# Boundary-Value Problems

## Function Summary

- BVP Solver
- BVP Helper Functions
- BVP Solver Options

### BVP Solver

| Solver | Description |
|---|---|
| `bvp4c` | Solve boundary value problems for ordinary differential equations. |
| `bvp5c` | Solve boundary value problems for ordinary differential equations. |

### BVP Helper Functions

| Function | Description |
|---|---|
| `bvpinit` | Form the initial guess for `bvp4c`. |
| `deval` | Evaluate the numerical solution using the output of `bvp4c`. |

### BVP Solver Options

An options structure contains named properties whose values are passed to `bvp4c`, and which affect problem solution. Use these functions to create, alter, or access an options structure.

| Function | Description |
|---|---|
| `bvpset` | Create/alter the BVP options structure. |
| `bvpget` | Extract properties from options structure created with `bvpset`. |

## Boundary Value Problems

The BVP solver is designed to handle systems of ordinary differential equations

$$y' = f(x, y)$$

where $x$ is the independent variable, $y$ is the dependent variable, and $y'$ represents the derivative of $y$ with respect to $x$ $dy/dx$.

See Choose an ODE Solver for general information about ODEs.

### Boundary Conditions

In a *boundary value problem*, the solution of interest satisfies certain boundary conditions. These conditions specify a relationship between the values of the solution at more than one $x$. In its basic syntax, `bvp4c` is designed to solve two-point BVPs, i.e., problems where the solution sought on an interval $[a, b]$ must satisfy the boundary conditions

$$g(y(a), y(b)) = 0$$

Unlike initial value problems, a boundary value problem may not have a solution, may have a finite number of solutions, or may have infinitely many solutions. As an integral part of the process of solving a BVP, you need to provide a guess for the required solution. The quality of this guess can be critical for the solver performance and even for a successful computation.

There may be other difficulties when solving BVPs, such as problems imposed on infinite intervals or problems that involve singular coefficients. Often BVPs involve unknown parameters $p$ that have to be determined as part of solving the problem

$$y' = f(x, y, p)$$

$$g(y(a), y(b), p) = 0$$

In this case, the boundary conditions must suffice to determine the value of $p$.

## BVP Solver

- The BVP Solver
- BVP Solver Syntax
- BVP Solver Options

### The BVP Solver

The function `bvp4c` solves two-point boundary value problems for ordinary differential equations (ODEs). It integrates a system of first-order ordinary differential equations

$$y' = f(x, y)$$

on the interval $[a, b]$, subject to general two-point boundary conditions

$$bc(y(a), y(b)) = 0$$

It can also accommodate other types of BVP problems, such as those that have any of the following:

- Unknown parameters
- Singularities in the solutions
- Multipoint conditions

In this case, the number of boundary conditions must be sufficient to determine the solution and the unknown parameters.

`bvp4c` produces a solution that is continuous on $[a,b]$ and has a continuous first derivative there. You can use the function `deval` and the output of `bvp4c` to evaluate the solution at specific points on the interval of integration.

`bvp4c` is a finite difference code that implements the 3-stage Lobatto IIIa formula. This is a collocation formula and the collocation polynomial provides a $C^1$-continuous solution that is fourth-order accurate uniformly in the interval of integration. Mesh selection and error control are based on the residual of the continuous solution.

The collocation technique uses a mesh of points to divide the interval of integration into subintervals. The solver determines a numerical solution by solving a global system of algebraic equations resulting from the boundary conditions, and the collocation conditions imposed on all the subintervals. The solver then estimates the error of the numerical solution on each subinterval. If the solution does not satisfy the tolerance criteria, the solver adapts the mesh and repeats the process. The user *must*provide the points of the initial mesh as well as an initial approximation of the solution at the mesh points.

### BVP Solver Syntax

The basic syntax of the BVP solver is

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

The input arguments are

| | | |
|---|---|---|
| `odefun` | A function handle that evaluates the differential equations. It has the basic form<br>`dydx = odefun(x,y)`<br>where `x` is a scalar, and `dydx` and `y` are column vectors. `odefun` can also accept a vector of unknown parameters and a variable number of known parameters, (see BVP Solver Options). | |
| `bcfun` | Handle to a function that evaluates the residual in the boundary conditions. It has the basic form<br>`res = bcfun(ya,yb)`<br>where `ya` and `yb` are column vectors representing `y(a)` and `y(b)`, and`res` is a column vector of the residual in satisfying the boundary conditions. `bcfun` can also accept a vector of unknown parameters and a variable number of known parameters, (see BVP Solver Options). | |
| `solinit` | Structure with fields `x` and `y`: | |
| | `x` | Ordered nodes of the initial mesh. Boundary conditions are imposed at *a* = `solinit.x(1)` and*b* = `solinit.x(end)`. |
| | `y` | Initial guess for the solution with `solinit.y(:,i)` a guess for the solution at the node `solinit.x(i)`. |
| | The structure can have any name, but the fields must be named `x` and `y`. It can also contain a vector that provides an initial guess for unknown parameters. You can form `solinit` with the helper function `bvpinit`. See the `bvpinit` reference page for details. | |

The output argument `sol` is a structure created by the solver. In the basic case the structure has fields `x`, `y`, `yp`, and `solver`.

| | |
|---|---|
| `sol.x` | Nodes of the mesh selected by `bvp4c` |
| `sol.y` | Approximation to $y(x)$ at the mesh points of `sol.x` |
| `sol.yp` | Approximation to $y'(x)$ at the mesh points of `sol.x` |
| `sol.solver` | `'bvp4c'` |

The structure `sol` returned by `bvp4c` contains an additional field if the problem involves unknown parameters:

| | |
|---|---|
| `sol.parameters` | Value of unknown parameters, if present, found by the solver. |

The function `deval` uses the output structure `sol` to evaluate the numerical solution at any point from `[a,b]`.

