NAG Toolbox

nag_ode_ivp_rk_zero_simple (d02bj)

1 Purpose

nag_ode_ivp_rk_zero_simple (d02bj) integrates a system of first-order ordinary differential equations over an interval with suitable initial conditions, using a fixed order Runge–Kutta method, 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, ifail] = nag_ode_ivp_rk_zero_simple(x, xend, y, fcn, tol, relabs, output,
g, 'n', n)
```

```
[x, y, ifail] = d02bj(x, xend, y, fcn, tol, relabs, output, g, 'n', n)
```

3 Description

nag_ode_ivp_rk_zero_simple (d02bj) 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 fixed order Runge–Kutta method. The system is defined by **fcn**, which evaluates f_i in terms of x and $y = (y_1, y_2, \ldots, y_n)$. The initial values of $y = (y_1, y_2, \ldots, y_n)$ must be given at $x = \mathbf{x}$.

The solution is returned via the **output** supplied by you and 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 $g(x, y) = 0$ 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 $g(x, y) = 0$, is controlled by the arguments **tol** and **relabs**.

4 References

Shampine L F (1994) *Numerical solution of ordinary differential equations* Chapman and Hall

5 Parameters

5.1 Compulsory Input Parameters

1: **x** – REAL (KIND=nag_wp)

The initial value of the independent variable x.

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 $x = \mathbf{x}$.

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: **tol** – REAL (KIND=nag_wp)

A **positive** tolerance for controlling the error in the integration. Hence **tol** affects the determination of the position where $g(x, y) = 0$, if g is supplied.

nag_ode_ivp_rk_zero_simple (d02bj) 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_rk_zero_simple (d02bj) 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_rk_zero_simple (d02bj) with **relabs** = 'D' and with each of **tol** $= 10.0^{-p}$ and **tol** $= 10.0^{-p-1}$ where p correct significant digits are required in the solution, y. The accuracy of the value x such that $g(x, y) = 0$ is indirectly controlled by varying **tol**. You should experiment to determine this accuracy.

Constraint: $10.0 \times$ ***machine precision*** $<$ **tol** < 0.01.

6: **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:

$$est = \max(e_i/(\tau_r \times \max(|y_i|, \tau_a))) \leq 1.0$$

where τ_r and τ_a are defined by

relabs	τ_r	τ_a
`M'	**tol**	1.0
`A'	ϵ_r	**tol**$/\epsilon_r$
`R'	**tol**	ϵ_a
`D'	**tol**	ϵ_a

where ϵ_r and ϵ_a are small machine-dependent numbers and e_i is an estimate of the local error at y_i, computed internally. If the 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** = 'M', 'A', 'R' or 'D'.

7: **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_rk_zero_simple (d02bj) 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_rk_zero_simple (d02bj) 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)

The output value of the independent variable x.

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_rk_zero_simple_dummy_output (d02bjx)`. (nag_ode_ivp_rk_zero_simple_dummy_output (d02bjx) is included in the NAG Toolbox.)

8: **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 point.

If you do not require the root-finding option, the actual argument **g** **must** be the string `nag_ode_ivp_rk_zero_simple_dummy_g (d02bjw)`. (nag_ode_ivp_rk_zero_simple_dummy_g (d02bjw) 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 equations.

Constraint: $\mathbf{n} > 0$.

5.3 Output Parameters

1: **x** – REAL (KIND=nag_wp)

If g is supplied by you, it contains the point where $g(x, y) = 0$, unless $g(x, y) \neq 0$ anywhere on the range **x** to **xend**, in which case, **x** will contain **xend** (and the error indicator **ifail** $= 6$ is set); if g is not supplied by you it contains **xend**. However, if an error has occurred, it contains the value of x at which the error occurred.

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

The computed values of the solution at the final point $x = \mathbf{x}$.

