Module 12.1: nag_ivp_ode_rk

Solution of Initial-value Problems for Ordinary Differential Equations by Runge–Kutta Methods

`nag_ivp_ode_rk` contains a set of procedures for solving the initial-value problem for a system of first-order ordinary differential equations. The procedures are based on Runge–Kutta methods.

Contents

Introduction

This module provides a set of procedures for solving the initial value problem for a system of first-order ordinary differential equations. The procedures, based on Runge–Kutta methods, integrate

$$y' = f(t, y), \text{ given the initial condition } y(t_0)$$

from the initial point t_0 towards the end point t_e, where y is the vector of n solution components and t is the independent variable.

Integration proceeds in a step by step fashion from the initial point t_0 towards the end point t_e and the approximate solution y is computed at points which depend on the particular integration procedure selected. For each component y_i, for $i = 1, 2, \ldots, n$, the estimate of the error made at the points of integration, i.e., the local error, is kept smaller than a user-specified accuracy.

The data related to initial conditions, t_0 and $y(t_0)$, the end point t_e, and the accuracy required in the process of computing the solution must be specified by a call to the setup procedure `nag_rk_setup`. This *must* be followed by one or more calls to an integration procedure.

The module offers a choice between the following two integration procedures.

- `nag_rk_interval` enables you to specify explicitly the points at which the solution is required across an interval.
- `nag_rk_step` computes the solution one step at a time. The size of each step is selected automatically by the procedure.

You may specify which of these procedures you intend to use in the call to `nag_rk_setup`.

The difference between `nag_rk_interval` and `nag_rk_step` is the flexibility which `nag_rk_step` offers you for performing different tasks between each integration step; for example, you may wish to locate a root of a function of the solution in between each step. The flexibility which `nag_rk_step` offers makes `nag_rk_setup` quite suitable for performing complicated tasks and solving difficult problems. Before using this procedure, you are advised to think carefully about the nature of your problem, and the tasks which you wish to perform.

There are two diagnostic procedures which can provide additional information about the integration after a call to either `nag_rk_interval` or `nag_rk_step`.

- `nag_rk_info` provides details of the step size and the cost of the integration.
- `nag_rk_global_err` provides information about global error assessment.

In addition there are two utility procedures which can *only* be used in conjunction with `nag_rk_step`.

- `nag_rk_interp` computes the solution by interpolation.
- `nag_rk_reset_end` may be used to reset the end point of integration t_e in between calls to `nag_rk_step`. This is more efficient than calling `nag_rk_setup` to reinitialise the integration.

`nag_rk_interval` uses `nag_rk_step`, `nag_rk_reset_end` and `nag_rk_interp` to compute the solution at user-specified points.

Procedure: nag_rk_setup

1 Description

`nag_rk_setup` is a set up procedure which *must* be called prior to either of the two integration procedures `nag_rk_interval` and `nag_rk_step`. It is used to specify the data defining the initial conditions, t_0 and $y(t_0)$, the end point t_e, the error control parameters and other options. It is also used to initialise the communicating structure `comm`.

The integration procedures in this module use relative local error control, with `tol` being the desired relative accuracy. An explanation of how the local error is controlled appears in Section 6.1.

There are optional arguments which specify the selection of a Runge–Kutta pair to be used for integration, the choice of the integration procedure, whether or not global error assessment should take place, and the setting of a trial initial step size. Advice on the choice of these optional arguments is provided in Section 6.

2 Usage

```
USE nag_ivp_ode_rk
CALL nag_rk_setup(t_start, y_start, t_end, tol, thresh, comm  [, optional arguments])
```

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '**x**(n)' is used in the argument descriptions to specify that the array `x` must have exactly n elements.

This procedure derives the value of the following problem parameter from the shape of the supplied arrays.

$n \geq 1$ — the number of equations

3.1 Mandatory Arguments

t_start — real(kind=*wp*), intent(in)

Input: the initial value of the independent variable, t_0.

y_start(n) — real(kind=*wp*), intent(in)

Input: the initial values $y_i(t_0)$ of the solution y_i, for $i = 1, 2, \ldots, n$ at t_0.

t_end — real(kind=*wp*), intent(in)

Input: the final value of the independent variable, t_e, at which the solution is required. `t_start` and `t_end` together determine the direction of integration.

Constraints: `t_end` must be distinguishable from `t_start` for the method and precision of the machine being used.

tol — real(kind=*wp*), intent(in)

Input: a relative error tolerance. See Section 6.1 and Section 6.4 for more details.

