# NAG Toolbox

# nag_ode_ivp_2nd_rkn (d02la)

## 1 Purpose

nag_ode_ivp_2nd_rkn (d02la) is a function for integrating a non-stiff system of second-order ordinary differential equations using Runge–Kutta–Nystrom techniques.

## 2 Syntax

```
[t, y, yp, ydp, rwork, ifail] = nag_ode_ivp_2nd_rkn(fcn, t, tend, y, yp, ydp, rwork, 'neq', neq)
[t, y, yp, ydp, rwork, ifail] = d02la(fcn, t, tend, y, yp, ydp, rwork, 'neq', neq)
```

**Note**: the interface to this routine has changed since earlier releases of the toolbox:

At Mark 22: *lrwork* was removed from the interface.

## 3 Description

Given the initial values $x, y_1, y_2, \ldots, y_{\mathbf{neq}}, y'_1, y'_2, \ldots, y'_{\mathbf{neq}}$ nag_ode_ivp_2nd_rkn (d02la) integrates a non-stiff system of second-order differential equations of the type

$$y''_i = f_i\left(x, y_1, y_2, \ldots, y_{\mathbf{neq}}\right), \quad i = 1, 2, \ldots, \mathbf{neq},$$

from $x = \mathbf{t}$ to $x = \mathbf{tend}$ using a Runge–Kutta–Nystrom formula pair. The system is defined by **fcn**, which evaluates $f_i$ in terms of $x$ and $y_1, y_2, \ldots, y_{\mathbf{neq}}$, where $y_1, y_2, \ldots, y_{\mathbf{neq}}$ are supplied at $x$.

There are two Runge–Kutta–Nystrom formula pairs implemented in this function. The lower order method is intended if you have moderate accuracy requirements and may be used in conjunction with the interpolation function nag_ode_ivp_2nd_rkn_interp (d02lz) to produce solutions and derivatives at user-specified points. The higher order method is intended if you have high accuracy requirements.

In one-step mode the function returns approximations to the solution, derivative and $f_i$ at each integration point. In interval mode these values are returned at the end of the integration range. You select the order of the method, the mode of operation, the error control and various optional inputs by a prior call to nag_ode_ivp_2nd_rkn_setup (d02lx).

For a description of the Runge–Kutta–Nystrom formula pairs see Dormand *et al.* (1986a) and Dormand *et al.* (1986b) and for a description of their practical implementation see Brankin *et al.* (1989).

## 4 References

Brankin R W, Dormand J R, Gladwell I, Prince P J and Seward W L (1989) Algorithm 670: A Runge–Kutta–Nystrom Code *ACM Trans. Math. Software* **15** 31–40

Dormand J R, El–Mikkawy M E A and Prince P J (1986a) Families of Runge–Kutta–Nystrom formulae *Mathematical Report TPMR 86-1* Teesside Polytechnic

Dormand J R, El–Mikkawy M E A and Prince P J (1986b) High order embedded Runge–Kutta–Nystrom formulae *Mathematical Report TPMR 86-2* Teesside Polytechnic

# 5 Parameters

## 5.1 Compulsory Input Parameters

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

**fcn** must evaluate the functions $f_i$ (that is the second derivatives $y_i''$) for given values of its arguments $x$, $y_1, y_2, \ldots, y_{\mathbf{neq}}$.

```
[f] = fcn(neq, t, y)
```

**Input Parameters**

1: **neq** – INTEGER

The number of differential equations.

2: **t** – REAL (KIND=nag_wp)

$x$, the value of the argument.

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

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

**Output Parameters**

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

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

2: **t** – REAL (KIND=nag_wp)

The initial value of the independent variable $x$.

*Constraint*: **t** $\neq$ **tend**.

3: **tend** – REAL (KIND=nag_wp)

The end point of the range of integration. If **tend** $<$ **t** on initial entry, integration will proceed in the negative direction. **tend** may be reset, in the direction of integration, before any continuation call.