### BVP Solver Options

For more advanced applications, you can specify solver options by passing an input argument `options`.

| | |
|---|---|
| `options` | Structure of optional parameters that change the default integration properties. This is the fourth input argument.<br>`sol = bvp4c(odefun,bcfun,solinit,options)`<br>You can create the structure options using the function `bvpset`. The`bvpset` reference page describes the properties you can specify. |

### Integrator Options

The default integration properties in the BVP solver `bvp4c` are selected to handle common problems. In some cases, you can improve solver performance by overriding these defaults. You do this by supplying `bvp4c` with an `options` structure that specifies one or more property values.

For example, to change the value of the relative error tolerance of `bvp4c` from the default value of `1e-3` to `1e-4`,

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

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

2. Pass the `options` structure to `bvp4c` as follows:

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

For a complete description of the available options, see the reference page for `bvpset`.

### Examples



**Solving the Problem.** This example determines the fourth eigenvalue of Mathieu's Equation. It illustrates how to write second-order differential equations as a system of two first-order ODEs and how to use `bvp4c` to determine an unknown parameter λ.

The task is to compute the fourth ($q = 5$) eigenvalue lambda λ of Mathieu's equation

$$y'' + (\lambda - 2\,q\cos 2x)y = 0$$

Because the unknown parameter λ is present, this second-order differential equation is subject to *three* boundary conditions

$$y(0) = 1$$

$$y'(0) = 0$$

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

**Note:** The file, `mat4bvp.m`, contains the complete code for this example. All the functions required by `bvp4c` are coded in this file as nested or local functions. To see the code in an editor, type `edit mat4bvp` at the command line. To run it, type `mat4bvp` at the command line.

1. **Rewrite the problem as a first-order system.** To use `bvp4c`, you must rewrite the equations as an equivalent system of first-order differential equations. Using a substitution $y_1 = y$ and $y_2 = y'$, the differential equation is written as a system of two first-order equations

$y_1' = y_2$

$y_2' = -(\lambda - 2q \cos 2x)y_1$

Note that the differential equations depend on the unknown parameter λ. The boundary conditions become

$y_1(0) - 1 = 0$

$y_2(0) = 0$

$y_2(\pi) = 0$

2. **Code the system of first-order ODEs.** Once you represent the equation as a first-order system, you can code it as a function that `bvp4c` can use. Because there is an unknown parameter, the function must be of the form

```
dydx = odefun(x,y,parameters)
```

The following code represents the system in the function, `mat4ode`. Variable `q` is shared with the outer function:

```
function dydx = mat4ode(x,y,lambda)
dydx = [ y(2)
         -(lambda - 2*q*cos(2*x))*y(1) ];
end   % End nested function mat4ode
```

4. **Code the boundary conditions function.** You must also code the boundary conditions in a function. Because there is an unknown parameter, the function must be of the form

```
res = bcfun(ya,yb,parameters)
```

The code below represents the boundary conditions in the function, `mat4bc`.

```
function res = mat4bc(ya,yb,lambda)
res = [  ya(2)
         yb(2)
        ya(1)-1 ];
```

6. **Create an initial guess.** To form the guess structure `solinit` with `bvpinit`, you need to provide initial guesses for both the solution and the unknown parameter.

The function `mat4init` provides an initial guess for the solution. `mat4init` uses $y = \cos 4x$ because this function satisfies the boundary conditions and has the correct qualitative behavior (the correct number of sign changes).

```
function yinit = mat4init(x)
yinit = [  cos(4*x)
           -4*sin(4*x) ];
```

In the call to `bvpinit`, the third argument, `lambda`, provides an initial guess for the unknown parameter λ.

```
lambda = 15;
solinit = bvpinit(linspace(0,pi,10),@mat4init,lambda);
```

This example uses the `@` symbol to pass `mat4init` as a function handle to `bvpinit`.

7. **Apply the BVP solver.** The `mat4bvp` example calls `bvp4c` with the functions `mat4ode` and `mat4bc` and the structure `solinit` created with `bvpinit`.

```
sol = bvp4c(@mat4ode,@mat4bc,solinit);
```

8. **View the results.** Complete the example by displaying the results:

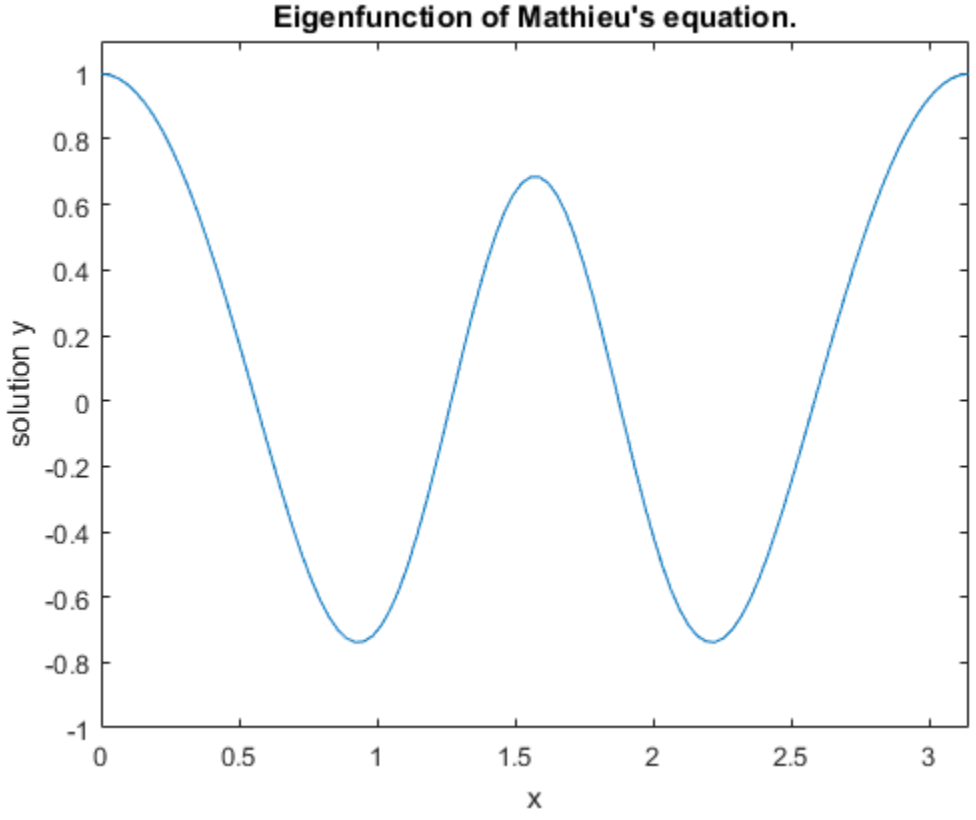
   a. Print the value of the unknown parameter λ found by `bvp4c`.

```
b.      fprintf('Fourth eigenvalue is approximately %7.3f.\n',...
                sol.parameters)
```