3: **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** ≥ 0.01,
or **tol** is too small
or **n** ≤ 0,
or **relabs** $\neq$ 'M', 'A', 'R' or 'D',
or **x** $=$ **xend**.

ifail $= 2$

With the given value of **tol**, no further progress can be made across the integration range from the current point $x = \mathbf{x}$. (See Section 9 for a discussion of this error exit.) The components $\mathbf{y}(1), \mathbf{y}(2), \ldots, \mathbf{y}(\mathbf{n})$ contain the computed values of the solution at the current point $x = \mathbf{x}$. If you have supplied g, then no point at which $g(x, y)$ changes sign has been located up to the point $x = \mathbf{x}$.

ifail $= 3$

tol is too small for nag_ode_ivp_rk_zero_simple (d02bj) 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 has not been reset or **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** has not been reset or 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 $= 7$

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** = 'D' unless you have a good reason for a different choice.

If the problem is a root-finding one, then the accuracy of the root determined will depend on the properties of $g(x, y)$ and on the values of **tol** and **relabs**. You should try to code **g** without introducing any unnecessary cancellation errors.

8 Further Comments

If more than one root is required, then to determine the second and later roots nag_ode_ivp_rk_zero_simple (d02bj) may be called again starting a short distance past the previously determined roots. Alternatively you may construct your own root-finding code using nag_roots_contfn_brent_rcomm (c05az), nag_ode_ivp_rkts_onestep (d02pf) and nag_ode_ivp_rkts_interp (d02ps).

If nag_ode_ivp_rk_zero_simple (d02bj) fails with **ifail** $= 3$, then it can be called again with a larger value of **tol** if this has not already been tried. If the accuracy requested is really needed and cannot be obtained with this function, the system may be very stiff (see below) or so badly scaled that it cannot be solved to the required accuracy.

If nag_ode_ivp_rk_zero_simple (d02bj) fails with **ifail** $= 2$, it is probable that it has been called with a value of **tol** which is so small that a solution cannot be obtained on the range **x** to **xend**. This can happen for well-behaved systems and very small values of **tol**. You should, however, consider whether there is a more fundamental difficulty. For example:

(a) in the region of a singularity (infinite value) of the solution, the function will usually stop with **ifail** $= 2$, unless overflow occurs first. Numerical integration cannot be continued through a singularity, and analytic treatment should be considered;

(b) for 'stiff' equations where the solution contains rapidly decaying components, the function will use very small steps in x (internally to nag_ode_ivp_rk_zero_simple (d02bj)) to preserve stability. This will exhibit itself by making the computing time excessively long, or occasionally by an exit with **ifail** $= 2$. Runge–Kutta methods are not efficient in such cases, and you should try nag_ode_ivp_bdf_zero_simple (d02ej).

9 Example

This example illustrates the solution of four different problems. In each case the differential system (for a projectile) is

$$y' = \tan\phi$$

$$v' = \frac{-0.032\tan\phi}{v} - \frac{0.02v}{\cos\phi}$$

$$\phi' = \frac{-0.032}{v^2}$$

over an interval $\mathbf{x} = 0.0$ to $\mathbf{xend} = 10.0$ starting with values $y = 0.5$, $v = 0.5$ and $\phi = \pi/5$. We solve each of the following problems with local error tolerances 1.0e−4 and 1.0e−5.

(i) To integrate to $x = 10.0$ producing intermediate output at intervals of 2.0 until a root is encountered where $y = 0.0$.

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

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

(iv) As (i) but with no intermediate output and no root-finding termination condition.

9.1 Program Text