4: **y**(**neq**) – REAL (KIND=nag_wp) array

The initial values of the solution $y_1, y_2, \ldots, y_{\mathbf{neq}}$.

5: **yp**(**neq**) – REAL (KIND=nag_wp) array

The initial values of the derivatives $y_1', y_2', \ldots, y_{\mathbf{neq}}'$.

6: **ydp**(**neq**) – REAL (KIND=nag_wp) array

Must be unchanged from a previous call to nag_ode_ivp_2nd_rkn (d02la).

7: **rwork**($lrwork$) – REAL (KIND=nag_wp) array

This **must** be the same argument **rwork** as supplied to nag_ode_ivp_2nd_rkn_setup (d02lx). It is used to pass information from nag_ode_ivp_2nd_rkn_setup (d02lx) to nag_ode_ivp_2nd_rkn (d02la), and from nag_ode_ivp_2nd_rkn (d02la) to both nag_ode_ivp_2nd_rkn_diag (d02ly) and nag_ode_ivp_2nd_rkn_interp (d02lz). Therefore the contents of this array **must not** be changed before the call to nag_ode_ivp_2nd_rkn (d02la) or calling either of the functions nag_ode_ivp_2nd_rkn_diag (d02ly) and nag_ode_ivp_2nd_rkn_interp (d02lz).

### 5.2 Optional Input Parameters

1: **neq** – INTEGER

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

The number of second-order ordinary differential equations to be solved by nag_ode_ivp_2nd_rkn (d02la). It must contain the same value as the argument **neq** used in a prior call to nag_ode_ivp_2nd_rkn_setup (d02lx).

*Constraint*: **neq** $\geq 1$.

### 5.3 Output Parameters

1: **t** – REAL (KIND=nag_wp)

The value of the independent variable, which is usually **tend**, unless an error has occurred or the code is operating in one-step mode. If the integration is to be continued, possibly with a new value for **tend**, **t** must not be changed.

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

The computed values of the solution at the exit value of **t**. If the integration is to be continued, possibly with a new value for **tend**, these values must not be changed.

3: **yp**(**neq**) – REAL (KIND=nag_wp) array

The computed values of the derivatives at the exit value of **t**. If the integration is to be continued, possibly with a new value for **tend**, these values must not be changed.

4: **ydp**(**neq**) – REAL (KIND=nag_wp) array

The computed values of the second derivative at the exit value of **t**, unless illegal input is detected, in which case the elements of **ydp** may not have been initialized. If the integration is to be continued, possibly with a new value for **tend**, these values must not be changed.

5: **rwork**($lrwork$) – REAL (KIND=nag_wp) array

6: **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** $= 1$

Illegal input detected, i.e., one of the following conditions:

on any call, **t** $=$ **tend**, or the value of **neq** or $lrwork$ has been altered;

on a continuation call, the direction of integration has been changed;

nag_ode_ivp_2nd_rkn_setup (d02lx) had not been called previously, or the previous call to nag_ode_ivp_2nd_rkn_setup (d02lx) resulted in an error exit.

This error exit can be caused if elements of **rwork** have been overwritten.

**ifail** $= 2$

The maximum number of steps has been attempted. (See argument **maxstp** in nag_ode_ivp_2nd_rkn_setup (d02lx).) If integration is to be continued then you need only reset **ifail** and call the function again and a further **maxstp** steps will be attempted.

**ifail** $= 3$

In order to satisfy the error requirements, the step size needed is too small for the ***machine precision*** being used.

**ifail** $= 4$

The code has detected two successive error exits at the current value of $x$ and cannot proceed. Check all input variables.

**ifail** $= 5$

The code has detected inefficient use of the integration method. The step size has been reduced by a significant amount too often in order to hit the output points specified by **tend**. (Of the last 100 or more successful steps more than $10\%$ are steps with sizes that have had to be reduced by a factor of greater than a half.)