   c. Use `deval` to evaluate the numerical solution at 100 equally spaced points in the interval [0, π], and plot its first component. This component approximates $y(x)$.

```
d.      xint = linspace(0,pi);
e.      Sxint = deval(sol,xint);
f.      plot(xint,Sxint(1,:))
g.      axis([0 pi -1 1.1])
h.      title('Eigenfunction of Mathieu''s equation.')
i.      xlabel('x')
        ylabel('solution y')
```

The following plot shows the eigenfunction associated with the final eigenvalue λ = 17.097.



**Finding Unknown Parameters.** The `bvp4c` solver can find unknown parameters $p$ for problems of the form

$$y' = f(x,y,p)$$

$$bc(y(a), y(b), p) = 0$$

You must provide `bvp4c` an initial guess for any unknown parameters in the vector`solinit.parameters`. When you call `bvpinit` to create the structure `solinit`, specify the initial guess as a vector in the additional argument `parameters`.

```
solinit = bvpinit(x,v,parameters)
```

The `bvp4c` function arguments `odefun` and `bcfun` must each have a third argument.

```
dydx = odefun(x,y,parameters)
```

```
res = bcfun(ya,yb,parameters)
```

While solving the differential equations, `bvp4c` adjusts the value of unknown parameters to satisfy the boundary conditions. The solver returns the final values of these unknown parameters in `sol.parameters`.

**Evaluating the Solution.** The collocation method implemented in `bvp4c` produces a $C^1$-continuous solution over the whole interval of integration $[a,b]$. You can evaluate the approximate solution, $S(x)$, at any point in $[a,b]$ using the helper function `deval` and the structure `sol` returned by `bvp4c`.

```
Sxint = deval(sol,xint)
```

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

### Continuation

- Introduction
- Using Continuation to Solve a BVP
- Using Continuation to Verify Consistency

**Introduction.** To solve a boundary value problem, you need to provide an initial guess for the solution. The quality of your initial guess can be critical to the solver performance, and to being able to solve the problem at all. However, coming up with a sufficiently good guess can be the most challenging part of solving a boundary value problem. Certainly, you should apply the knowledge of the problem's physical origin. Often a problem can be solved as a sequence of relatively simpler problems, i.e., a*continuation*.

This example shows how to use continuation to:

- Solve a difficult BVP
- Verify a solution's consistent behavior

**Using Continuation to Solve a BVP.** This example solves the differential equation

$$\varepsilon y'' + xy' = \varepsilon\pi^2 \cos(\pi x) - \pi x \sin(\pi x)$$

for $\varepsilon = 10^{-4}$, on the interval [–1 1], with boundary conditions $y(-1) = -2$ and $y(1) = 0$. For $0 < \varepsilon < 1$, the solution has a transition layer at $x = 0$. Because of this rapid change in the solution for small values of ε, the problem becomes difficult to solve numerically.

The example solves the problem as a sequence of relatively simpler problems, i.e., a continuation. The solution of one problem is used as the initial guess for solving the next problem.

**Note:** The file, `shockbvp.m`, contains the complete code for this example. All required functions are coded as nested functions in this file. To see the code in an editor, type `edit shockbvp` at the command line. To run it, type `shockbvp` at the command line.

**Note:** This problem appears in [1] to illustrate the mesh selection capability of a well established BVP code COLSYS.

1. **Code the ODE and boundary condition functions.** Code the differential equation and the boundary conditions as functions that `bvp4c` can use:

   The code below represents the differential equation and the boundary conditions in the functions `shockODE` and `shockBC`. Note that `shockODE` is vectorized to improve solver performance. The additional parameter ε is represented by `e` and is shared with the outer function.

```
function dydx = shockODE(x,y)
pix = pi*x;
dydx = [  y(2,:)
          -x/e.*y(2,:) - pi^2*cos(pix) - pix/e.*sin(pix) ];
end %   End nested function shockODE


function res = shockBC(ya,yb)
res = [ ya(1)+2
        yb(1)   ];
end %   End nested function shockBC
```

2. **Provide analytical partial derivatives.** For this problem, the solver benefits from using analytical partial derivatives. The code below represents the derivatives in functions `shockJac` and `shockBCJac`.

```
function jac = shockJac(x,y)
jac = [ 0   1
        0 -x/e ];
end %   End nested function shockJac

function [dBCdya,dBCdyb] = shockBCJac(ya,yb)
dBCdya = [ 1 0
           0 0 ];
```

```
dBCdyb = [ 0 0
           1 0 ];
end %   End nested function shockBCJac
```

shockJac shares `e` with the outer function.

Tell `bvp4c` to use these functions to evaluate the partial derivatives by setting the options FJacobian and BCJacobian. Also set `'Vectorized'` to `'on'` to indicate that the differential equation function `shockODE` is vectorized.

```
options = bvpset('FJacobian',@shockJac,...
                 'BCJacobian',@shockBCJac,...
                 'Vectorized','on');
```

14. **Create an initial guess.** You must provide `bvp4c` with a guess structure that contains an initial mesh and a guess for values of the solution at the mesh points. A constant guess of $y(x) \equiv 1$ and $y'(x) \equiv 0$, and a mesh of five equally spaced points on [–1 1] suffice to solve the problem for $\varepsilon = 10^{-2}$. Use `bvpinit` to form the guess structure.

