NAG Toolbox

nag_ode_bvp_shoot_genpar (d02hb)

1 Purpose

nag_ode_bvp_shoot_genpar (d02hb) solves a two-point boundary value problem for a system of ordinary differential equations, using initial value techniques and Newton iteration; it generalizes function nag_ode_bvp_shoot_bval (d02ha) to include the case where parameters other than boundary values are to be determined.

2 Syntax

```
[p, soln, w, ifail] = nag_ode_bvp_shoot_genpar(p, pe, e, m1, fcn, bc, range,
'n1', n1, 'n', n)
[p, soln, w, ifail] = d02hb(p, pe, e, m1, fcn, bc, range, 'n1', n1, 'n', n)
```

3 Description

nag_ode_bvp_shoot_genpar (d02hb) solves a two-point boundary value problem by determining the unknown parameters $p_1, p_2, \ldots, p_{n_1}$ of the problem. These parameters may be, but need not be, boundary values; they may include eigenvalue parameters in the coefficients of the differential equations, length of the range of integration, etc. The notation and methods used are similar to those of nag_ode_bvp_shoot_bval (d02ha) and you are advised to study this first. (The parameters $p_1, p_2, \ldots, p_{n_1}$ correspond precisely to the unknown boundary conditions in nag_ode_bvp_shoot_bval (d02ha).) It is assumed that we have a system of n first-order ordinary differential equations of the form:

$$\frac{dy_i}{dx} = f_i(x, y_1, y_2, \ldots, y_n), \quad i = 1, 2, \ldots, n,$$

and that the derivatives f_i are evaluated by **fcn**. The system, including the boundary conditions given by **bc** and the range of integration given by **range**, involves the n_1 unknown parameters $p_1, p_2, \ldots, p_{n_1}$ which are to be determined, and for which initial estimates must be supplied. The number of unknown parameters n_1 must not exceed the number of equations n. If $n_1 < n$, we assume that $(n - n_1)$ equations of the system are not involved in the matching process. These are usually referred to as 'driving equations'; they are independent of the parameters and of the solutions of the other n_1 equations. In numbering the equations for **fcn**, the driving equations must be put **first**.

The estimated values of the parameters are corrected by a form of Newton iteration. The Newton correction on each iteration is calculated using a Jacobian matrix whose (i, j)th element depends on the derivative of the ith component of the solution, y_i, with respect to the jth parameter, p_j. This matrix is calculated by a simple numerical differentiation technique which requires n_1 evaluations of the differential system.

If the argument **ifail** is set appropriately, the function automatically prints messages to inform you of the flow of the calculation. These messages are discussed in detail in Section 9.

nag_ode_bvp_shoot_genpar (d02hb) is a simplified version of nag_ode_bvp_shoot_genpar_algeq (d02sa) which is described in detail in Gladwell (1979).

4 References

Gladwell I (1979) The development of the boundary value codes in the ordinary differential equations chapter of the NAG Library *Codes for Boundary Value Problems in Ordinary Differential Equations. Lecture Notes in Computer Science* (eds B Childs, M Scott, J W Daniel, E Denman and P Nelson) **76** Springer–Verlag

5 Parameters

You are strongly recommended to read Section 3 and Section 9 in conjunction with this section.

5.1 Compulsory Input Parameters

1: **p**(**n1**) – REAL (KIND=nag_wp) array

An estimate for the ith argument, p_i, for $i = 1, 2, \ldots, n_1$.

2: **pe**(**n1**) – REAL (KIND=nag_wp) array

The elements of **pe** must be given small positive values. The element **pe**(i) is used

(i) in the convergence test on the ith argument in the Newton iteration, and

(ii) in perturbing the ith argument when approximating the derivatives of the components of the solution with respect to this argument for use in the Newton iteration.

The elements **pe**(i) should not be chosen too small. They should usually be several orders of magnitude larger than ***machine precision***.