```
     function d02bj_example

fprintf('d02bj example results\n\n');

% For communication with save.
global ykeep ncall xkeep;

% Initialize variables and arrays.
x = 0;
xend = 10;
y = [0.5; 0.5; pi/5];
relabs = 'Default';

xOut1 = [];
ncall = 0;
ykeep = zeros(1,length(y));
xkeep = zeros(1,1);

fprintf('Case 1: intermediate output, root-finding\n\n');
for j = 4:5
    tol = double(10)^(-j);
    disp(['Calculation with tol = ',num2str(tol)]);
    disp(' X     Y(1)       Y(2)       Y(3)');

    [xOut, yOut, ifail] = d02bj(x, xend, y, @fcn, tol, ...
        relabs, @output, @g);

    disp(' ');
    disp(['Root of Y(1) = 0.0 at ',num2str(xOut)]);
    disp('Solution is' );
    fprintf('  %8.4f   %8.4f   %8.4f\n\n', yOut);
end

fprintf('Case 2: no intermediate output, root-finding \n\n');
for j = 4:5
    tol = double(10)^(-j);
    disp(['Calculation with tol = ',num2str(tol)]);

    [xOut, yOut, ifail] = d02bj(x, xend, y, @fcn, tol, ...
        relabs, 'd02bjx',@g);

    disp(['Root of Y(1) = 0.0 at ',num2str(xOut)]);
```

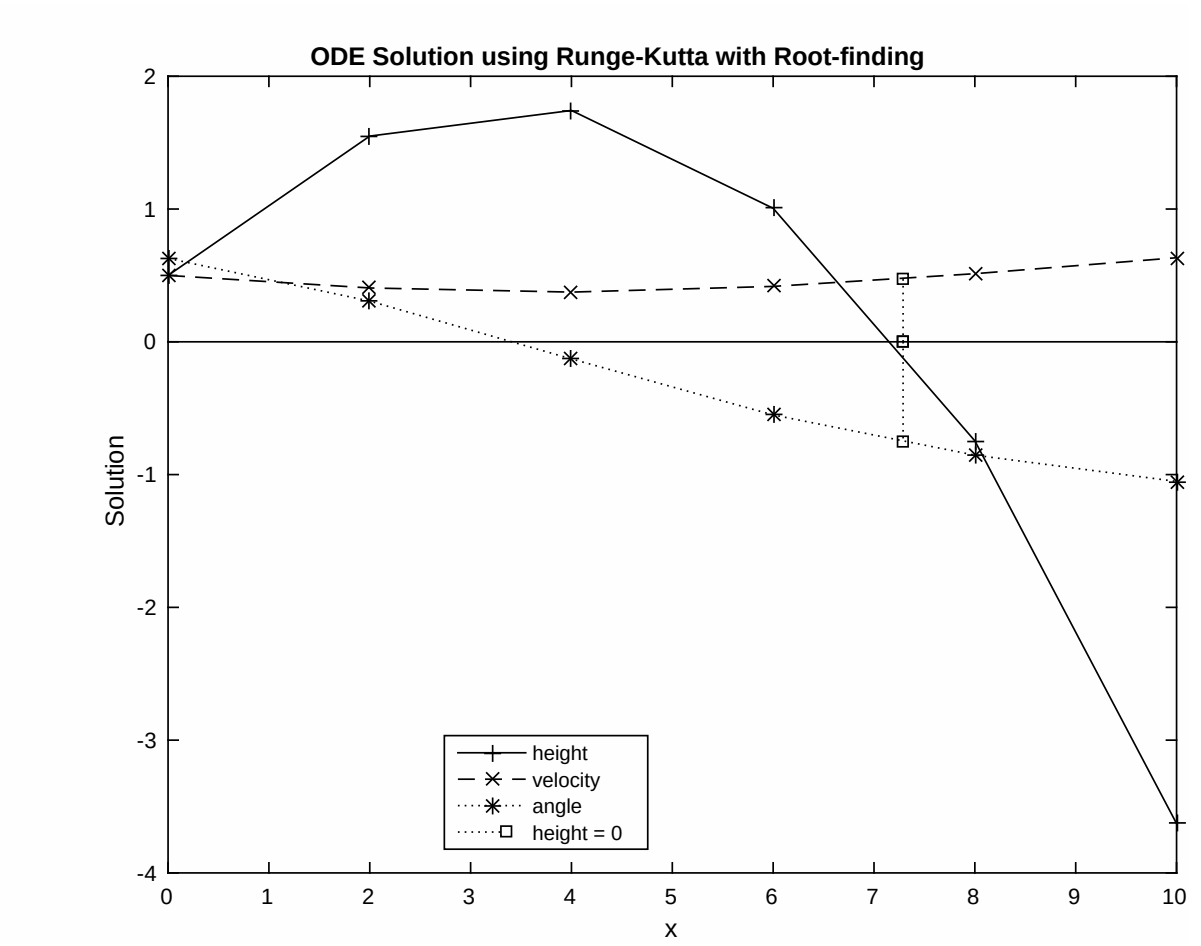
```
    disp('Solution is' );
    fprintf('  %8.4f   %8.4f   %8.4f\n\n', yOut);

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

fprintf('Case 3: intermediate output, no root-finding\n\n');
for j = 4:5
    tol = double(10)^(-j);
    disp(['Calculation with tol = ',num2str(tol)]);
    fprintf(' X     Y(1)       Y(2)       Y(3) \n');