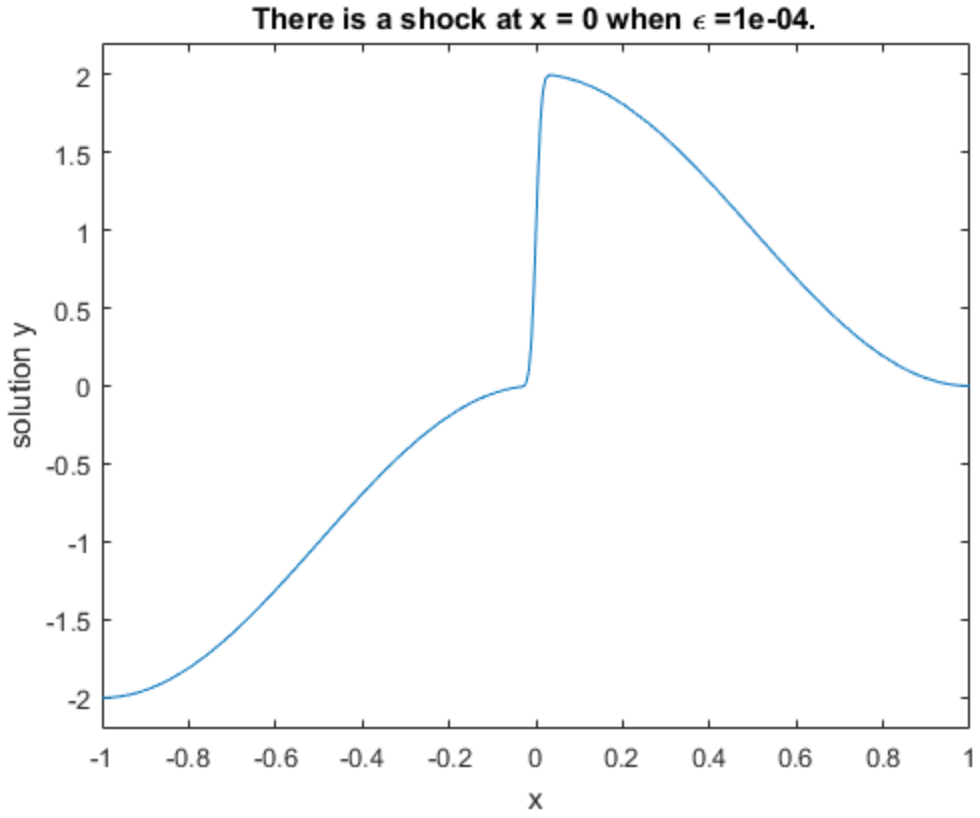
```
sol = bvpinit([-1 -0.5 0 0.5 1],[1 0]);
```

16. **Use continuation to solve the problem.** To obtain the solution for the parameter $\varepsilon = 10^{-4}$, the example uses continuation by solving a sequence of problems for $\varepsilon = 10^{-2}$, $10^{-3}$, $10^{-4}$. The solver `bvp4c` does not perform continuation automatically, but the code's user interface has been designed to make continuation easy. The code uses the output `sol` that `bvp4c` produces for one value of `e` as the guess in the next iteration.

```
e = 0.1;
for i=2:4
    e = e/10;
    sol = bvp4c(@shockODE,@shockBC,sol,options);
end
```

21. **View the results.** Complete the example by displaying the final solution

```
plot(sol.x,sol.y(1,:))
axis([-1 1 -2.2 2.2])
title(['There is a shock at x = 0 when \epsilon = '...
      sprintf('%.e',e) '.'])
xlabel('x')
ylabel('solution y')
```




**Using Continuation to Verify Consistency.** Falkner-Skan BVPs arise from similarity solutions of viscous, incompressible, laminar flow over a flat plate. An example is

$$f''' + ff'' + \beta(1 - (f')^2) = 0$$

for $\beta = 0.5$ on the interval $[0, \infty]$ with boundary conditions $f(0) = 0$, $f'(0) = 0$, and $f'(\infty) = 1$.

The BVP cannot be solved on an infinite interval, and it would be impractical to solve it for even a very large finite interval. So, the example tries to solve a sequence of problems posed on increasingly larger intervals to verify the solution's consistent behavior as the boundary approaches $\infty$.

The example imposes the infinite boundary condition at a finite point called `infinity`. The example then uses continuation in this end point to get convergence for increasingly larger values of `infinity`. It uses `bvpinit` to extrapolate the solution`sol` for one value of `infinity` as an initial guess for the new value of `infinity`. The plot of each successive solution is superimposed over those of previous solutions so they can easily be compared for consistency.

**Note:** The file, `fsbvp.m`, contains the complete code for this example. All required functions are coded as nested functions in this file. To see the code in an editor, type `edit fsbvp` at the command line. To run it, type `fsbvp` at the command line.

1. **Code the ODE and boundary condition functions.** Code the differential equation and the boundary conditions as functions that `bvp4c` can use. The problem parameter `beta` is shared with the outer function.

```
2. function dfdeta = fsode(eta,f)
```

```
dfdeta = [ f(2)
            f(3)
           -f(1)*f(3) - beta*(1 - f(2)^2) ];
end %   End nested function fsode

function res = fsbc(f0,finf)
res = [f0(1)
       f0(2)
       finf(2) - 1];
end %   End nested function fsbc
```

13. **Create an initial guess.** You must provide `bvp4c` with a guess structure that contains an initial mesh and a guess for values of the solution at the mesh points. A crude mesh of five points and a constant guess that satisfies the boundary conditions are good enough to get convergence when `infinity = 3`.

```
infinity = 3;
maxinfinity = 6;

solinit = bvpinit(linspace(0,infinity,5),[0 0 1]);
```

18. **Solve on the initial interval.** The example obtains the solution for `infinity = 3`. It then prints the computed value of $f''(0)$ for comparison with the value reported by Cebeci and Keller [2]:

```
sol = bvp4c(@fsode,@fsbc,solinit);
eta = sol.x;
f = sol.y;

fprintf('\n');
fprintf('Cebeci & Keller report that f''''(0) = 0.92768.\n')
fprintf('Value computed using infinity = %g is %7.5f.\n', ...
         infinity,f(3,1))
```

