end
```



Example: Large, Stiff, Sparse Problem

`brussode` illustrates the solution of a (potentially) large stiff sparse problem. The problem is the classic "Brusselator" system [3] that models diffusion in a chemical reaction

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

and is solved on the time interval `[0,10]` with $\alpha = 1/50$ and

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

There are $2N$ equations in the system, but the Jacobian is banded with a constant width 5 if $u_1, v_1, u_2, v_2, \ldots$

$2N$
the equations are ordered as $u_1, v_1, u_2, v_2, \ldots$

In the call `brussode(N)`, where `N` corresponds to N, the parameter `N` ≥ 2 specifies the number of grid points. The resulting system consists of `2N` equations. By default, `N` is `20`. The problem becomes increasingly stiff and the Jacobian increasingly sparse as `N` increases.

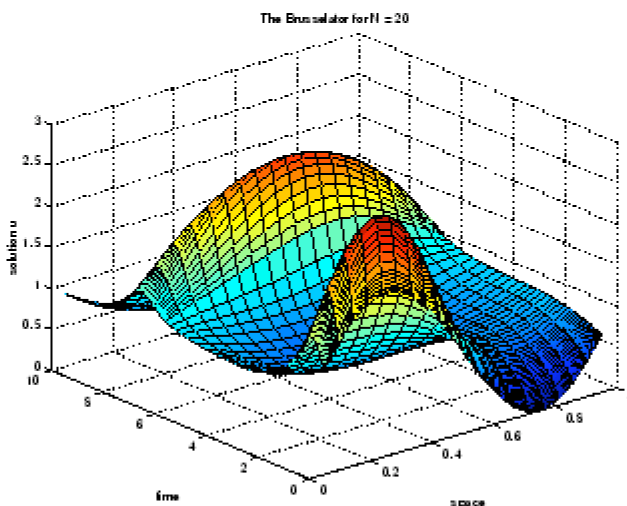
The nested function `f(t,y)` returns the derivatives vector for the Brusselator problem. The subfunction `jpattern(N)` returns a sparse matrix of `1`s and `0`s showing the locations of nonzeros in the Jacobian $\partial f / \partial y$. The example assigns this matrix to the property `JPattern`, and the solver uses the sparsity pattern to generate the Jacobian numerically as a sparse matrix. Providing a sparsity pattern can significantly reduce the number of function evaluations required to generate the Jacobian and can accelerate integration.

For the Brusselator problem, if the sparsity pattern is not supplied, 2`N` evaluations of the function are needed to compute the 2`N`-by-2`N` Jacobian matrix. If the sparsity pattern is supplied, only four evaluations are needed, regardless of the value of `N`.

To run this example, click on the example name, or type brussode at the command line. From the command line, you can specify a value of N as an argument to `brussode`. The default is N = 20.

```
function brussode(N)
%BRUSSODE  Stiff problem modeling a chemical reaction

if nargin < 1
  N = 20;
end

tspan = [0; 10];
y0 = [1+sin((2*pi/(N+1))*(1:N));
repmat(3,1,N)];

options = odeset('Vectorized','on','JPattern',jpattern(N));

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

u = y(:,1:2:end);
x = (1:N)/(N+1);
surf(x,t,u);
view(-40,30);
xlabel('space');
ylabel('time');
zlabel('solution u');
title(['The Brusselator for N = ' num2str(N)]);
% ------------------------------------------------------------------
function dydt = f(t,y)
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);
end   % End nested function f
end   % End function brussode
% ---------------------------------------------------------------
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);
end;
```



Example: Simple Event Location

`ballode` models the motion of a bouncing ball. This example illustrates the event location capabilities of the ODE solvers.

The equations for the bouncing ball are

$$y'_1 = y_2$$
$$y'_2 = -9.8$$

In this example, the event function is coded in a subfunction `events`