Constraints: $10 \times$ `EPSILON(1.0_`*wp*`)` $\leq$ `tol` ≤ 0.01.

thresh(n) — real(kind=*wp*), intent(in)

Input: a vector of thresholds. See Section 6.1 and Section 6.2 for details of how this argument is used for the control of local error.

Constraints: `thresh` $\geq$ `SQRT(TINY(1.0_`*wp*`))`.

comm — type(nag_rk_comm_*wp*), intent(out)

Output: this argument is the communicating structure, and is initialised to hold information concerning how the user wants to perform the integration.

Note: to reduce the risk of corrupting the data accidentally, the components of this structure are private.

The procedure allocates between $10n$ and $32n$ real(kind=*wp*) elements of storage to the structure. If you wish to deallocate this storage when the integration is complete and the structure is no longer required, you must call the procedure `nag_deallocate`, as illustrated in Example 1. If `comm` was used in a previous call to this procedure, the information generated by that call will be lost.

3.2 Optional Arguments

Note. Optional arguments must be supplied by keyword, not by position. The order in which they are described below may differ from the order in which they occur in the argument list.

method — integer, intent(in), optional

Input: the Runge–Kutta method to be used. See Section 6.3 for advice on the choice of this argument.

If `method` = 1 then a 2(3) pair is used;

if `method` = 2 then a 4(5) pair is used;

if `method` = 3 then a 7(8) pair is used.

Default: `method` = 2.

Constraints: 1 ≤ `method` ≤ 3.

call_step — logical, intent(in), optional

Input: determines whether integration is to be performed by a call to this procedure (see the Module Introduction).

If `call_step` = `.false.`, then `nag_rk_interval` is to be used for the integration;

if `call_step` = `.true.`, then this procedure is to be used for integration.

Default: `call_step` = `.false.`.

global_err — logical, intent(in), optional

Input: specifies whether a global error assessment is to be computed. See Section 6.4 for more details.

Default: `global_err` = `.false.`, i.e., no global error assessment is to be computed.

h_start — real(kind=*wp*), intent(in), optional

Input: a trial value for the size of the first step in the integration to be attempted. The absolute value of `h_start` is used with the direction being determined by `t_start` and `t_end`. If `h_start` = 0.0 then the size of the first step is computed automatically. See Section 6.5 for more details.

Note: if `h_start` > | `t_end` − `t_start` |, then `h_start` is internally set to zero.

Default: `h_start` = 0.0.

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document `nag_error_handling` (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to `nag_set_error` before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
302	An array argument has an invalid shape.
303	Array arguments have inconsistent shapes.
320	The procedure was unable to allocate enough memory.

5 Examples of Usage

Complete examples of the use of this procedure appear in Examples 1, 2 and 3 of this module document.

6 Further Comments

6.1 Accuracy

Essentially, the integration procedures use relative local error control, with `tol` being the desired relative accuracy. The magnitude of the local error in y_i on any step will not be greater than `tol` $\times$ max(η_i,`thresh(`i`)`) where η_i is an average magnitude of y_i over the step. If `thresh(`i`)` is smaller than the current value of η_i, this is a relative error control and `tol` indicates how many significant digits are required in y_i. If `thresh(`i`)` is larger than the current value of η_i, this is an absolute error control with tolerance `tol` $\times$ `thresh(`i`)`. Relative error control is the recommended mode of operation, but pure relative error control (i.e., `thresh(`i`)` = 0.0) is not permitted.

6.2 Choice of Thresholds

An appropriate threshold depends on the general size of y_i in the course of the integration. It is often the case that a solution component y_i is of no interest when it is smaller in magnitude than a certain threshold. You can inform the code of this by setting `thresh(`i`)` to this threshold. In this way you avoid the cost of computing unnecessary significant digits in y_i. This matter is important whenever the initial value `y_start(`i`)` vanishes. For example, when one of the solution components might vanish, you should not specify a very small threshold. In particular, this is not advised when $y_i = 0.0$. Physical reasoning may help you select suitable threshold values.

6.3 Choice of Runge–Kutta Method

The Runge–Kutta formulae pairs available are of orders 2(3), 4(5) and 7(8). As a rule, the smaller the required relative accuracy, the higher the order of Runge–Kutta method to be used. For a relative accuracy requirement, experience suggests:

required accuracy	most efficient Runge–Kutta method
$10^{-2} - 10^{-4}$	2(3)
$10^{-3} - 10^{-6}$	4(5)
$10^{-5} - 10 \times$ `EPSILON(1.0_`wp`)`	7(8)

Making the required accuracy smaller will normally make the integration more expensive. However, within the range of required accuracy appropriate to a given Runge–Kutta method, this increase in cost is modest.