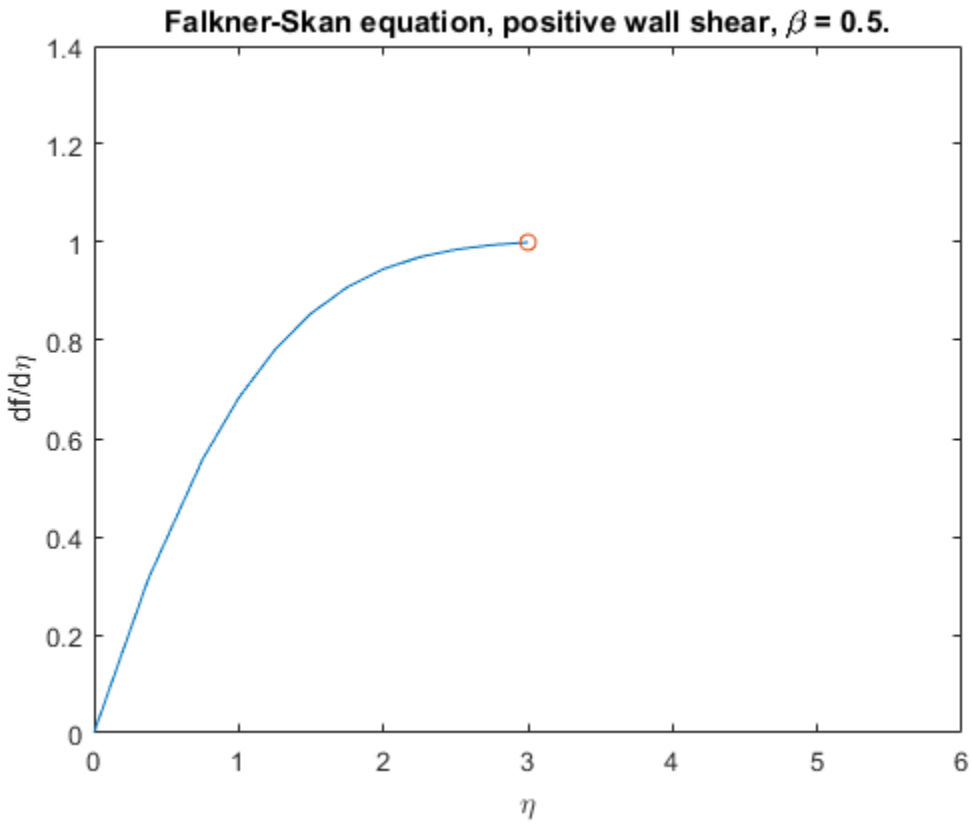
The example prints

```
Cebeci & Keller report that f''(0) = 0.92768.
Value computed using infinity = 3 is 0.92915.
```

27. **Setup the figure and plot the initial solution.**

```
figure
plot(eta,f(2,:),eta(end),f(2,end),'o');
axis([0 maxinfinity 0 1.4]);
```

```
title('Falkner-Skan equation, positive wall shear, \beta = 0.5.')
xlabel('\eta')
ylabel('df/d\eta')
hold on
drawnow
shg
```




37. **Use continuation to solve the problem and plot subsequent solutions.** The example then solves the problem for `infinity = 4, 5, 6`. It uses `bvpinit` to extrapolate the solution `sol` for one value of `infinity` as an initial guess for the next value of `infinity`. For each iteration, the example prints the computed value of $f''(0)$ and superimposes a plot of the solution in the existing figure.

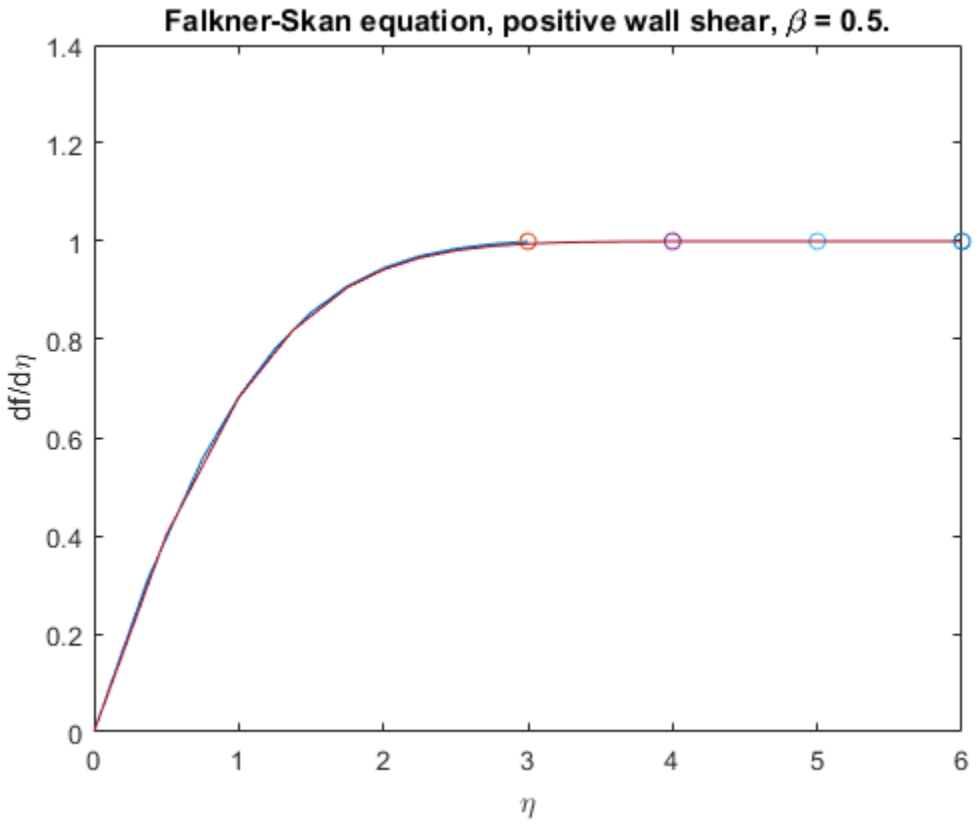
```
for Bnew = infinity+1:maxinfinity

  solinit = bvpinit(sol,[0 Bnew]); % Extend solution to Bnew.
  sol = bvp4c(@fsode,@fsbc,solinit);
  eta = sol.x;
  f = sol.y;

```

```
45.   fprintf('Value computed using infinity = %g is %7.5f.\n', ...
46.           Bnew,f(3,1))
47.   plot(eta,f(2,:),eta(end),f(2,end),'o');
48.   drawnow
49.
50. end
51. hold off
```

The example prints

```
Value computed using infinity = 4 is 0.92774.
Value computed using infinity = 5 is 0.92770.
Value computed using infinity = 6 is 0.92770.
```

Note that the values approach 0.92768 as reported by Cebeci and Keller. The superimposed plots confirm the consistency of the solution's behavior.