Constraint: **pe**$(i) > 0.0$, for $i = 1, 2, \ldots,$ **n1**.

3: **e**(**n**) – REAL (KIND=nag_wp) array

The elements of **e** must be given positive values. The element **e**(i) is used in the bound on the local error in the ith component of the solution y_i during integration.

The elements **e**(i) should not be chosen too small. They should usually be several orders of magnitude larger than ***machine precision***.

Constraint: **e**$(i) > 0.0$, for $i = 1, 2, \ldots,$ **n**.

4: **m1** – INTEGER

A value which controls exit values.

m1 $= 1$
The final solution is not calculated.

m1 > 1
The final values of the solution at interval (length of range)/(**m1** $- 1$) are calculated and stored sequentially in the array **soln** starting with the values of the solutions evaluated at the first end point (see **range**) stored in the first column of **soln**.

Constraint: **m1** ≥ 1.

5: **fcn** – SUBROUTINE, supplied by the user.

fcn must evaluate the functions f_i (i.e., the derivatives y_i'), for $i = 1, 2, \ldots, n$, at a general point x.

```
[f] = fcn(x, y, p)
```

Input Parameters

1: **x** – REAL (KIND=nag_wp)

x, the value of the argument.

2: **y**(:) – REAL (KIND=nag_wp) array

y_i, for $i = 1, 2, \ldots, n$, the value of the argument.

3: **p**(:) – REAL (KIND=nag_wp) array

The current estimate of the argument p_i, for $i = 1, 2, \ldots, n_1$.

Output Parameters

1: **f**(:) – REAL (KIND=nag_wp) array

The value of f_i, for $i = 1, 2, \ldots, n$. The f_i may depend upon the parameters p_j, for $j = 1, 2, \ldots, n_1$. If there are any driving equations (see Section 3) then these must be numbered first in the ordering of the components of **f** in **fcn**.

6: **bc** – SUBROUTINE, supplied by the user.

bc must place in **g1** and **g2** the boundary conditions at a and b respectively (see **range**).

```
[g1, g2] = bc(p)
```

Input Parameters

1: **p**(:) – REAL (KIND=nag_wp) array

An estimate of the argument p_i, for $i = 1, 2, \ldots, n_1$.

Output Parameters

1: **g1**(:) – REAL (KIND=nag_wp) array

The value of $y_i(a)$, (where this may be a known value or a function of the parameters p_j, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n_1$).

2: **g2**(:) – REAL (KIND=nag_wp) array

The value of $y_i(b)$, for $i = 1, 2, \ldots, n$, (where these may be known values or functions of the parameters p_j, for $j = 1, 2, \ldots, n_1$). If $n > n_1$, so that there are some driving equations, then the first $n - n_1$ values of **g2** need not be set since they are never used.

7: **range** – SUBROUTINE, supplied by the user.

range must evaluate the boundary points a and b, each of which may depend on the arguments $p_1, p_2, \ldots, p_{n_1}$. The integrations in the shooting method are always from a to b.

```
[a, b] = range(p)
```

Input Parameters

1: **p**(:) – REAL (KIND=nag_wp) array

The current estimate of the ith argument, p_i, for $i = 1, 2, \ldots, n_1$.

Output Parameters

1: **a** – REAL (KIND=nag_wp)

a, one of the boundary points.

2: **b** – REAL (KIND=nag_wp)

The second boundary point, b. Note that **b** $>$ **a** forces the direction of integration to be that of increasing x. If **a** and **b** are interchanged the direction of integration is reversed.

5.2 Optional Input Parameters

1: **n1** – INTEGER

Default: the dimension of the arrays **p**, **pe**. (An error is raised if these dimensions are not equal.)

n_1, the number of arguments.

Constraint: $1 \leq \mathbf{n1} \leq \mathbf{n}$.

2: **n** – INTEGER

Default: the dimension of the array **e**.

n, the total number of differential equations.

