NAG Toolbox

nag_ode_ivp_bdf_zero_simple (d02ej)

1 Purpose

nag_ode_ivp_bdf_zero_simple (d02ej) integrates a stiff system of first-order ordinary differential equations over an interval with suitable initial conditions, using a variable-order, variable-step method implementing the Backward Differentiation Formulae (BDF), until a user-specified function, if supplied, of the solution is zero, and returns the solution at points specified by you, if desired.

2 Syntax

```
[x, y, tol, ifail] = nag_ode_ivp_bdf_zero_simple(x, xend, y, fcn, pederv, tol, relabs, output, g, 'n', n)
[x, y, tol, ifail] = d02ej(x, xend, y, fcn, pederv, tol, relabs, output, g, 'n', n)
```

3 Description

nag_ode_ivp_bdf_zero_simple (d02ej) advances the solution of a system of ordinary differential equations

$$y_i' = f_i(x, y_1, y_2, \ldots, y_n), \quad i = 1, 2, \ldots, n,$$

from $x = \mathbf{x}$ to $x = \mathbf{xend}$ using a variable-order, variable-step method implementing the BDF. The system is defined by **fcn**, which evaluates f_i in terms of x and $y_1, y_2, \ldots, y_n$ (see Section 5). The initial values of $y_1, y_2, \ldots, y_n$ must be given at $x = \mathbf{x}$.

The solution is returned via the **output** at points specified by you, if desired: this solution is obtained by C^1 interpolation on solution values produced by the method. As the integration proceeds a check can be made on the user-specified function $g(x, y)$ to determine an interval where it changes sign. The position of this sign change is then determined accurately by C^1 interpolation to the solution. It is assumed that $g(x, y)$ is a continuous function of the variables, so that a solution of $g(x, y) = 0.0$ can be determined by searching for a change in sign in $g(x, y)$. The accuracy of the integration, the interpolation and, indirectly, of the determination of the position where $g(x, y) = 0.0$, is controlled by the arguments **tol** and **relabs**. The Jacobian of the system $y' = f(x, y)$ may be supplied in **pederv**, if it is available.

For a description of BDF and their practical implementation see Hall and Watt (1976).

4 References

Hall G and Watt J M (ed.) (1976) *Modern Numerical Methods for Ordinary Differential Equations* Clarendon Press, Oxford

5 Parameters

5.1 Compulsory Input Parameters

1: **x** – REAL (KIND=nag_wp)

The initial value of the independent variable x.

Constraint: $\mathbf{x} \neq \mathbf{xend}$.

2: **xend** – REAL (KIND=nag_wp)

The final value of the independent variable. If **xend** $<$ **x**, integration will proceed in the negative direction.

Constraint: **xend** $\neq$ **x**.

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

The initial values of the solution $y_1, y_2, \ldots, y_n$ at $x =$ **x**.

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

fcn must evaluate the functions f_i (i.e., the derivatives y_i') for given values of its arguments $x, y_1, \ldots, y_n$.

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

Input Parameters

1: **x** – REAL (KIND=nag_wp)

x, the value of the independent variable.

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

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

Output Parameters

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

The value of f_i, for $i = 1, 2, \ldots, n$.

5: **pederv** – SUBROUTINE, supplied by the NAG Library or the user.

pederv must evaluate the Jacobian of the system (that is, the partial derivatives $\frac{\partial f_i}{\partial y_j}$) for given values of the variables $x, y_1, y_2, \ldots, y_n$.

```
[pw] = pederv(x, y)
```

Input Parameters

1: **x** – REAL (KIND=nag_wp)

x, the value of the independent variable.

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

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

Output Parameters

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

pw$(n \times (i-1) + j)$ must contain the value of $\frac{\partial f_i}{\partial y_j}$, for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n$.

If you do not wish to supply the Jacobian, the actual argument **pederv must** be the string `nag_ode_ivp_bdf_zero_simple_dummy_pederv (d02ejy)`. (nag_ode_ivp_bdf_zero_simple_dummy_pederv (d02ejy) is included in the NAG Toolbox.)