**Singular BVPs**



**Introduction.** The function `bvp4c` solves a class of singular BVPs of the form

$$y'0=\frac{1}{x}Sy+f(x,y)=g\left(y(0),y(b)\right) \qquad \textbf{(12-1)}$$

It can also accommodate unknown parameters for problems of the form

$$y'0=\frac{1}{x}Sy+f(x,y,p)=\left(g(y(0),y(b),p\right)$$

Singular problems must be posed on an interval $[0,b]$ with $b > 0$. Use `bvpset` to pass the constant matrix $S$ to `bvp4c` as the value of the `'SingularTerm'` integration property. Boundary conditions at $x = 0$ must be consistent with the necessary condition for a smooth solution, $Sy(0) = 0$. An initial guess should also satisfy this necessary condition.

When you solve a singular BVP using

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

`bvp4c` requires that your function `odefun(x,y)` return only the value of the $f(x, y)$ term in Equation 5-2.

**Emden's equation.** Emden's equation arises in modeling a spherical body of gas. The PDE of the model is reduced by symmetry to the ODE

$$y''+\frac{2}{x}y'+y^5=0$$

on an interval [0,1]. The coefficient $2/x$ is singular at $x = 0$, but symmetry implies the boundary condition $y'(0) = 0$. With this boundary condition, the term

$$\frac{2}{x}y'(0)$$

is well-defined as $x$ approaches 0. For the boundary condition $y(1)=\frac{3}{2}$, this BVP has the analytical solution

$$y(x)=\left(1+\frac{x^2}{3}\right)^{-1/2}$$

**Note:** The file, `emdenbvp.m`, contains the complete code for this example. It contains all the required functions coded as local functions. To see the code in an editor, type `edit emdenbvp` at the command line. To run it, type `emdenbvp` at the command line.

1. **Rewrite the problem as a first-order system and identify the singular term.** Using a substitution $y_1 = y$ and $y_2 = y'$, write the differential equation as a system of two first-order equations

   $$y_1'=y_2 \quad y_2'=-\frac{2}{x}y_2-y_5^1$$

   The boundary conditions become

   $$y_2(0)=0 \quad y_1(1)=\frac{3}{2}$$

   Writing the ODE system in a vector-matrix form

$$\begin{bmatrix} y_1' \\ y_2' \end{bmatrix} = \frac{1}{x} \begin{bmatrix} 0 & 0 \\ 0 & -2 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} + \begin{bmatrix} y_2 \\ -y_5^1 \end{bmatrix}$$

the terms of Equation 5-2 are identified as

$$S = \begin{bmatrix} 0 & 0 \\ 0 & -2 \end{bmatrix}$$

and

$$f(x,y) = \begin{bmatrix} y_2 \\ -y_5^1 \end{bmatrix}$$

2. **Code the ODE and boundary condition functions.** Code the differential equation and the boundary conditions as functions that `bvp4c` can use.

```
function dydx = emdenode(x,y)
dydx = [  y(2)
          -y(1)^5 ];
function res = emdenbc(ya,yb)
res = [ ya(2)
         yb(1) - sqrt(3)/2 ];
```

9. **Setup integration properties.** Use the matrix as the value of the `'SingularTerm'` integration property.

```
S = [0,0;0,-2];
options = bvpset('SingularTerm',S);
```

12. **Create an initial guess.** This example starts with a mesh of five points and a constant guess for the solution.

$$y_1(x) \equiv \sqrt{3}/2 \quad y_2(x) \equiv 0$$

Use `bvpinit` to form the guess structure

```
guess = [sqrt(3)/2;0];
solinit = bvpinit(linspace(0,1,5),guess);
```

13. **Solve the problem.** Use the standard `bvp4c` syntax to solve the problem.

```
sol = bvp4c(@emdenode,@emdenbc,solinit,options);
```

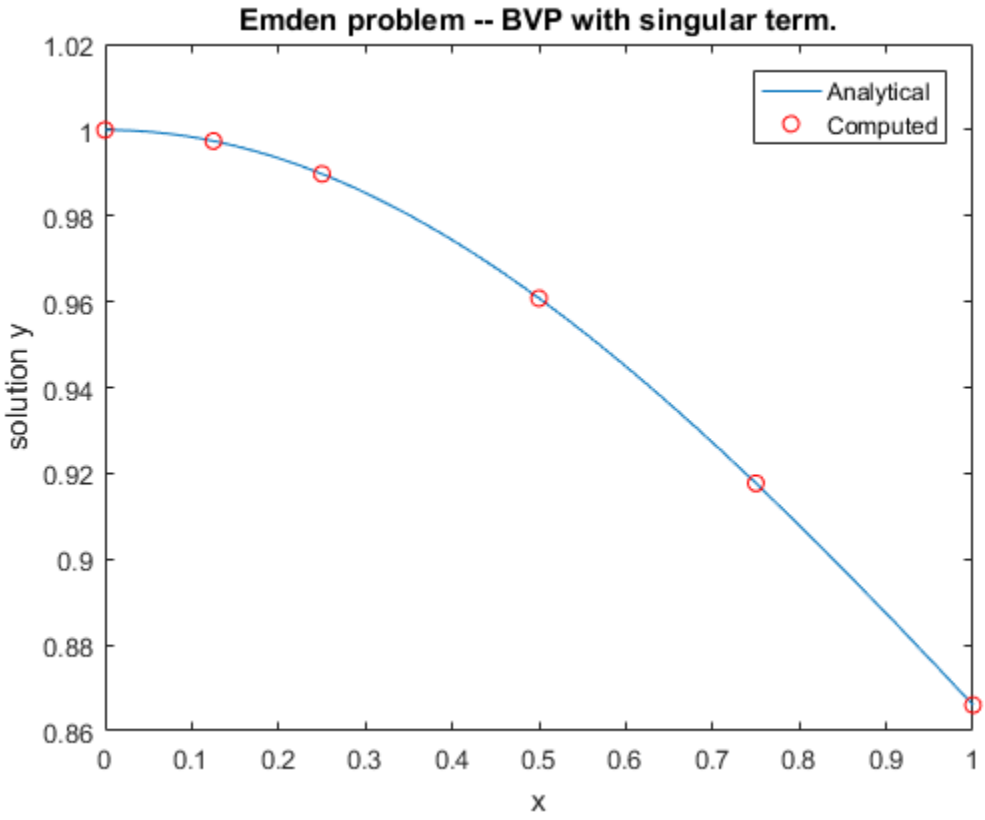
15. **View the results.** This problem has an analytical solution