Constraint: $\mathbf{n} \geq \mathbf{n1}$.

5.3 Output Parameters

1: **p**(**n1**) – REAL (KIND=nag_wp) array

The corrected value for the ith argument, unless an error has occurred, when it contains the last calculated value of the argument.

2: **soln**(**n**, **m1**) – REAL (KIND=nag_wp) array

The solution when $\mathbf{m1} > 1$.

3: **w**(**n**, sdw) – REAL (KIND=nag_wp) array

$sdw = 3\mathbf{n} + 14 + \max(11, \mathbf{n})$.

With **ifail** $= 2$, 3, 4 or 5 (see Section 6), $\mathbf{w}(i, 1)$, for $i = 1, 2, \ldots, n$, contains the solution at the point x when the error occurred. $\mathbf{w}(1, 2)$ contains x.

4: **ifail** – INTEGER

ifail $= 0$ unless the function detects an error (see Section 5).

6 Error Indicators and Warnings

Errors or warnings detected by the function:

ifail $= 1$

One or more of the arguments **n**, **n1**, **m1**, sdw, **e** or **pe** is incorrectly set.

ifail $= 2$

The step length for the integration became too short whilst calculating the residual (see Section 9).

ifail $= 3$

No initial step length could be chosen for the integration whilst calculating the residual.

Note: **ifail** $= 2$ or 3 can occur due to choosing too small a value for **e** or due to choosing the wrong direction of integration. Try varying **e** and interchanging a and b. These error exits can also occur for very poor initial choices of the parameters in the array **p** and, in extreme cases, because nag_ode_bvp_shoot_genpar (d02hb) cannot be used to solve the problem posed.

ifail $= 4$

As for **ifail** $= 2$ but the error occurred when calculating the Jacobian.

ifail $= 5$

As for **ifail** $= 3$ but the error occurred when calculating the Jacobian.

ifail $= 6$

The calculated Jacobian has an insignificant column. This can occur because a parameter p_i is incorrectly entered when posing the problem.

Note: **ifail** $= 4$, 5 or 6 usually indicate a badly scaled problem. You may vary the size of **pe**. Otherwise the use of the more general nag_ode_bvp_shoot_genpar_algeq (d02sa) which affords more control over the calculations is advised.

ifail $= 7$

The linear algebra function used (nag_lapack_dgesvd (f08kb)) has failed. This error exit should not occur and can be avoided by changing the initial estimates p_i.

ifail $= 8$

The Newton iteration has failed to converge. This can indicate a poor initial choice of parameters p_i or a very difficult problem. Consider varying the elements **pe**(i) if the residuals are small in the monitoring output. If the residuals are large, try varying the initial parameters p_i.

ifail $= 9$
ifail $= 10$
ifail $= 11$
ifail $= 12$
ifail $= 13$

Indicates that a serious error has occurred. Check all array subscripts and function argument lists in the call to nag_ode_bvp_shoot_genpar (d02hb). Seek expert help.

ifail $= -99$

An unexpected error has been triggered by this routine. Please contact NAG.

ifail $= -399$

Your licence key may have expired or may not have been installed correctly.

ifail $= -999$

Dynamic memory allocation failed.

7 Accuracy

If the process converges, the accuracy to which the unknown parameters are determined is usually close to that specified by you; the solution, if requested, may be determined to a required accuracy by varying **e**.

8 Further Comments

The time taken by nag_ode_bvp_shoot_genpar (d02hb) depends on the complexity of the system, and on the number of iterations required. In practice, integration of the differential equations is by far the most costly process involved.

Wherever they occur in the function, the error arguments contained in the arrays **e** and **pe** are used in 'mixed' form; that is **e**(i) always occurs in expressions of the form

$$\mathbf{e}(i) \times (1 + |y_i|)$$

and **pe**(i) always occurs in expressions of the form

$$\mathbf{pe}(i) \times (1 + |p_i|).$$

Though not ideal for every application, it is expected that this mixture of absolute and relative error testing will be adequate for most purposes.