6: **tol** – REAL (KIND=nag_wp)

Must be set to a **positive** tolerance for controlling the error in the integration. Hence **tol** affects the determination of the position where $g(x, y) = 0.0$, if **g** is supplied.

nag_ode_ivp_bdf_zero_simple (d02ej) has been designed so that, for most problems, a reduction in **tol** leads to an approximately proportional reduction in the error in the solution. However, the actual relation between **tol** and the accuracy achieved cannot be guaranteed. You are strongly recommended to call nag_ode_ivp_bdf_zero_simple (d02ej) with more than one value for **tol** and to compare the results obtained to estimate their accuracy. In the absence of any prior knowledge, you might compare the results obtained by calling nag_ode_ivp_bdf_zero_simple (d02ej) with **tol** $= 10^{-p}$ and **tol** $= 10^{-p-1}$ if p correct decimal digits are required in the solution.

Constraint: **tol** > 0.0.

7: **relabs** – CHARACTER(1)

The type of error control. At each step in the numerical solution an estimate of the local error, est, is made. For the current step to be accepted the following condition must be satisfied:

$$est = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(e_i/(\tau_r \times |y_i| + \tau_a)\right)^2} \leq 1.0$$

where τ_r and τ_a are defined by

relabs	τ_r	τ_a
`M'	**tol**	**tol**
`A'	0.0	**tol**
`R'	**tol**	ϵ
`D'	**tol**	ϵ

where ϵ is a small machine-dependent number and e_i is an estimate of the local error at y_i, computed internally. If the appropriate condition is not satisfied, the step size is reduced and the solution is recomputed on the current step. If you wish to measure the error in the computed solution in terms of the number of correct decimal places, then **relabs** should be set to `A' on entry, whereas if the error requirement is in terms of the number of correct significant digits, then **relabs** should be set to `R'. If you prefer a mixed error test, then **relabs** should be set to `M', otherwise if you have no preference, **relabs** should be set to the default `D'. Note that in this case `D' is taken to be `R'.

Constraint: **relabs** = 'A', 'M', 'R' or 'D'.

8: **output** – SUBROUTINE, supplied by the NAG Library or the user.

output permits access to intermediate values of the computed solution (for example to print or plot them), at successive user-specified points. It is initially called by nag_ode_ivp_bdf_zero_simple (d02ej) with **xsol** = **x** (the initial value of x). You must reset **xsol** to the next point (between the current **xsol** and **xend**) where **output** is to be called, and so on at each call to **output**. If, after a call to **output**, the reset point **xsol** is beyond **xend**, nag_ode_ivp_bdf_zero_simple (d02ej) will integrate to **xend** with no further calls to **output**; if a call to **output** is required at the point **xsol** = **xend**, then **xsol** must be given precisely the value **xend**.

```
[xsol] = output(xsol, y)
```

Input Parameters

1: **xsol** – REAL (KIND=nag_wp)

x, the value of the independent variable.

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

The computed solution at the point **xsol**.

Output Parameters

1: **xsol** – REAL (KIND=nag_wp)

You must set **xsol** to the next value of x at which **output** is to be called.

If you do not wish to access intermediate output, the actual argument **output must** be the string `nag_ode_ivp_bdf_zero_simple_dummy_output (d02ejx)`. (nag_ode_ivp_bdf_zero_simple_dummy_output (d02ejx) is included in the NAG Toolbox.)

9: **g** – REAL (KIND=nag_wp) FUNCTION, supplied by the user.

g must evaluate the function $g(x,y)$ for specified values x,y. It specifies the function g for which the first position x where $g(x,y) = 0$ is to be found.

```
[result] = g(x, y)
```

Input Parameters

1: **x** – REAL (KIND=nag_wp)

x, the value of the independent variable.

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

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

Output Parameters

1: **result**

The value of $g(x,y)$ at the specified values x,y.