$$y(x) = \left(1 + \frac{x^2}{3}\right)^{-1/2}$$

The example evaluates the analytical solution at 100 equally spaced points and plots it along with the numerical solution computed using `bvp4c`.

```
x = linspace(0,1);
```

```
truy = 1 ./ sqrt(1 + (x.^2)/3);
plot(x,truy,sol.x,sol.y(1,:),'ro');
title('Emden problem -- BVP with singular term.')
legend('Analytical','Computed');
xlabel('x');
ylabel('solution y');
```




### Multipoint BVPs

In multipoint boundary value problems, the solution of interest satisfies conditions at points inside the interval of integration. The `bvp4c` function is useful in solving such problems.

The following example shows how the multipoint capability in `bvp4c` can improve efficiency when you are solving a nonsmooth problem. The following equations are solved on $0 \le x \le \lambda$ for constant parameters `n`, $\kappa$, $\lambda > 1$, and $\eta = \lambda^2/(n \times \kappa^2)$. These are subject to boundary conditions `v(0) = 0` and $C(\lambda) = 1$:

```
v' = (C - 1)/n
C' = (v * C - min(x,1))/η
```

The term `min(x,1)` is not smooth at `xc = 1`, and this can affect the solver's efficiency. By introducing an interface point at `xc = 1`, smooth solutions can be obtained on [0,1] and [1,λ]. To get a continuous solution over the entire interval [0,λ], the example imposes matching conditions at the interface.

**Note:** The file, `threebvp.m`, contains the complete code for this example, and it solves the problem for $\lambda$ = 2, n = 0.05, and several values of $\kappa$. All required functions are coded as nested functions in `threebvp.m`. To see the code in an editor, type `edit threebvp` at the command line. To run it, type `threebvp` at the command line.

The example takes you through the following steps:

1. **Determine the interfaces and divide the interval of integration into regions.** Introducing an interface point at `xc = 1` divides the problem into two regions in which the solutions remain smooth. The differential equations for the two regions are

   Region 1: $0 \le x \le 1$

   ```
   v' = (C - 1)/n
   C' = (v * C - x)/η
   ```

   Region 2: $1 \le x \le \lambda$

   ```
   v' = (C - 1)/n
   C' = (v * C - 1)/η
   ```

   Note that the interface `xc = 1` is included in both regions. At `xc = 1`, `bvp4c` produces a *left* and *right* solution. These solutions are denoted as `v(1-)`, `C(1-)` and `v(1+)`, `C(1+)` respectively.

2. **Determine the boundary conditions.** Solving two first-order differential equations in two regions requires imposing four boundary conditions. Two of these conditions come from the original formulation; the others enforce the continuity of the solution across the interface `xc = 1`:

   ```
   3. v(0) = 0
   4. C(λ) - 1 = 0
   5. v(1-) - v(1+) = 0
   6. C(1-) - C(1+) = 0
   ```

   Here, `v(1-), C(1-)` and `v(1+), C(1+)` denote the left and right solution at the interface.

7. **Code the derivative function.** In the derivative function, `y(1)` corresponds to `v(x)`, and `y(2)` corresponds to `C(x)`. The additional input argument `region` identifies the region in which the derivative is evaluated. `bvp4c` enumerates regions from left to right, starting with 1. Note that the problem parameters `n` and η are shared with the outer function:

   ```
   8. function dydx = f(x,y,region)
   9.     dydx = zeros(2,1);
   10.    dydx(1) = (y(2) - 1)/n;
   11.
   12.    % The definition of C'(x) depends on the region.
   13.    switch region
   14.       case 1                                % x in [0 1]
   15.          dydx(2) = (y(1)*y(2) - x)/η;
   ```

```
    case 2                                  % x in [1 λ]
       dydx(2) = (y(1)*y(2) - 1)/η;
    end
end                 % End nested function f
```

20. **Code the boundary conditions function.** For multipoint BVPs, the arguments of the boundary conditions function, `YL` and `YR`, become matrices. In particular, the `k`th column `YL(:,k)` represents the solution at the left boundary of the `k`th region. Similarly, `YR(:,k)` represents the solution at the right boundary of the `k`th region.