**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 integration is determined by the arguments **tol**, **thres** and **thresp** in a prior call to nag_ode_ivp_2nd_rkn_setup (d02lx). Note that only the local error at each step is controlled by these arguments. The error estimates obtained are not strict bounds but are usually reliable over one step. Over a number of steps the overall error may accumulate in various ways, depending on the system. The code is designed so that a reduction in **tol** should lead to an approximately proportional reduction in the error. You are strongly recommended to call nag_ode_ivp_2nd_rkn (d02la) with more than one value for **tol** and to compare the results obtained to estimate their accuracy.

The accuracy obtained depends on the type of error test used. If the solution oscillates around zero a relative error test should be avoided, whereas if the solution is exponentially increasing an absolute error test should not be used. For a description of the error test see the specifications of the arguments **tol**, **thres** and **thresp** in function document nag_ode_ivp_2nd_rkn_setup (d02lx).

## 8 Further Comments

If nag_ode_ivp_2nd_rkn (d02la) fails with **ifail** $= 3$ then the value of **tol** may be so small that a solution cannot be obtained, in which case the function should be called again with a larger value for **tol**. If the accuracy requested is really needed then you should consider whether there is a more fundamental difficulty. For example:

(a) in the region of a singularity the solution components will usually be of a large magnitude. nag_ode_ivp_2nd_rkn (d02la) could be used in one-step mode to monitor the size of the solution with the aim of trapping the solution before the singularity. In any case numerical integration cannot be continued through a singularity, and analytical treatment may be necessary;

(b) if the solution contains fast oscillatory components, the function will require a very small step size to preserve stability. This will usually be exhibited by excessive computing time and sometimes an error exit with **ifail** $= 3$. The Runge–Kutta–Nystrom methods are not efficient in such cases and you should consider reposing your problem as a system of first-order ordinary differential equations and then using a function from Sub-chapter D02M–N with the Blend formulae (see nag_ode_ivp_stiff_dassl (d02mv)).

nag_ode_ivp_2nd_rkn (d02la) can be used for producing results at short intervals (for example, for tabulation), in two ways. By far the less efficient is to call nag_ode_ivp_2nd_rkn (d02la) successively over short intervals, $t + (i-1) \times h$ to $t + i \times h$, although this is the only way if the higher order method has been selected and precisely **not** what it is intended for. A more efficient way, **only** for use when the lower order method has been selected, is to use nag_ode_ivp_2nd_rkn (d02la) in one-step mode. The output values of arguments **y**, **yp**, **ydp**, **t** and **rwork** are set correctly for a call to nag_ode_ivp_2nd_rkn_interp (d02lz) to compute the solution and derivative at the required points.

# 9 Example

This example solves the following system (the two body problem)

$$\begin{aligned} y_1'' &= -y_1/\left(y_1^2+y_2^2\right)^{3/2} \\ y_2'' &= -y_2/\left(y_1^2+y_2^2\right)^{3/2} \end{aligned}$$

over the range $[0, 20]$ with initial conditions $y_1 = 1.0 - \epsilon$, $y_2 = 0.0$, $y_1' = 0.0$ and $y_2' = \sqrt{\left(\dfrac{1+\epsilon}{1-\epsilon}\right)}$ where $\epsilon$, the eccentricity, is 0.5. The system is solved using the lower order method with relative local error tolerances 1.0e−4 and 1.0e−5 and default threshold tolerances. nag_ode_ivp_2nd_rkn (d02la) is used in one-step mode (**onestp** = *true*) and nag_ode_ivp_2nd_rkn_interp (d02lz) provides solution values at intervals of 2.0.