In the following situations a choice of Runge–Kutta method is not advised.

1. None of the methods are efficient when the system of ordinary differential equations is 'stiff'. For a mildly stiff problem, the 2(3) method may solve the system with an acceptable efficiency. However, if the problem is moderately or very stiff, a code designed specifically for such a system will be much more efficient.

2. In the case when one of the solution components might vanish, with a very small threshold (see above), the methods will have to work hard to compute the significant digits. In this situation you are advised *not* to use the 2(3) method.

3. The 7(8) Runge–Kutta pair cannot be used in conjunction with the interpolation procedure `nag_rk_interp`.

In general, for efficiency, the higher the order of method, the more smoothness is required in the behaviour of the solution.

6.4 Local Error Control and Global Error Assessment

We define the true or global error as the difference between the numerical solution and the true solution and local error as the error made in each step of computation. `nag_rk_interval` and `nag_rk_step` are designed to control local error rather than the true (global) error. However, control of the local error controls the true error indirectly. Roughly speaking, the solution computed by the code satisfies the differential equation to within a fraction of the error tolerance. Furthermore, the closeness of the numerical solution to the true solution depends on the stability of the problem. Most practical problems are at least moderately stable, and the true error is then comparable to the error tolerance. The accuracy of the numerical solution can be judged by the following two (indirect and direct) approaches.

1. You could reduce `tol` substantially, e.g.,, use $0.1 \times$ `tol`, and solve the problem again. This will usually result in a rather more accurate solution, and the true error of the first integration can be estimated by comparison. The assessment of the true error by this approach is generally satisfactory.

2. Alternatively, a global error assessment can be computed automatically if the argument `global_err` is set to `.true.`. The direct assessment of the global error at each step is performed by a *primary integration*, that is computing a solution y_i at the point t_i with an internally computed step size, and a *secondary integration*, that is computing a more accurate solution $\hat{y_i}$ at the same point (t_i) by taking two (or more) steps. A comparison of these two results provides a mean for assessing the quality of the solution in the primary integration. The primary integration is exactly that computed when `global_err` = `.false.`.

Because both ways of assessing true errors cost significantly more than the cost of the integration itself, such assessments should be used mostly for spot checks, selecting appropriate tolerances for local error control, and exploratory computations.
When assessment of the global error is requested directly, this error assessment is updated at each step. Its value can be obtained at any time by a call to procedure `nag_rk_global_err`. The code monitors the computation of the global error assessment and reports any doubts it has about the reliability of the results. The assessment scheme requires some smoothness of $f(t, y)$, and it can be deceived if f is not sufficiently smooth. At very crude tolerances the numerical solution can become so inaccurate that it is impossible to continue assessing the accuracy reliably. At very stringent tolerances the effects of finite precision arithmetic can make it impossible to assess the accuracy reliably. The cost of direct global error assessment is roughly twice the cost of the integration itself with `method` = 2 or 3, and three times with `method` = 1.

6.5 Choice of the Initial Step Size

The first step of the integration is critical because it sets the scale of the problem. The integrator will find a starting step size automatically if `h_start` = 0.0 (the default). Automatic selection of the first step is so effective that you should normally use it. Nevertheless, you might want to specify a trial value for the first step to be certain that the code recognizes the scale on which phenomena occur near the initial point. Also, automatic computation of the first step size involves some cost, so supplying a good value for this step size will result in a less expensive start. If you are confident that you have a good value, supply it in the optional argument `h_start`.

6.6 Resetting the End-point of the Integration Interval

If integration is taking place using this procedure, the value of t_end may be reset during the integration without the overhead associated with a complete restart; this can be achieved by a call to nag_rk_reset_end.

Procedure: nag_rk_interval

1 Description

`nag_rk_interval` is a procedure for solving the initial value problem for a system of first-order ordinary differential equations across an interval, using a Runge–Kutta method. This procedure is designed to compute an approximate solution at user-specified points. A call must first be made to `nag_rk_setup` to specify the problem and how it is to be solved. Thereafter you call `nag_rk_interval` repeatedly with successive values of `t_want`, the points at which you want the solution, in the range from `t_start` to `t_end` (as specified in `nag_rk_setup`). In this manner this procedure returns the point at which it has computed a solution, `t_got` (usually `t_want`), the solution there, `y_got`, and its derivative, `yp_got`. If this procedure encounters some difficulty in taking a step towards `t_want`, then it returns the point of difficulty `t_got` and derivative `yp_got` computed there.