```
[value,isterminal,direction] = events(t,y)
```

which returns

- A value of the event function
- The information whether or not the integration should stop when `value = 0` (`isterminal = 1` or `0`, respectively)
- The desired directionality of the zero crossings:

`-1`	Detect zero crossings in the negative direction only
`0`	Detect all zero crossings
`1`	Detect zero crossings in the positive direction only

The length of `value`, `isterminal`, and `direction` is the same as the number of event functions. The `i`th element of each vector, corresponds to the `i`th event function. For an example of more advanced event location, see `orbitode` (Example: Advanced Event Location).

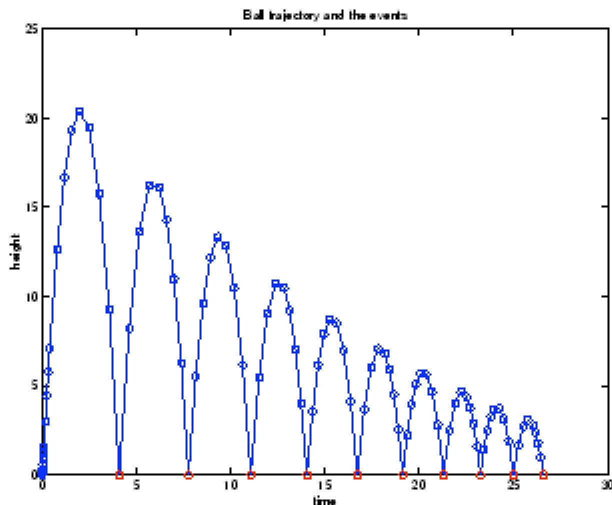
In `ballode`, setting the `Events` property to `@events` causes the solver to stop the integration (`isterminal = 1`) when the ball hits the ground (the height `y(1)` is `0`) during its fall (`direction = -1`). The example then restarts the integration with initial conditions corresponding to a ball that bounced.

To run this example, click on the example name, or type `ballode` at the command line.

```
function ballode
%BALLODE  Run a demo of a bouncing ball.

tstart = 0;
tfinal = 30;
y0 = [0; 20];
refine = 4;
options = odeset('Events',@events,'OutputFcn', @odeplot,...
                 'OutputSel',1,'Refine',refine);

set(gca,'xlim',[0 30],'ylim',[0 25]);
box on
hold on;
```

```
tout = tstart;
yout = y0.';
teout = [];
yeout = [];
ieout = [];
for i = 1:10
  % Solve until the first terminal event.
  [t,y,te,ye,ie] = ode23(@f,[tstart tfinal],y0,options);
  if ~ishold
    hold on
  end
  % Accumulate output.
  nt = length(t);
  tout = [tout; t(2:nt)];
  yout = [yout; y(2:nt,:)];
  teout = [teout; te];    % Events at tstart are never reported.
  yeout = [yeout; ye];
  ieout = [ieout; ie];

  ud = get(gcf,'UserData');
  if ud.stop
    break;
  end

  % Set the new initial conditions, with .9 attenuation.
  y0(1) = 0;
  y0(2) = -.9*y(nt,2);

  % A good guess of a valid first time step is the length of
  % the last valid time step, so use it for faster computation.
  options = odeset(options,'InitialStep',t(nt)-t(nt-refine),...
                           'MaxStep',t(nt)-t(1));
  tstart = t(nt);
end

plot(teout,yeout(:,1),'ro')
xlabel('time');
ylabel('height');
title('Ball trajectory and the events');
hold off
odeplot([],[],'done');
% --------------------------------------------------------------
function dydt = f(t,y)
dydt = [y(2); -9.8];
% --------------------------------------------------------------
function [value,isterminal,direction] = events(t,y)
% Locate the time when height passes through zero in a
```