If you do not require the root-finding option, the actual argument **g must** be the string `nag_ode_ivp_bdf_zero_simple_dummy_g (d02ejw)`. (nag_ode_ivp_bdf_zero_simple_dummy_g (d02ejw) is included in the NAG Toolbox.)

5.2 Optional Input Parameters

1: **n** – INTEGER

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

n, the number of differential equations.

Constraint: **n** ≥ 1.

5.3 Output Parameters

1: **x** – REAL (KIND=nag_wp)

If **g** is supplied by you, **x** contains the point where $g(x,y) = 0.0$, unless $g(x,y) \neq 0.0$ anywhere on the range **x** to **xend**, in which case, **x** will contain **xend**. If **g** is not supplied **x** contains **xend**, unless an error has occurred, when it contains the value of x at the error.

2: **y**$(\mathbf{n})$ – REAL (KIND=nag_wp) array

The computed values of the solution at the final point $x =$ **x**.

3: **tol** – REAL (KIND=nag_wp)

Normally unchanged. However if the range **x** to **xend** is so short that a small change in **tol** is unlikely to make any change in the computed solution, then, on return, **tol** has its sign changed.

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 $= 1$

On entry, **tol** ≤ 0.0,
or **x** $=$ **xend**,
or **n** ≤ 0,
or **relabs** $\neq$ 'M','A','R','D',
or $iw < (12 + \mathbf{n}) \times \mathbf{n} + 50$.

ifail $= 2$

With the given value of **tol**, no further progress can be made across the integration range from the current point $x =$ **x**. (See Section 5 for a discussion of this error test.) The components $\mathbf{y}(1), \mathbf{y}(2), \ldots, \mathbf{y}(\mathbf{n})$ contain the computed values of the solution at the current point $x =$ **x**. If you have supplied **g**, then no point at which $g(x, y)$ changes sign has been located up to the point $x =$ **x**.

ifail $= 3$

tol is too small for nag_ode_ivp_bdf_zero_simple (d02ej) to take an initial step. **x** and $\mathbf{y}(1), \mathbf{y}(2), \ldots, \mathbf{y}(\mathbf{n})$ retain their initial values.

ifail $= 4$

xsol lies behind **x** in the direction of integration, after the initial call to **output**, if the **output** option was selected.

ifail $= 5$

A value of **xsol** returned by the **output** lies behind the last value of **xsol** in the direction of integration, if the **output** option was selected.

ifail $= 6$

At no point in the range **x** to **xend** did the function $g(x, y)$ change sign, if g was supplied. It is assumed that $g(x, y) = 0$ has no solution.

ifail $= 7$ (nag_roots_contfn_brent_rcomm (c05az))

A serious error has occurred in an internal call to the specified function. Check all function calls and array dimensions. Seek expert help.

ifail $= 8$ (nag_ode_ivp_stiff_c1_interp (d02xk))

A serious error has occurred in an internal call to the specified function. Check all function calls and array dimensions. Seek expert help.

ifail $= 9$

A serious error has occurred in an internal call to an interpolation function. Check all (sub) program calls and array dimensions. 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

The accuracy of the computation of the solution vector **y** may be controlled by varying the local error tolerance **tol**. In general, a decrease in local error tolerance should lead to an increase in accuracy. You are advised to choose **relabs** = 'R' unless you have a good reason for a different choice. It is particularly appropriate if the solution decays.

If the problem is a root-finding one, then the accuracy of the root determined will depend strongly on $\frac{\partial g}{\partial x}$ and $\frac{\partial g}{\partial y_i}$, for $i = 1, 2, \ldots, n$. Large values for these quantities may imply large errors in the root.

8 Further Comments

If more than one root is required, then to determine the second and later roots nag_ode_ivp_bdf_zero_simple (d02ej) may be called again starting a short distance past the previously determined roots. Alternatively you may construct your own root-finding code using nag_ode_ivp_stiff_exp_fulljac (d02nb) (and other functions in Sub-chapter D02M–N), nag_roots_contfn_brent_rcomm (c05az) and nag_ode_ivp_stiff_c1_interp (d02xk).