    In the example, y(0) is approximated by `YL(:,1)`, while y(λ) is approximated by `YR(:,end)`. Continuity of the solution at the internal interface requires that `YR(:,1) = YL(:,2)`. Nested function `bc` computes the residual in the boundary conditions:

```
function res = bc(YL,YR)
   res = [YL(1,1)               % v(0) = 0
          YR(1,1) - YL(1,2)     % Continuity of v(x) at x=1
          YR(2,1) - YL(2,2)     % Continuity of C(x) at x=1
          YR(2,end) - 1];       % C(λ) = 1
end                             % End nested function bc
```

21. **Create an initial guess.** For multipoint BVPs, when creating an initial guess using `bvpinit`, use double entries in `xinit` for the interface point `xc`. This example uses a constant guess `yinit = [1;1]`:

```
xc = 1;
xinit = [0, 0.25, 0.5, 0.75, xc, xc, 1.25, 1.5, 1.75, 2];
solinit = bvpinit(xinit,yinit)
```

    For multipoint BVPs, you can use different guesses in different regions. To do that, you specify the initial guess for `y` as a function using the following syntax:

```
solinit = bvpinit(xinit,@yinitfcn)
```

    The initial guess function must have the following general form:

```
function y = yinitfcn(x,region)
   switch region
   case 1            % x in [0, 1]
      y = [1;1];     % initial guess for y(x) 0≤x≤1
   case 2 % x in [1, λ]
      y = [1;1];     % initial guess for y(x), 1≤x≤λ
end
```

25. **Apply the solver.** The `bvp4c` function uses the same syntax for multipoint BVPs as it does for two-point BVPs:

```
26. sol = bvp4c(@f,@bc,solinit);
```

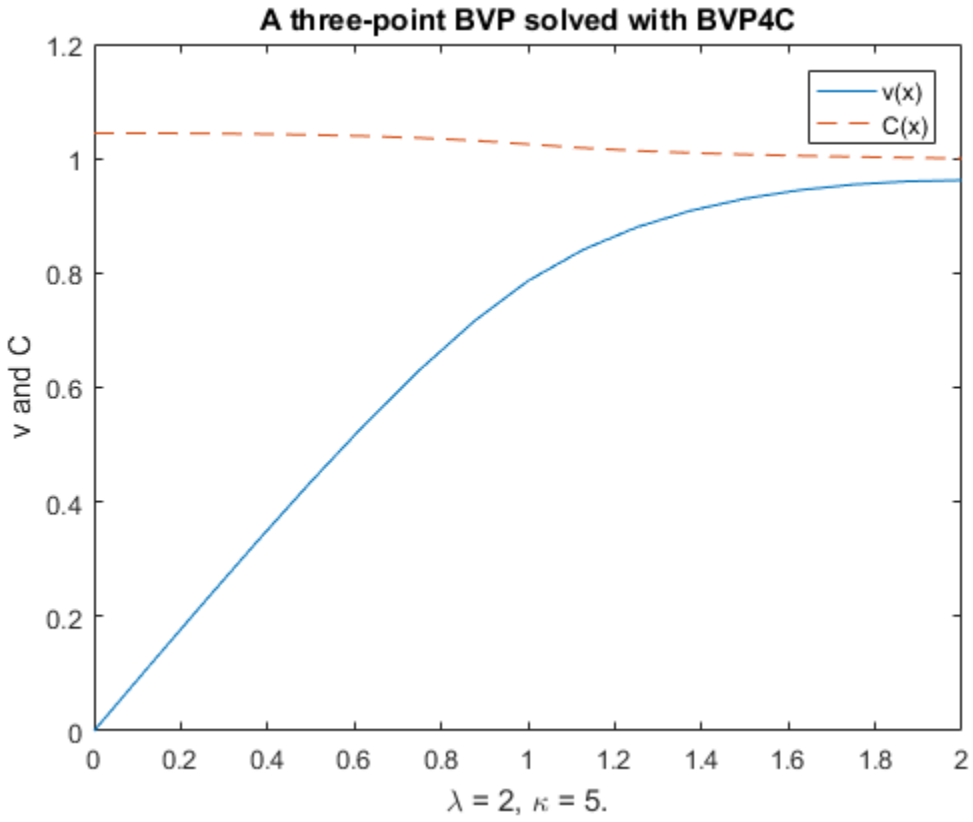
The mesh points returned in `sol.x` are adapted to the solution behavior, but the mesh still includes a double entry for the interface point `xc = 1`. Corresponding columns of `sol.y` represent the left and right solution at `xc`.

27. **View the results.** Using `deval`, the solution can be evaluated at any point in the interval of integration.

    Note that, with the left and right values computed at the interface, the solution is not uniquely defined at `xc = 1`. When evaluating the solution exactly at the interface, `deval` issues a warning and returns the average of the left and right solution values. Call `deval` at `xc-eps(xc)` and `xc+eps(xc)` to get the limit values at `xc`.

    The example plots the solution approximated at the mesh points selected by the solver:

```
plot(sol.x,sol.y(1,:),sol.x,sol.y(2,:),'--')
legend('v(x)','C(x)')
title(['A three-point BVP solved with BVP4C'])
xlabel(['\lambda = 2, \kappa = 5.'])
ylabel('v and C')
```




### Additional Examples

The following additional examples are available. Type

```
edit examplename
```

to view the code and

```
examplename
```

to run the example.

| Example Name | Description |
|---|---|
| `emdenbvp` | Emden's equation, a singular BVP |
| `fsbvp` | Falkner-Skan BVP on an infinite interval |
| `mat4bvp` | Fourth eigenfunction of Mathieu's equation |
| `shockbvp` | Solution with a shock layer near x = 0 |
| `twobvp` | BVP with exactly two solutions |
| `threebvp` | Three-point boundary value problem |

For additional examples, see Tutorial on Solving BVPs with BVP4C.

## References


[1] Ascher, U., R. Mattheij, and R. Russell, "*Numerical Solution of Boundary Value Problems for Ordinary Differential Equations*," SIAM, Philadelphia, PA, 1995, p. 372.

[2] Cebeci, T. and H. B. Keller, "*Shooting and Parallel Shooting Methods for Solving the Falkner-Skan Boundary-layer Equation*," J. Comp. Phys., Vol. 7, 1971, pp. 289-300.


## See Also

`bvp4c` | `bvp5c` | `ode45` | `pdepe`


Was this topic helpful?

**MATLAB Documentation**

- Examples
- Functions
- Release Notes
- PDF Documentation

**Other Documentation**

- Simulink
- Symbolic Math Toolbox
- Statistics and Machine Learning Toolbox
- Image Processing Toolbox
- Signal Processing Toolbox
- Documentation Home

**Support**

- MATLAB Answers
- Installation Help
- Bug Reports
- Product Requirements

- Software Downloads

© 1994-2017 The MathWorks, Inc.