```
% decreasing direction and stop integration.
value = y(1);       % Detect height = 0
isterminal = 1;     % Stop the integration
direction = -1;     % Negative direction only
```



Example: Advanced Event Location

`orbitode` illustrates the solution of a standard test problem for those solvers that are intended for nonstiff problems. It traces the path of a spaceship traveling around the moon and returning to the earth (Shampine and Gordon [8], p. 246).

The `orbitode` problem is a system of the following four equations shown:

$$y_1' = y_3$$

$$y_2' = y_4$$

$$y_3' = 2y_4 + y_1 - \frac{\mu^*(y_1 + \mu)}{r_1^3} - \frac{\mu(y_1 - \mu^*)}{r_2^3}$$

$$y_4' = -2y_3 + y_2 - \frac{\mu^* y_2}{r_1^3} - \frac{\mu y_2}{r_2^3}$$

where

$$\mu = 1/82.45$$

$$\mu^* = 1 - \mu$$

$$r_1 = \sqrt{(y_1 + \mu)^2 + y_2^2}$$

$$r_2 = \sqrt{(y_1 - \mu^*)^2 + y_2^2}$$

The first two solution components are coordinates of the body of infinitesimal mass, so plotting one against the other gives the orbit of the body. The initial conditions have been chosen to make the orbit periodic. The value of μ corresponds to a spaceship traveling around the moon and the earth. Moderately stringent tolerances are necessary to reproduce

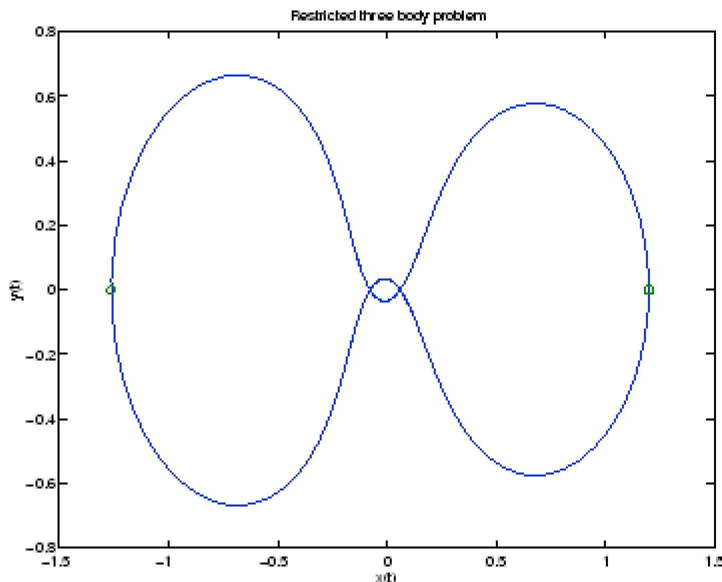
the qualitative behavior of the orbit. Suitable values are `1e-5` for [RelTol] and `1e-4` for [AbsTol].

The nested `events` function includes event functions that locate the point of maximum distance from the starting point and the time the spaceship returns to the starting point. Note that the events are located accurately, even though the step sizes used by the integrator are *not* determined by the location of the events. In this example, the ability to specify the direction of the zero crossing is critical. Both the point of return to the initial point and the point of maximum distance have the same event function value, and the direction of the crossing is used to distinguish them.

To run this example, click on the example name, or type [orbitode] at the command line. The example uses the output function [odephas2] to produce the two-dimensional phase plane plot and let you to see the progress of the integration.

```
function orbitode
%ORBITODE  Restricted three-body problem

mu = 1 / 82.45;
mustar = 1 - mu;
y0 = [1.2; 0; 0; -1.04935750983031990726];
tspan = [0 7];

options = odeset('RelTol',1e-5,'AbsTol',1e-4,...
                 'OutputFcn',@odephas2,'Events',@events);

[t,y,te,ye,ie] = ode45(@f,tspan,y0,options);

plot(y(:,1),y(:,2),ye(:,1),ye(:,2),'o');
title ('Restricted three body problem')
ylabel ('y(t)')
xlabel ('x(t)')
% --------------------------------------------------------------
function dydt = f(t,y)
r13 = ((y(1) + mu)^2 + y(2)^2) ^ 1.5;
r23 = ((y(1) - mustar)^2 + y(2)^2) ^ 1.5;
dydt = [ y(3)
         y(4)
         2*y(4) + y(1) - mustar*((y(1)+mu)/r13) - ...
                         mu*((y(1)-mustar)/r23)
        -2*y(3) + y(2) - mustar*(y(2)/r13) - mu*(y(2)/r23) ];
end   % End nested function f
% --------------------------------------------------------------
function [value,isterminal,direction] = events(t,y)
% Locate the time when the object returns closest to the
% initial point y0 and starts to move away, and stop integration.
% Also locate the time when the object is farthest from the
% initial point y0 and starts to move closer.
```