If it is easy to code, you should supply **pederv**. However, it is important to be aware that if **pederv** is coded incorrectly, a very inefficient integration may result and possibly even a failure to complete the integration (see **ifail** $= 2$).

9 Example

We illustrate the solution of five different problems. In each case the differential system is the well-known stiff Robertson problem.

$$\begin{aligned} a' &= -0.04a + 10^4 bc \\ b' &= 0.04a - 10^4 bc - 3 \times 10^7 b^2 \\ c' &= 3 \times 10^7 b^2 \end{aligned}$$

with initial conditions $a = 1.0$, $b = c = 0.0$ at $x = 0.0$. We solve each of the following problems with local error tolerances 1.0e−3 and 1.0e−4.

(i) To integrate to $x = 10.0$ producing output at intervals of 2.0 until a point is encountered where $a = 0.9$. The Jacobian is calculated numerically.

(ii) As (i) but with the Jacobian calculated analytically.

(iii) As (i) but with no intermediate output.

(iv) As (i) but with no termination on a root-finding condition.

(v) Integrating the equations as in (i) but with no intermediate output and no root-finding termination condition.

9.1 Program Text

```
     function d02ej_example

fprintf('d02ej example results\n\n');

% For communication with save.
global ykeep ncall xkeep;

% Initialize variables and arrays.
x = 0;
xend = 10;
y = [1; 0; 0];
tol = 0.001;
relabs = 'Default';

ncall = 0;
ykeep = zeros(1,length(y));
xkeep = zeros(1,1);

disp('Case 1: calculating Jacobian internally,');
fprintf('intermediate output, root-finding\n\n');
for j = 3:4
  tol = double(10)^(-j);
  disp(['Calculation with tol = ',num2str(tol)]);
  disp(' X     Y(1)        Y(2)        Y(3)');

  % output outputs intermediate values.
  [xOut, yOut, tolOut, ifail] = ...
    d02ej(...
         x, xend, y, @fcn,'d02ejy', tol, relabs, @output, @g);

  disp(' ');
  disp(['Root of Y(1)-0.9 = 0.0 at ',num2str(xOut)]);
  disp('Solution is' );
  fprintf('  %8.4f   %8.4f   %8.4f\n\n', yOut);
  if (tol < 0.0)
    disp('Range too short for tol');
  end
end

disp('Case 2: calculating Jacobian by pederv,');
fprintf('intermediate output, root-finding\n\n');
for j = 3:4
  tol = double(10)^(-j);
  disp(['Calculation with tol = ',num2str(tol)]);
  disp(' X     Y(1)        Y(2)        Y(3)');

  % pederv evaluates Jacobian.
  [xOut, yOut, tolOut, ifail] = ...
    d02ej(...
          x, xend, y, @fcn, @pederv, tol, relabs, @output, @g);
  disp(' ');
  disp(['Root of Y(1)-0.9 = 0.0 at ',num2str(xOut)]);
  disp('Solution is' );
  fprintf('  %8.4f   %8.4f   %8.4f\n\n', yOut);
  xOut1 = xOut;
  if (tol < 0.0)
    disp('Range too short for tol');
  end

  % Store the x value for plotting.
  xOut1 = xOut;
end

disp('Case 3: calculating Jacobian internally,');
fprintf('no intermediate output, root-finding\n\n');
for j = 3:4
  tol = double(10)^(-j);
  disp(['Calculation with tol = ',num2str(tol)]);
```

```
  [xOut, yOut, tolOut, ifail] = ...
    d02ej(...
           x, xend, y, @fcn, 'd02ejy', tol, relabs, 'd02ejx', @g);
  disp(' ');
  disp(['Root of Y(1)-0.9 = 0.0 at ',num2str(xOut)]);
  disp('Solution is' );
  fprintf('  %8.4f   %8.4f   %8.4f\n\n', yOut);
  if (tol < 0.0)
    disp('Range too short for tol');
  end
end

disp('Case 4: calculating Jacobian internally,');
fprintf('intermediate output, no root-finding\n\n');
for j = 3:4
  tol = double(10)^(-j);