## 9.1 Program Text

```
function d02la_example

fprintf('d02la example results\n\n');

% Initialize variables and arrays.
ecc = 0.5;
tend = 20;
tstart = 0;
tinc = 0.1;

neq = 2;
lrwork = 16+20*neq;
nwant = neq;
thres = zeros(neq, 1);
thresp = zeros(neq, 1);

fprintf('d02la example program results \n\n');

for itol = 4:5
    tol = 10^(-itol);
    thres(1) = 0.0;
    thresp(1) = 0.0;
    h = 0.0;
    maxstp = 0;
    start = true;
    high = false;
    onestp = true;
    rwork = zeros(lrwork,1);
    t = 0;
    y = [1-ecc; 0];
    yp = [0; sqrt((1+ecc)/(1-ecc))];
    ydp = zeros(neq, 1);

    % d02lx is a setup routine to be called prior to d02la.
    [start, rwork, ifail] = d02lx(h, tol, thres, thresp, nag_int(maxstp),...
        start, onestp, high, rwork, 'neq', nag_int(neq));

    fprintf('Calculation with tol = %1.1e \n\n',tol);
    % fprintf('   t          y_1          y_2 ');
    t = tstart;
    tnext = t + tinc;
```

```
    % fprintf('\n %4.1f    %10.5f    %10.5f\n', t, y);

    if itol == 4
        ncall = 1;
        tkeep = tnext;
        ykeep = y;
        nkeep = 1;
    end

    while (t < tend)

        [tout, y, yp, ydp, rwork, ifail] = d02la(@fcn, t, tend, y,...
            yp, ydp, rwork);
        t = tout;

        while (tnext <= t)
            % d02lz interpolates components of solution provided by d02la.

             [ywant, ypwant, ifail] = d02lz(t, y, yp, nag_int(neq), tnext,...
                rwork, 'neq', nag_int(neq));

            ncall = ncall+1;
            % fprintf(' %4.1f    %10.5f    %10.5f\n', tnext, ywant);

            if itol == 4
                tkeep(ncall) = tnext;
                ykeep(:,ncall) = ywant;
            end

            tnext = tnext + tinc;
        end
    end

    % d02ly is a diagnostic routine for use after calling d02la.

    [hnext, hused, hstart, nsucc, nfail, natt, thres, thresp, ifail] =...
        d02ly(nag_int(neq), rwork);

    fprintf('\nNumber of successful steps = %1.0f\n',nsucc);
    fprintf('Number of failed     steps = %1.0f\n',nfail);
    fprintf('\n\n');
end

% Find the index of the point halfway through the first orbit (i.e. the
% first point at which the y coord is negative).  This is used to calculate
% the end points of the orbits in display_plot.
nhalf = 1;
while ykeep(2,nhalf) >= 0
    nhalf = nhalf+1;
end
% Plot results.
fig1 = figure;
display_plot(ykeep, nhalf);

function ydp = fcn(neq, t, y)
% Evaluate second derivatives.
r = sqrt(y(1)^2+y(2)^2)^3;
f = zeros(2,1);
ydp(1) = -y(1)/r;
ydp(2) = -y(2)/r;

function display_plot(ykeep, nhalf)

% Find the end points of the three orbits.
norb1 = 2*(nhalf-1); norb2 = 2*norb1; norb3 = 3*norb1;

% Plot the orbits, including displacements.
plot(ykeep(1,1:norb1), ykeep(2,1:norb1), '-+', ...
       ykeep(1,norb1+1:norb2), ykeep(2,norb1+1:norb2)+0.1, '--x', ...
       ykeep(1,norb2+1:norb3), ykeep(2,norb2+1:norb3)+0.2, ':*');
% Add title.
```

```
title({'Second-order ODE Solution using Runge-Kutta-Nystrom.', ...
    'The Two-body Problem (using shifts to distinguish orbits)'});
% Label the axes.
xlabel('x'); ylabel('y');
% Add a legend.
legend('1st orbit', '2nd orbit+(0,0.1)', '3rd orbit+(0,0.2)', ...
       'Location', 'Best');
```

## 9.2 Program Results

```
     d02la example results

d02la example program results

Calculation with tol = 1.0e-04

Number of successful steps = 108
Number of failed     steps = 16

Calculation with tol = 1.0e-05

Number of successful steps = 169
Number of failed     steps = 15
```