```
%
% The current distance of the body is
%
%   DSQ = (y(1)-y0(1))^2 + (y(2)-y0(2))^2
%       = <y(1:2)-y0(1:2),y(1:2)-y0(1:2)>
%
% A local minimum of DSQ occurs when d/dt DSQ crosses zero
% heading in the positive direction.  We can compute d(DSQ)/dt as
%
%  d(DSQ)/dt = 2*(y(1:2)-y0(1:2))'*dy(1:2)/dt = ...
%                   2*(y(1:2)-y0(1:2))'*y(3:4)
%
dDSQdt = 2 * ((y(1:2)-y0(1:2))' * y(3:4));
value = [dDSQdt; dDSQdt];
isterminal = [1; 0];             % Stop at local minimum
direction = [1; -1];             % [local minimum, local maximum]
end   % End nested function events
end
```



Example: Differential-Algebraic Problem

`hb1dae` reformulates the hb1ode example as a *differential-algebraic equation* (DAE) problem. The Robertson problem coded in `hb1ode` is a classic test problem for codes that solve stiff ODEs.

$$y_1' = -0.04y_1 + 10^4 y_2 y_3$$

$$y_2' = 0.04y_1 - 10^4 y_2 y_3 - 3 \cdot 10^7 y_2^2$$

$$y_3' = 3 \cdot 10^7 y_2^2$$

Note The Robertson problem appears as an example in the prolog to LSODI [4].

In hb1ode, the problem is solved with initial conditions $y_1(0) = 1$, $y_2(0) = 0$, $y_3(0) = 0$ to steady state. These differential equations satisfy a linear conservation law that is used to reformulate the problem as the DAE

$$y_1' = -0.04y_1 + 10^4 y_2 y_3$$

$$y_2' = 0.04y_1 - 10^4 y_2 y_3 - 3 \cdot 10^7 y_2^2$$

$$0 = y_1 + y_2 + y_3 - 1$$

These equations do not have a solution for $y(0)$ with components that do not sum to 1. The problem has the form of $My' = f(t, y)$ with

$$M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

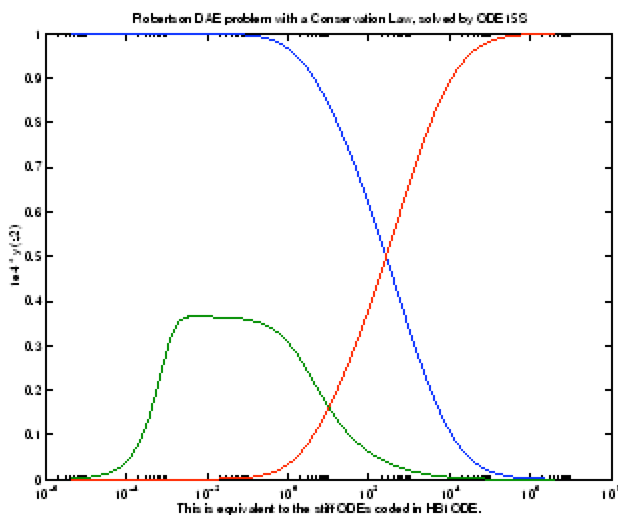
M is singular, but hb1dae does not inform the solver of this. The solver must recognize that the problem is a DAE, not an ODE. Similarly, although consistent initial conditions are obvious, the example uses an inconsistent value $y_3(0) = 10^{-3}$ to illustrate computation of consistent initial conditions.

To run this example, click on the example name, or type hb1dae at the command line. Note that hb1dae

- Imposes a much smaller absolute error tolerance on y_2 than on the other components. This is because y_2 is much smaller than the other components and its major change takes place in a relatively short time.
- Specifies additional points at which the solution is computed to more clearly show the behavior of y_2.
- Multiplies y_2 by 10^4 to make y_2 visible when plotting it with the rest of the solution.
- Uses a logarithmic scale to plot the solution on the long time interval.