  disp(['Calculation with tol = ',num2str(tol)]);
  disp(' X     Y(1)       Y(2)       Y(3)');

  % save stores intermediate values in xkeep, ykeep, which are
  % plotted later (it also outputs them).
  [xOut, yOut, tolOut, ifail] = ...
    d02ej(...
           x, xend, y, @fcn, 'd02ejy', tol, relabs, @save, 'd02ejw');
  disp(' ');
  if (tol < 0.0)
    disp('Range too short for tol');
  end
end

disp('Case 5: calculating Jacobian internally,');
fprintf('no intermediate output, no root-finding (integrate to xend)\n\n');
for j = 3:4
  tol = double(10)^(-j);
  disp(['Calculation with tol = ',num2str(tol)]);
  disp(' X     Y(1)        Y(2)        Y(3)');
  fprintf('%d    %8.6f    %8.6f    %8.6f\n',x,y);

  [xOut, yOut, tolOut, ifail] = ...
    d02ej(...
           x, xend, y, @fcn, 'd02ejy', tol, relabs,'d02ejx', 'd02ejw');

  fprintf('%2d   %8.6f    %8.6f    %8.6f\n\n', xOut, yOut);
  if (tol < 0)
    disp('Range too short for tol');
  end
end
% Plot results.
nres = 0.5*length(xkeep);
xplot = xkeep(nres+1:2*nres);
yplot = ykeep(nres+1:2*nres, :);
fig1 = figure;
display_plot(xplot, yplot, xOut1);

function xsolOut = save(xsol, y)
  % For communication with main routine.
  global ykeep ncall xkeep;

  % This version of the intermediate output routine stores the values
  % (so they can be plotted in the main routine).
  ncall = ncall+1;
  ykeep(ncall,:) = y;
  xkeep(ncall,:) = xsol;
  fprintf('%2d    %8.6f    %8.6f    %8.6f\n', xsol, y);
  xsolOut = xsol + 2;

function xsolOut = output(xsol, y)
  % Output intermediate values of solution.
  fprintf('%2d    %8.6f    %8.6f    %8.6f\n', xsol, y);
  xsolOut = xsol + 2;
```

```
function f = fcn(x, y)
  % Evaluate the derivatives.
  f = zeros(3,1);
  f(1) = -0.04*y(1) + 1.0d4*y(2)*y(3);
  f(2) = 0.04*y(1) - 1.0d4*y(2)*y(3) - 3.0d7*y(2)*y(2);
  f(3) = 3.0d7*y(2)*y(2);

function result = g(x, y)
  % Evaluate g(x,y) when root-finding option is selected.
  result = y(1) - 0.9;

function pw = pederv(x, y)
  % Evaluate the Jacobian.
  pw = zeros(3,3);
  pw(1,1) = -0.04d0;
  pw(1,2) = 1.0d4*y(3);
  pw(1,3) = 1.0d4*y(2);
  pw(2,1) = 0.04d0;
  pw(2,2) = -1.0d4*y(3) - 6.0d7*y(2);
  pw(2,3) = -1.0d4*y(2);
  pw(3,1) = 0.0d0;
  pw(3,2) = 6.0d7*y(2);
  pw(3,3) = 0.0d0;

function display_plot(xplot, yplot, xOut1)
  % Formatting for title and axis labels.
  % Plot the three curves.
  plot(xplot, yplot(:,1), '-+', ...
       xplot, yplot(:,2), '--x', ...
       xplot, yplot(:,3), ':*');
  set(gca, 'YLim', [-0.05 1.05]);
  % Mark the height=0 point.
  do_stem(xplot, yplot, xOut1);
  % Add title.
  title('ODE Solution: BDF Method with Root-finding');
  % Label the x axis.
  xlabel('x');
  % Label the y axis.
  ylabel('Solution (a,b,c)');

function do_stem(xplot, yplot, xOut1)
  % Find the x bin that xOut1 lies in.
  for i = 1:length(xplot)
    if xplot(i) > xOut1
      break
    end
  end
  % Use linear interpolation to find the corresponding y values on the
  % two curves.
  dx = xplot(i)-xplot(i-1);
  ddx = xOut1-xplot(i-1);
  d1 = ddx/dx;

  f1 = yplot(i-1,2) + d1*(yplot(i,2)-yplot(i-1,2));
  f2 = yplot(i-1,3) + d1*(yplot(i,3)-yplot(i-1,3));
  % Plot the line from the x axis to the two y values.
  hold on
  stem([xOut1,xOut1],[f1,0],'k:s');
  hold on
  stem([xOut1,xOut1],[f2,0],'k:s');
  y = (abs(f1-f2)/2);
  text('Position',[xOut1-0.15,y],'String',['a = ',num2str(f1)],'Rotation',90);
```