You may determine a suitable direction of integration a to b and suitable values for $\mathbf{e}(i)$ by integrations with nag_ode_ivp_rkts_range (d02pe). The best direction of integration is usually the direction of decreasing solutions.

You are strongly recommended to set **ifail** to obtain self-explanatory error messages, and also monitoring information about the course of the computation. You may select the channel numbers on which this output is to appear by calls of nag_file_set_unit_error (x04aa) (for error messages) or nag_file_set_unit_advisory (x04ab) (for monitoring information) – see Section 10 for an example. Otherwise the default channel numbers will be used. The monitoring information produced at each iteration includes the current parameter values, the residuals and 2-norms: a basic norm and a current norm. At each iteration the aim is to find parameter values which make the current norm less than the basic norm. Both these norms should tend to zero as should the residuals. (They would all be zero if the exact parameters were used as input.) For more details, in particular about the other monitoring information printed, you are advised to consult the specification of nag_ode_bvp_shoot_genpar_algeq (d02sa), and especially the description of the argument **monit** there.

The computing time for integrating the differential equations can sometimes depend critically on the quality of the initial estimates for the parameters p_i. If it seems that too much computing time is required and, in particular, if the values of the residuals printed by the monitoring function are much larger than the expected values of the solution at b, then the coding of **fcn**, **bc** and **range** should be checked for errors. If no errors can be found, an independent attempt should be made to improve the initial estimates for p_i.

The function can be used to solve a very wide range of problems, for example:

(a) eigenvalue problems, including problems where the eigenvalue occurs in the boundary conditions;

(b) problems where the differential equations depend on some parameters which are to be determined so as to satisfy certain boundary conditions (see Example 2 in Section 10);

(c) problems where one of the end points of the range of integration is to be determined as the point where a variable y_i takes a particular value (see Example 2 in Section 10);

(d) singular problems and problems on infinite ranges of integration where the values of the solution at a or b or both are determined by a power series or an asymptotic expansion (or a more complicated expression) and where some of the coefficients in the expression are to be determined (see Example 1 in Section 10); and

(e) differential equations with certain terms defined by other independent (driving) differential equations.

9 Example

For this function two examples are presented. There is a single example program for nag_ode_bvp_shoot_genpar (d02hb), with a main program and the code to solve the two example problems given in Example 1 (EX1) and Example 2 (EX2).

Example 1 (EX1)

This example finds the solution of the differential equation

$$y'' = \left(y^3 - y'\right)/2x$$

on the range $0 \leq x \leq 16$, with boundary conditions $y(0) = 0.1$ and $y(16) = 1/6$. We cannot use the differential equation at $x = 0$ because it is singular, so we take a truncated power series expansion

$$y(x) = 1/10 + p_1 \times \sqrt{x}/10 + x/100$$

near the origin where p_1 is one of the parameters to be determined. We choose the interval as $[0.1, 16]$

and setting $p_2 = y'(16)$, we can determine all the boundary conditions. We take X1 = 16. We write $y = \mathbf{y}(1)$, $y' = \mathbf{y}(2)$, and estimate PARAM(1) = 0.2, PARAM(2) = 0.0. Note the call to nag_file_set_unit_advisory (x04ab) before the call to nag_ode_bvp_shoot_genpar (d02hb).

Example 2 (EX2)

This example finds the gravitational constant p_1 and the range p_2 over which a projectile must be fired to hit the target with a given velocity.