```
function hb1dae
%HB1DAE  Stiff differential-algebraic equation (DAE)

% A constant, singular mass matrix
M = [1 0 0
     0 1 0
     0 0 0];

% Use an inconsistent initial condition to test initialization.
y0 = [1; 0; 1e-3];
tspan = [0 4*logspace(-6,6)];

% Use the LSODI example tolerances. The 'MassSingular' property
% is left at its default 'maybe' to test the automatic detection
% of a DAE.
options = odeset('Mass',M,'RelTol',1e-4,...
```

```
                    'AbsTol',[1e-6 1e-10 1e-6],'Vectorized','on');

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

y(:,2) = 1e4*y(:,2);

semilogx(t,y);
ylabel('1e4 * y(:,2)');
title(['Robertson DAE problem with a Conservation Law, '...
       'solved by ODE15S']);
xlabel('This is equivalent to the stiff ODEs coded in HB1ODE.');
% ------------------------------------------------------------
function out = f(t,y)
out = [ -0.04*y(1,:) + 1e4*y(2,:).*y(3,:)
         0.04*y(1,:) - 1e4*y(2,:).*y(3,:) - 3e7*y(2,:).^2
         y(1,:) + y(2,:) + y(3,:) - 1 ];
```



Example: Computing Nonnegative Solutions

If certain components of the solution must be nonnegative, use odeset to set the `NonNegative` property for the indices of these components.

> **Note** This option is not available for ode23s, ode15i, or for implicit solvers (ode15s, ode23t, ode23tb) applied to problems where there is a mass matrix.

Imposing nonnegativity is not always a trivial task. We suggest that you use this option only when necessary, for example in instances in which the application of a solution or integration will fail otherwise.

Consider the following initial value problem solved on the interval `[0, 40]`:

```
y' = - |y|, y(0) = 1
```

The solution of this problem decays to zero. If a solver produces a negative approximate

solution, it begins to track the solution of the ODE through this value, the solution goes off to minus infinity, and the computation fails. Using the `NonNegative` property prevents this from happening.

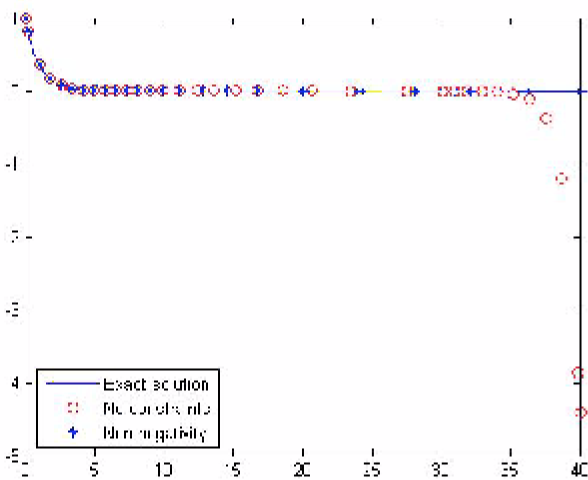
In this example, the first call to [ode45] uses the defaults for the solver parameters:

```
ode = @(t,y) -abs(y);
[t0,y0] = ode45(ode,[0, 40], 1);
```

The second uses options to impose nonnegativity conditions:

```
options = odeset('NonNegative',1);
[t1,y1] = ode45(ode,[0, 40], 1, options);
```

This plot compares the numerical solution to the exact solution.



Here is a more complete view of the code used to obtain this plot:

```
ode = @(t,y) -abs(y);
options = odeset('Refine',1);
[t0,y0] = ode45(ode,[0, 40], 1,options);
options = odeset(options,'NonNegative',1);
[t1,y1] = ode45(ode,[0, 40], 1, options);
t = linspace(0,40,1000);
y = exp(-t);
plot(t,y,'b-',t0,y0,'ro',t1,y1,'b*');
legend('Exact solution','No constraints','Nonnegativity', ...
       'Location','SouthWest')
```

The MATLAB kneeode Demo. The MATLAB `kneeode` demo solves the "knee problem" by imposing a nonnegativity constraint on the numerical solution. The initial value problem is