In the call to `nag_rk_setup` you can specify the first step size for `nag_rk_interval` to attempt. Alternatively `nag_rk_interval` will compute an appropriate first step. Thereafter this procedure estimates an appropriate step size for its next step. This value and other details of the integration can be obtained after any call to this procedure by a call to the procedure `nag_rk_info`. The local error is controlled at every step as specified in Section 6.1.

For more complicated tasks, you are referred to procedures `nag_rk_step`, `nag_rk_interp` and `nag_rk_reset_end`.

2 Usage

```
USE nag_ivp_ode_rk

CALL nag_rk_interval(f, t_want, t_got, y_got, yp_got, comm  [, optional arguments])
```

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '`x`(n)' is used in the argument descriptions to specify that the array `x` must have exactly n elements.

This procedure derives the value of the following problem parameter from the shape of the supplied arrays.

$n \geq 1$ — the number of equations

3.1 Mandatory Arguments

f — function

The function `f` supplied by the user must evaluate the functions f_i (that is the first derivatives y_i') for the given values of the arguments t and y_i.

The specification of `f` is:

```
function f(t, y)

real(kind=wp), intent(in) ::  t
```

Input: the current value of the independent variable, t.

```
real(kind=wp), intent(in) ::  y(:)
```

Shape: `y` has shape (n).

Input: the current values of the dependent variables, y_i, for $i = 1, 2, \ldots, n$.

```
real(kind=wp) ::  f(SIZE(y))
```
Result: the values of the first derivatives, y_i', for $i = 1, 2, \ldots, n$.

t_want — real(kind=*wp*), intent(in)

Input: the next value of the independent variable, t, where a solution is desired.

Constraints: `t_want` must be closer to `t_end` than the previous value of `t_got` (or `t_start` on the first call to this procedure). Note that `t_want` must not lie beyond `t_end` in the direction of integration.

t_got — real(kind=*wp*), intent(out)

Output: the value of the independent variable, t, at which the solution has been computed. On successful exit, `t_got` = `t_want`. If, however, `error%level` = 2, a solution has still been computed at `t_got` but `t_got` ≠ `t_want`.

y_got(n) — real(kind=*wp*), intent(out)

Output: an approximation to the true solution at the value of `t_got`. At each step of the integration to `t_got`, the local error has been controlled as specified in `nag_rk_setup`. The local error has still been controlled even if `error%level` = 2.

yp_got(n) — real(kind=*wp*), intent(out)

Output: an approximation to the first derivative of the true solution at `t_got`.

comm — type(nag_rk_comm_*wp*), intent(inout)

Input: information required by the procedure in order to perform the integration.

Output: information necessary for a subsequent call to any of the procedures in the module.

3.2 Optional Argument

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document `nag_error_handling` (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to `nag_set_error` before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	**Description**
301	An input argument has an invalid value.
302	An array argument has an invalid shape.
306	Invalid sequence of calls. For example, the last call to this procedure returned with an error and you are calling it again with the same argument `comm`.

Failures (error%level = 2):

error%code	**Description**
201	It does not appear possible to achieve the accuracy specified by tol and thresh in the call to nag_rk_setup. You cannot continue integrating this problem. A larger value for method, if possible, will permit greater accuracy. To increase method and/or continue with larger values of tol and/or thresh, restart the integration from t_got, y_got by a call to nag_rk_setup.
202	The global error assessment may not be reliable beyond the current integration point t_got. This error can only occur if global_err =.true. in the call to nag_rk_setup. It may occur because either too little or too much accuracy has been requested or because f is not smooth enough beyond t_got for the current solution y_got. The integration cannot be continued. This error return does not mean that you cannot integrate beyond t_got, rather that you cannot do it with global_err = .true.. However, it may also indicate problems with the primary integration.
203	This procedure is being used inefficiently because the step size has been reduced drastically many times to get answers at many points t_want. This return is possible only when method = 3 has been selected in the preceding call of nag_rk_setup. The integration was interrupted, so t_got $\neq$ t_want. If you really need the solution at this many points, you should change to method = 2 because it is (much) more efficient in this situation. To change method, restart the integration from t_got, y_got by a call to nag_rk_setup. If you wish to continue on towards t_want with method = 3, just call this procedure again without altering any of the arguments other than error. The internal monitor of this kind of inefficiency will be reset automatically so that the integration can proceed.
204	A considerable amount of work has been expended in the (primary) integration. This is measured by counting the number of calls to the function f. At least 5000 calls have been made since the last time this counter was reset. Calls to f in a secondary integration for global error assessment are not counted in this total. The integration was interrupted, so t_got $\neq$ t_want. If you wish to continue on towards t_want, just call this procedure again without altering any of the arguments other than error. The counter measuring work will be reset to zero automatically.
205	It appears that this problem is 'stiff'. The methods implemented in this procedure can solve such problems, but they are inefficient. The integration was interrupted so t_got $\neq$ t_want. If you want to continue on towards t_want, just call this procedure again without altering any of the arguments other than error. The internal stiffness monitor will be reset automatically.