The differential equations are

$$y' = \tan\phi$$
$$v' = \frac{-\left(p_1 \sin\phi + 0.00002v^2\right)}{v\cos\phi}$$
$$\phi\prime = \frac{-p_1}{v^2}$$

on the range $0 < x < p_2$, with boundary conditions

$$y = 0, \quad v = 500, \quad \phi = 0.5 \quad \text{at} \quad x = 0,$$
$$y = 0, \quad v = 450, \quad \phi = p_3 \quad \text{at} \quad x = p_2.$$

We write $y = \mathbf{y}(1)$, $v = \mathbf{y}(2)$, $\phi = \mathbf{y}(3)$. We estimate $p_1 = \text{PARAM}(1) = 32$, $p_2 = \text{PARAM}(2) = 6000$ and $p_3 = \text{PARAM}(3) = 0.54$ (though this last estimate is not important).

9.1 Program Text

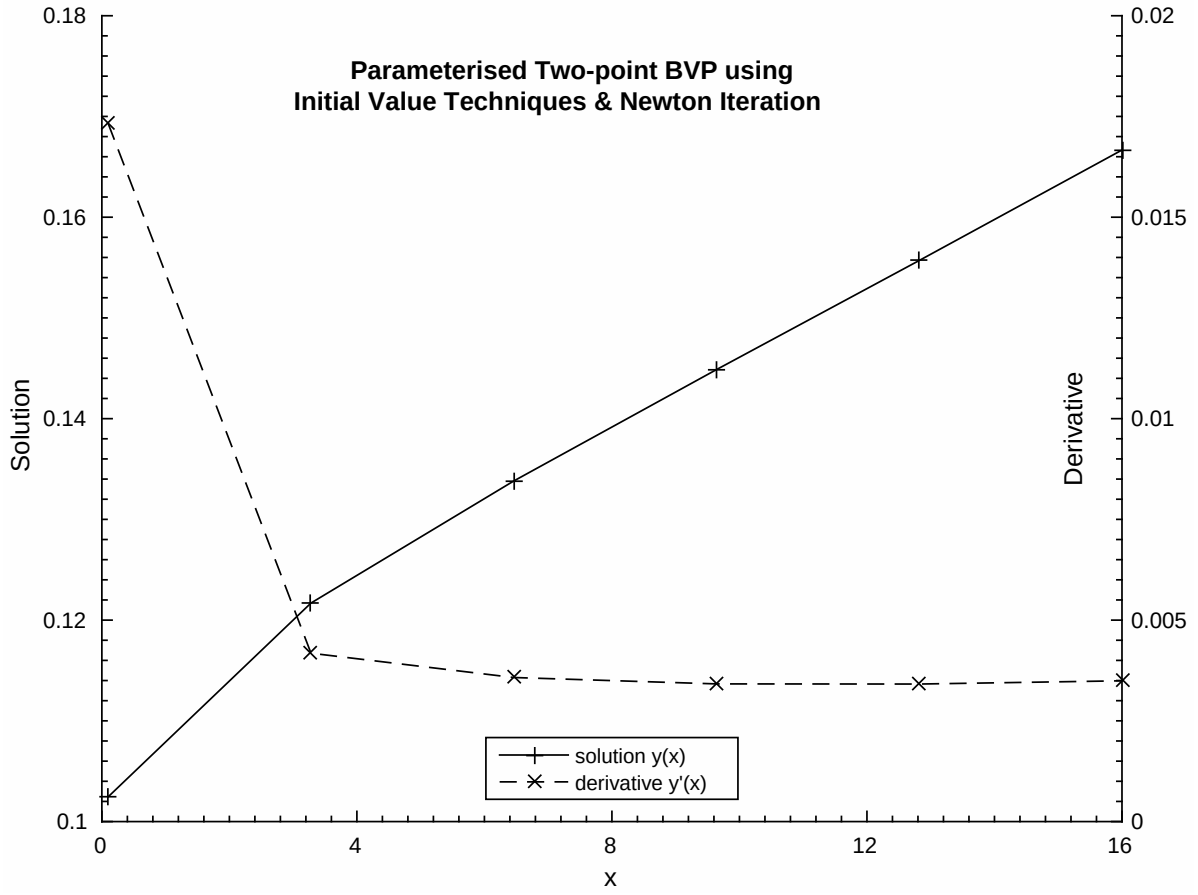
```
     function d02hb_example

fprintf('d02hb example results\n\n');

% Set up initial values for first case.
n = nag_int(2);
n1 = nag_int(2);
p = [0.2;0];
pe = [1e-05;0.001];
e = [0.0001;0.0001];
m1 = nag_int(6);
marray = zeros(1,m1);

fprintf('d02hb example program results \n');

fprintf('\nCase 1 \n\n');

[pOut1, solnOut1, w1, ifail] = ...
  d02hb(...
         p, pe, e, m1, @fcn1, @bc1, @range1, 'n1', n1, 'n', n);

% Output results.
fprintf('Final parameters \n')
fprintf('    %1.3e   ',pOut1);
fprintf('\n\n Final solution \n')
fprintf('X-value     Components of solution \n')

[x0, x1] = range1(p);
h = (x1-x0)/double(m1-1);
for i = 1:m1;
  m = x0 + double(i-1)*h;
  fprintf('%7.3f', m);
  fprintf('%12.4f',solnOut1(1:n,i));
  fprintf('\n');
  % Save results for plotting.
  marray(i) = m;
end

fprintf('\nCase 2 \n\n');
% Set up initial values for second case.
n = nag_int(3);
n1 = n;
```