```
ε*y' = (1-x)*y - y^2,    y(0) = 1
```

For $0 < \varepsilon < 1$, the solution of this problem approaches null isoclines $y = 1 - x$ and $y = 0$

for x < 1 and x > 1, respectively. The numerical solution, when computed with default tolerances, follows the y = 1 - x isocline for the whole interval of integration. Imposing nonnegativity constraints results in the correct solution.

Here is the code that makes up the kneeode demo:

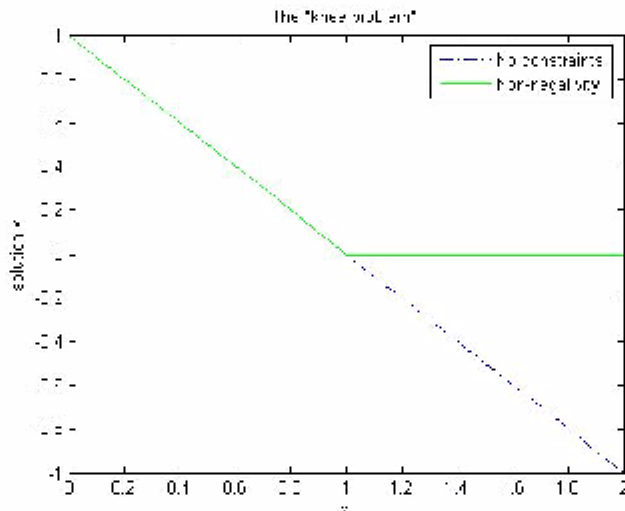
```
function kneeode
%KNEEODE  The "knee problem" with Nonnegativity constraints.

% Problem parameter
epsilon = 1e-6;

y0 = 1;
xspan = [0, 2];

% Solve without imposing constraints
options = [];
[x1,y1] = ode15s(@odefcn,xspan,y0,options);

% Impose nonnegativity constraint
options = odeset('NonNegative',1);
[x2,y2] = ode15s(@odefcn,xspan,y0,options);

figure
plot(x1,y1,'b.-',x2,y2,'g-')
axis([0,2,-1,1]);
title('The "knee problem"');
legend('No constraints','nonnegativity')
xlabel('x');
ylabel('solution y')

   function yp = odefcn(x,y)
      yp = ((1 - x)*y - y^2)/epsilon;
   end
end  % kneeode
```

The derivative function is defined within nested function odefcn. The value of epsilon used in odefcn is obtained from the outer function:

```
function yp = odefcn(x,y)
yp = ((1 - x)*y - y^2)/epsilon;
end
```

The demo solves the problem using the ode15s function, first with the default options, and then by imposing a nonnegativity constraint. To run the demo, type kneeode at the MATLAB command prompt.

Here is the output plot. The plot confirms correct solution behavior after imposing constraints.



Summary of Code Examples

The following table lists the M-files for all the ODE initial value problem examples. Click the example name to see the code in an editor. Type the example name at the command line to run it.

> **Note** The Differential Equations Examples browser enables you to view the code for the ODE examples and DAE examples. You can also run the examples from the browser. Click these links to invoke the browser, or type `odeexamples('ode')` or `odeexamples('dae')` at the command line.

Example	Description
`amp1dae`	Stiff DAE — electrical circuit
`ballode`	Simple event location — bouncing ball
`batonode`	ODE with time- and state-dependent mass matrix — motion of a baton
`brussode`	Stiff large problem — diffusion in a chemical reaction (the Brusselator)
`burgersode`	ODE with strongly state-dependent mass matrix — Burgers' equation solved using a moving mesh technique
`fem1ode`	Stiff problem with a time-dependent mass matrix — finite element method
`fem2ode`	Stiff problem with a constant mass matrix — finite element method
`hb1ode`	Stiff ODE problem solved on a very long interval — Robertson chemical reaction

`hb1dae`	Robertson problem — stiff, linearly implicit DAE from a conservation law
`ihb1dae`	Robertson problem — stiff, fully implicit DAE
`iburgersode`	Burgers' equation solved as implicit ODE system
`kneeode`	The "knee problem" with nonnegativity constraints
`orbitode`	Advanced event location — restricted three body problem
`rigidode`	Nonstiff problem — Euler equations of a rigid body without external forces
`vdpode`	Parameterizable van der Pol equation (stiff for large μ)

Back to Top

Questions and Answers, and Troubleshooting

This section contains a number of tables that answer questions about the use and operation of the ODE solvers:

- General ODE Solver Questions
- Problem Size, Memory Use, and Computation Speed
- Time Steps for Integration
- Error Tolerance and Other Options
- Solving Different Kinds of Problems
- Troubleshooting

General ODE Solver Questions

Question	Answer
How do the ODE solvers differ from `quad` or `quadl`?	`quad` and `quadl` solve problems of the form $y' = f(t)$. The ODE solvers handle more general problems $y' = f(t, y)$, linearly implicit problems that involve a mass matrix $M(t, y)y' = f(t, y)$, and fully implicit problems $f(t, y, y') = 0$.
Can I solve ODE systems in which there are more equations than unknowns, or vice versa?	No.

Problem Size, Memory Use, and Computation Speed

Question	Answer
How large a problem can I solve with the ODE suite?	The primary constraints are memory and time. At each time step, the solvers for nonstiff problems allocate vectors of length `n`, where `n` is the number of equations in the system. The solvers for stiff problems but also allocate an `n`-by-`n` Jacobian matrix. For these solvers it may be advantageous to use the sparse option. If the problem is nonstiff, or if you are using the sparse option, it may be possible to solve a problem with thousands of unknowns. In this case, however, storage of the result can be problematic. Try asking the solver to evaluate the solution at specific points only, or call the solver with no output arguments and use an output function to monitor the solution.
I'm solving a very large system, but only care about a couple of the components of `y`. Is there any way to avoid storing all of the elements?	Yes. The user-installable output function capability is designed specifically for this purpose. When you call the solver with no output arguments, the solver does not allocate storage to hold the entire solution history. Instead, the solver calls `OutputFcn(t,y,flag)` at each time step. To keep the history of specific elements, write an output function that stores or plots only the elements you care about.
What is the startup cost of the integration and how can I reduce it?	The biggest startup cost occurs as the solver attempts to find a step size appropriate to the scale of the problem. If you happen to know an appropriate step size, use the `InitialStep` property. For example, if you repeatedly call the integrator in an event location loop, the last step that was taken before the event is probably on scale for the next integration. See `ballode` for an example.

Time Steps for Integration

Question	Answer

The first step size that the integrator takes is too large, and it misses important behavior.	You can specify the first step size with the `InitialStep` property. The integrator tries this value, then reduces it if necessary.
Can I integrate with fixed step sizes?	No.

Error Tolerance and Other Options

Question	Answer
How do I choose `RelTol` and `AbsTol`?	`RelTol`, the relative accuracy tolerance, controls the number of correct digits in the answer. `AbsTol`, the absolute error tolerance, controls the difference between the answer and the solution. At each step, the error `e` in component `i` of the solution satisfies `|e(i)| ≤max(RelTol*abs(y(i)),AbsTol(i))` Roughly speaking, this means that you want `RelTol` correct digits in all solution components except those smaller than thresholds `AbsTol(i)`. Even if you are not interested in a component `y(i)` when it is small, you may have to specify `AbsTol(i)` small enough to get some correct digits in `y(i)` so that you can accurately compute more interesting components.
I want answers that are correct to the precision of the computer. Why can't I simply set `RelTol` to `eps`?	You can get close to machine precision, but not that close. The solvers do not allow `RelTol` near `eps` because they try to approximate a continuous function. At tolerances comparable to `eps`, the machine arithmetic causes all functions to look discontinuous.