    % save stores intermediate values in xkeep, ykeep, which are
    % plotted later (it also outputs them).
    [xOut, yOut, ifail] = d02bj(x, xend, y, @fcn, tol, ...
        relabs, @save,'d02bjw');
    fprintf('\n');
end

fprintf(['Case 4: no intermediate output, no root-finding ', ...
    '(integrate to xend)\n\n']);
for j = 4:5
    tol = double(10)^(-j);
    disp(['Calculation with tol = ',num2str(tol)]);
    disp(' X     Y(1)       Y(2)       Y(3)');
    fprintf('%2d   %8.4f   %8.4f   %8.4f\n', x, y);

    [xOut, yOut, ifail] = ...
        d02bj(x, xend, y, @fcn, tol, relabs, 'd02bjx','d02bjw');
    fprintf('%2d   %8.4f   %8.4f   %8.4f\n\n', xOut, yOut);
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   %8.4f   %8.4f   %8.4f\n', xsol, y);
xsolOut = xsol + 2;

function xsolOut = output(xsol, y)
% Output intermediate values of solution.
fprintf('%2d   %8.4f   %8.4f   %8.4f\n', xsol, y);
xsolOut = xsol + 2;

function f = fcn(x,y)
% Evaluate the derivatives.
f = zeros(3,1);
f(1) = tan(y(3));
f(2) = -0.032*tan(y(3))/y(2) - 0.02*y(2)/cos(y(3));
f(3) = -0.032/y(2)^2;

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

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), ':*');
% Mark the height=0 point.
do_stem(xplot, yplot, xOut1);
% Add title.
title('ODE Solution using Runge-Kutta with Root-finding');
% Label the axes.
xlabel('x');
ylabel('Solution');
% Add legend.
legend('height','velocity','angle','height = 0','Location','Best');

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');
```

9.2 Program Results

```
     d02bj example results

Case 1: intermediate output, root-finding

Calculation with tol = 0.0001
 X     Y(1)       Y(2)       Y(3)
 0     0.5000     0.5000     0.6283
 2     1.5493     0.4055     0.3066
 4     1.7423     0.3743    -0.1289
 6     1.0055     0.4173    -0.5507

Root of Y(1) = 0.0 at 7.2882
Solution is
   -0.0000     0.4749    -0.7601

Calculation with tol = 1e-05
 X     Y(1)       Y(2)       Y(3)
 0     0.5000     0.5000     0.6283
 2     1.5493     0.4055     0.3066
 4     1.7423     0.3743    -0.1289
 6     1.0055     0.4173    -0.5507

Root of Y(1) = 0.0 at 7.2883
Solution is
   -0.0000     0.4749    -0.7601

Case 2: no intermediate output, root-finding

Calculation with tol = 0.0001
Root of Y(1) = 0.0 at 7.2882
Solution is
   -0.0000     0.4749    -0.7601
```

```
Calculation with tol = 1e-05
Root of Y(1) = 0.0 at 7.2883
Solution is
   -0.0000     0.4749    -0.7601

Case 3: intermediate output, no root-finding

Calculation with tol = 0.0001
 X     Y(1)       Y(2)       Y(3)
 0     0.5000     0.5000     0.6283
 2     1.5493     0.4055     0.3066
 4     1.7423     0.3743    -0.1289
 6     1.0055     0.4173    -0.5507
 8    -0.7460     0.5130    -0.8537
10    -3.6283     0.6333    -1.0515

Calculation with tol = 1e-05
 X     Y(1)       Y(2)       Y(3)
 0     0.5000     0.5000     0.6283
 2     1.5493     0.4055     0.3066
 4     1.7423     0.3743    -0.1289
 6     1.0055     0.4173    -0.5507
 8    -0.7459     0.5130    -0.8537
10    -3.6282     0.6333    -1.0515

Case 4: no intermediate output, no root-finding (integrate to xend)

Calculation with tol = 0.0001
 X     Y(1)       Y(2)       Y(3)
 0     0.5000     0.5000     0.6283
10    -3.6283     0.6333    -1.0515

Calculation with tol = 1e-05
 X     Y(1)       Y(2)       Y(3)
 0     0.5000     0.5000     0.6283
10    -3.6282     0.6333    -1.0515
```


ODE Solution using Runge-Kutta with Root-finding
Solution
x
height
velocity
angle
height = 0
2
1
0
-1
-2
-3
-4
0
1
2
3
4
5
6
7
8
9
10