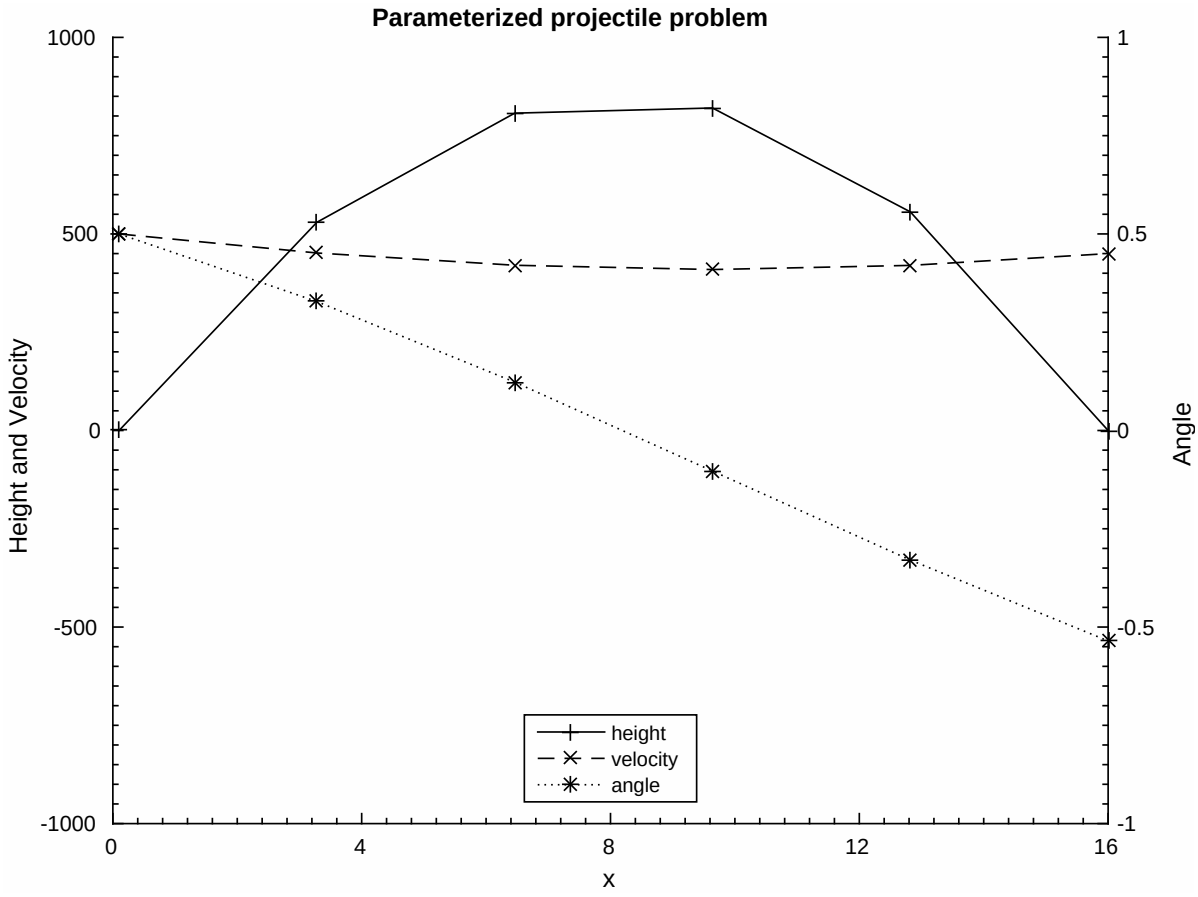
```
p = [32;6000;0.54];
pe = [1e-05;1e-4;1e-4];
e = [1e-2;1e-2;1e-2];
m1 = nag_int(6);
qarray = zeros(1,m1);

[pOut2, solnOut2, w2, ifail] = ...
  d02hb(...
         p, pe, e, nag_int(m1), @fcn2, @bc2, @range2, 'n1', n1, 'n', n);
% Output results.
fprintf('Final parameters \n')
fprintf('    %1.3e   ',pOut2);
fprintf('\n\n Final solution \n')
fprintf(' X-value        Components of solution \n')

[x0, x1] = range2(pOut2);
h = (x1-x0)/double(m1-1);
for i = 1:m1;
  q = x0 + double(i-1)*h;
  fprintf('%6.0f', q);
  fprintf('%12.4f',solnOut2(1:n,i));
  fprintf('\n');
  % Save results for plotting.
  qarray(i) = q;
end

% Plot results.
fig1 = figure;
display_plot(marray, solnOut1, 'Derivative');

fig2 = figure;
display_plot(marray, solnOut2, 'Angle');

function [g1, g2] = bc1(p)
  % Evaluate boundary conditions.
  z=0.1;
  g1(1) = 0.1+p(1)*sqrt(z)*0.1+0.01*z;
  g1(2) = p(1)*0.05/sqrt(z) + 0.01;
  g2(1) = 1/6;
  g2(2) = p(2);

function [g1, g2] = bc2(p)
  % Evaluate boundary conditions.
  g1(1) = 0.0;
  g1(2) = 500.0;
  g1(3) = 0.5;
  g2(1) = 0.0;
  g2(2) = 450.0;
  g2(3) = p(3);

function f = fcn1(x, y, p)
  % Evaluate derivatives.
  f = zeros(2,1);
  f(1) = y(2);
  f(2) = (y(1)^3-y(2))/2/x;

function f = fcn2(x, y, p)
  % Evaluate derivatives.
  f = zeros(3,1);
  f(1) = tan(y(3));
  f(2) = -p(1)*tan(y(3))/y(2) - 0.00002*y(2)/cos(y(3));
  f(3) = -p(1)/y(2)^2;

function [x0, x1] = range1(p)
  % Evaluate boundary points.
  x0 = 0.1;
  x1 = 16.0;

function [x0, x1] = range2(p)
  % Evaluate boundary points.
  x0 = 0.0;
```

```
  x1 = p(2);

function display_plot(x, y, ylabelString)
  % Choose the type of plot based on the y axis label.
  if strncmp(ylabelString, 'Derivative', 8)