How do I tell the solver that I don't care about getting an accurate answer for one of the solution components?	You can increase the absolute error tolerance corresponding to this solution component. If the tolerance is bigger than the component, this specifies no correct digits for the component. The solver may have to get some correct digits in this component to compute other components accurately, but it generally handles this automatically.

Solving Different Kinds of Problems

Question	Answer
Can the solvers handle partial differential equations (PDEs) that have been discretized by the method of lines?	Yes, because the discretization produces a system of ODEs. Depending on the discretization, you might have a form involving mass matrices – the ODE solvers provide for this. Often the system is stiff. This is to be expected when the PDE is parabolic and when there are phenomena that happen on very different time scales such as a chemical reaction in a fluid flow. In such cases, use one of the four solvers: `ode15s`, `ode23s`, `ode23t`, `ode23tb`. If there are many equations, set the `JPattern` property. This might make the difference between success and failure due to the computation being too expensive. For an example that uses `JPattern`, see Example: Large, Stiff, Sparse Problem. When the system is not stiff, or not very stiff, `ode23` or `ode45` is more efficient than `ode15s`, `ode23s`, `ode23t`, or `ode23tb`. Parabolic-elliptic partial differential equations in 1-D can be solved directly with the MATLAB PDE solver, `pdepe`. For more information, see Partial Differential Equations.

Can I solve differential-algebraic equation (DAE) systems?	Yes. The solvers ode15s and ode23t can solve some DAEs of the form $M(t,y)y' = f(t,y)$ where $M(t,y)$ is singular. The DAEs must be of index 1. ode15i can solve fully implicit DAEs of index 1, $f(t,y,y') = 0$. For examples, see amp1dae, hb1dae, or ihb1dae.
Can I integrate a set of sampled data?	Not directly. You have to represent the data as a function by interpolation or some other scheme for fitting data. The smoothness of this function is critical. A piecewise polynomial fit like a spline can look smooth to the eye, but rough to a solver; the solver takes small steps where the derivatives of the fit have jumps. Either use a smooth function to represent the data or use one of the lower order solvers (ode23, ode23s, ode23t, ode23tb) that is less sensitive to this.
What do I do when I have the final and not the initial value?	All the solvers of the ODE suite allow you to solve backwards or forwards in time. The syntax for the solvers is `[t,y] = ode45(odefun,[t0 tf],y0);`and the syntax accepts `t0 > tf`.

Troubleshooting

Question	Answer

The solution doesn't look like what I expected.	If you're right about its appearance, you need to reduce the error tolerances from their default values. A smaller relative error tolerance is needed to compute accurately the solution of problems integrated over "long" intervals, as well as solutions of problems that are moderately unstable. You should check whether there are solution components that stay smaller than their absolute error tolerance for some time. If so, you are not asking for any correct digits in these components. This may be acceptable for these components, but failing to compute them accurately may degrade the accuracy of other components that depend on them.
My plots aren't smooth enough.	Increase the value of `Refine` from its default of `4` in `ode45` and `1` in the other solvers. The bigger the value of `Refine`, the more output points. Execution speed is not affected much by the value of `Refine`.
I'm plotting the solution as it is computed and it looks fine, but the code gets stuck at some point.	First verify that the ODE function is smooth near the point where the code gets stuck. If it isn't, the solver must take small steps to deal with this. It may help to break `tspan` into pieces on which the ODE function is smooth. If the function is smooth and the code is taking extremely small steps, you are probably trying to solve a stiff problem with a solver not intended for this purpose. Switch to `ode15s`, `ode23s`, `ode23t`, or `ode23tb`.

My integration proceeds very slowly, using too many time steps.	First, check that your `tspan` is not too long. Remember that the solver uses as many time points as necessary to produce a smooth solution. If the ODE function changes on a time scale that is very short compared to the `tspan`, the solver uses a lot of time steps. Long-time integration is a hard problem. Break `tspan` into smaller pieces. If the ODE function does not change noticeably on the `tspan` interval, it could be that your problem is stiff. Try using `ode15s`, `ode23s`, `ode23t`, or `ode23tb`. Finally, make sure that the ODE function is written in an efficient way. The solvers evaluate the derivatives in the ODE function many times. The cost of numerical integration depends critically on the expense of evaluating the ODE function. Rather than recompute complicated constant parameters at each evaluation, store them in globals or calculate them once and pass them to nested functions.
I know that the solution undergoes a radical change at time t where `t0 ≤ t ≤ tf` but the integrator steps past without "seeing" it.	If you know there is a sharp change at time `t`, it might help to break the `tspan` interval into two pieces, `[t0 t]` and `[t tf]`, and call the integrator twice. If the differential equation has periodic coefficients or solution, you might restrict the maximum step size to the length of the period so the integrator won't step over periods.

© 1984-2007 The MathWorks, Inc. • Terms of Use • Patents • Trademarks • Acknowledgments