9.2 Program Results

```
     d02ej example results

Case 1: calculating Jacobian internally,
intermediate output, root-finding

Calculation with tol = 0.001
 X     Y(1)        Y(2)        Y(3)
 0    1.000000    0.000000    0.000000
 2    0.941629    0.000027    0.058344
 4    0.905507    0.000022    0.094470

Root of Y(1)-0.9 = 0.0 at 4.3767
Solution is
    0.9000     0.0000     0.1000

Calculation with tol = 0.0001
 X     Y(1)        Y(2)        Y(3)
 0    1.000000    0.000000    0.000000
 2    0.941608    0.000027    0.058365
 4    0.905513    0.000022    0.094464

Root of Y(1)-0.9 = 0.0 at 4.3767
Solution is
    0.9000     0.0000     0.1000

Case 2: calculating Jacobian by pederv,
intermediate output, root-finding

Calculation with tol = 0.001
 X     Y(1)        Y(2)        Y(3)
 0    1.000000    0.000000    0.000000
 2    0.941629    0.000027    0.058344
 4    0.905507    0.000022    0.094470

Root of Y(1)-0.9 = 0.0 at 4.3767
Solution is
    0.9000     0.0000     0.1000

Calculation with tol = 0.0001
 X     Y(1)        Y(2)        Y(3)
 0    1.000000    0.000000    0.000000
 2    0.941608    0.000027    0.058365
 4    0.905513    0.000022    0.094464

Root of Y(1)-0.9 = 0.0 at 4.3767
Solution is
    0.9000     0.0000     0.1000

Case 3: calculating Jacobian internally,
no intermediate output, root-finding

Calculation with tol = 0.001

Root of Y(1)-0.9 = 0.0 at 4.3767
Solution is
    0.9000     0.0000     0.1000

Calculation with tol = 0.0001

Root of Y(1)-0.9 = 0.0 at 4.3767
Solution is
    0.9000     0.0000     0.1000

Case 4: calculating Jacobian internally,
intermediate output, no root-finding

Calculation with tol = 0.001
 X     Y(1)       Y(2)       Y(3)
 0    1.000000    0.000000    0.000000
```

```
 2    0.941629    0.000027    0.058344
 4    0.905507    0.000022    0.094470
 6    0.879302    0.000020    0.120678
 8    0.858580    0.000018    0.141402
10    0.841361    0.000016    0.158623

Calculation with tol = 0.0001
 X     Y(1)        Y(2)        Y(3)
 0    1.000000    0.000000    0.000000
 2    0.941608    0.000027    0.058365
 4    0.905513    0.000022    0.094464
 6    0.879261    0.000020    0.120720
 8    0.858536    0.000018    0.141446
10    0.841357    0.000016    0.158627

Case 5: calculating Jacobian internally,
no intermediate output, no root-finding (integrate to xend)

Calculation with tol = 0.001
 X     Y(1)        Y(2)        Y(3)
0    1.000000    0.000000    0.000000
10   0.841361    0.000016    0.158623

Calculation with tol = 0.0001
 X     Y(1)        Y(2)        Y(3)
0    1.000000    0.000000    0.000000
10   0.841357    0.000016    0.158627
```