    % Plot the results as a linear graph (2 y axes).
    [haxes, hline1, hline2] = plotyy(x, y(1,:), x, y(2,:));
    % Set the axis limits and the tick specifications to beautify the plot.
    set(haxes(1), 'YLim', [0.1 0.18]);
    set(haxes(1), 'XMinorTick', 'on', 'YMinorTick', 'on');
    set(haxes(1), 'YTick', [0.1 0.12 0.14 0.16 0.18]);
    set(haxes(2), 'YLim', [0 0.02]);
    set(haxes(2), 'YMinorTick', 'on');
    set(haxes(2), 'YTick', [0:0.005:0.02]);
    for iaxis = 1:2
      % These properties must be the same for both sets of axes.
      set(haxes(iaxis), 'XLim', [0 16]);
      set(haxes(iaxis), 'XTick', [0:4:16]);
    end
    set(gca, 'box', 'off');
    % Set the title.
    title({['Parameterised Two-point BVP using'], ...
           ['Initial Value Techniques & Newton Iteration']}, ...
          'position',[8,0.17]);
    % Label the axes.
    xlabel('x');
    ylabel(haxes(1),'Solution');
    yh2 = ylabel(haxes(2),ylabelString);
    set(yh2,'position',[15,0.01]);
    % Add legend.
    legend('solution y(x)','derivative y''(x)','Location','South');
    % Set some features of the two lines.
    set(hline1, 'Linewidth', 0.5, 'Marker', '+', 'LineStyle', '-');
    set(hline2, 'Linewidth', 0.5, 'Marker', 'x', 'LineStyle', '--');
  else
    % Plot the results as a linear graph (2 y axes).
    [haxes, hline1, hline2] = plotyy(x, y(1,:), x, y(3,:));
    % Add a third curve,
    hold on
    hline3 = plot(x, y(2,:),'Color','Magenta');
    % Set the axis limits and the tick specifications to beautify the plot.
    set(haxes(1), 'YLim', [-1000 1000]);
    set(haxes(1), 'XMinorTick', 'on', 'YMinorTick', 'on');
    set(haxes(1), 'YTick', [-1000:500:1000]);
    set(haxes(2), 'YLim', [-1 1]);
    set(haxes(2), 'YMinorTick', 'on');
    set(haxes(2), 'YTick', [-1:0.5:1]);
    for iaxis = 1:2
      % These properties must be the same for both sets of axes.
      set(haxes(iaxis), 'XLim', [0 16]);
      set(haxes(iaxis), 'XTick', [0:4:16]);
    end
    set(gca, 'box', 'off');
    % Set the title.
    title('Parameterized projectile problem');
    % Label the axes.
    xlabel('x');
    ylabel(haxes(1),'Height and Velocity');
    ylabel(haxes(2),ylabelString);
    % Add legend.
    legend('height','velocity','angle','Location','South');
    % Set some features of the three lines.
    set(hline1, 'Linewidth', 0.5, 'Marker', '+', 'LineStyle', '-');
    set(hline2, 'Linewidth', 0.5, 'Marker', '*', 'LineStyle', ':');
    set(hline3, 'Linewidth', 0.5, 'Marker', 'x', 'LineStyle', '--');
end
```

9.2 Program Results

```
     d02hb example results

d02hb example program results

Case 1

Final parameters
    4.629e-02       3.494e-03

 Final solution
X-value     Components of solution
  0.100      0.1025      0.0173
  3.280      0.1217      0.0042
  6.460      0.1338      0.0036
  9.640      0.1449      0.0034
 12.820      0.1557      0.0034
 16.000      0.1667      0.0035

Case 2

Final parameters
    3.239e+01       5.962e+03       -5.353e-01

 Final solution
 X-value        Components of solution
     0      0.0000    500.0000      0.5000
  1192    529.6471    451.5554      0.3280
  2385    807.2140    420.3050      0.1231
  3577    820.3972    409.4254     -0.1032
  4769    556.1322    419.9890     -0.3296
  5962     -0.0003    450.0000     -0.5353
```




Parameterized projectile problem
Height and Velocity
Angle
x
1000
500
0
-500
-1000
1
0.5
0
-0.5
-1
0
4
8
12
16
height
velocity
angle