5 Examples of Usage

A complete example of the use of this procedure appears in Example 1 of this module document.

Assume that all relevant arguments have been declared correctly as described in Section 3, and that input and input/output arguments have been appropriately initialized.
The following code fragment illustrates the use of this procedure:

```
...
CALL nag_rk_setup( t_start,y_start,t_end,tol,thresh,comm )
DO
   t_want = ...
   IF ( t_want > t_end ) EXIT
   CALL nag_rk_interval( f,t_want,t_got,y_got,yp_got,comm )

   PRINT*, t_got, y_got
END DO
        ...
```

6 Further Comments

6.1 Accuracy

The accuracy of the solution is determined by the arguments `tol` and `thresh` in a previous call to `nag_rk_setup` (see Section 6 of the procedure document for `nag_rk_setup` for further details and advice). 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 properties of the differential system. If you wish to assess the true error, you must set `global_err` = `.true.` in the call to `nag_rk_setup`. This assessment can be obtained after any call to `nag_rk_interval` by a call to the procedure `nag_rk_global_err`.

If `nag_rk_interval` returns with `error%code` = 201 and the accuracy specified by `tol` and `thresh` is really required, then you should consider whether there is a more fundamental difficulty. For example, the solution may contain a singularity. In such a region the solution components will usually be of a large magnitude. Successive output values of `y_got` should be monitored (or the procedure `nag_rk_step` should be used since this takes one integration step at a time) with the aim of trapping the solution before the singularity. In any case numerical solution cannot be continued through a singularity, and analytical treatment may be necessary.

Performance statistics are available after any return from this procedure by a call to the procedure `nag_rk_info` (if `error%level` ≤ 2). If `global_err` = `.true.` in the call to `nag_rk_setup`, global error assessment is available after return from `nag_rk_interval` (if `error%level` ≤ 2) by a call to the procedure `nag_rk_global_err`.

After a failure with `error%code` = 201 or 202 the diagnostic procedures `nag_rk_info` and `nag_rk_global_err` may be called only once.

If `nag_rk_interval` returns with `error%code` = 205, then it is advisable to change to another code more suited to the solution of stiff problems.

Procedure: nag_rk_info

1 Description

`nag_rk_info` provides details about an integration performed by either `nag_rk_interval` or `nag_rk_step`.

Note: all the output arguments of this procedure are optional. However, at least one output argument must be present in every call statement.

2 Usage

```
USE nag_ivp_ode_rk
CALL nag_rk_info(comm  [, optional arguments])
```

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '$\mathbf{x}(n)$' is used in the argument descriptions to specify that the array `x` must have exactly n elements.

The procedure derives the value of the following problem parameter from the argument `comm`.

$n \geq 1$ — the number of equations

3.1 Mandatory Argument

comm — type(nag_rk_comm_*wp*), intent(inout)

Input: information required by the procedure in order to provide details about the integration.

Output: information necessary for a subsequent call to any of the procedures in the module.

3.2 Optional Arguments

Note. Optional arguments must be supplied by keyword, not by position. The order in which they are described below may differ from the order in which they occur in the argument list.

num_fun_eval — integer, intent(out), optional

Output: the total number of evaluations of f used in the primary integration so far; this does not include evaluations of f for the secondary integration specified by a prior call to `nag_rk_setup` with `global_err = .true.`.

step_cost — integer, intent(out), optional

Output: the cost in terms of the number of evaluations of f of a typical step with the method being used for the integration.

waste — real(kind=*wp*), intent(out), optional

Output: the ratio of the number of attempted steps that failed to meet the local error requirement to the total number of steps attempted so far in the integration. A 'large' fraction indicates that the integrator is having trouble with the problem being solved. This can happen when the problem is 'stiff' and also when the solution has discontinuities in a low order derivative.

num_steps_ok — integer, intent(out), optional

Output: the number of accepted steps.

h_next — real(kind=wp), intent(out), optional

Output: the step size the integrator plans to use for the next step.

y_max(n) — real(kind=wp), intent(out), optional

Output: y_max(i) contains the largest value of $|y_i|$ computed at any step in the integration so far.

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document nag_error_handling (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to nag_set_error before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
302	An array argument has an invalid shape.
305	Invalid absence of an optional argument.
306	Invalid sequence of calls. For example, without a previous call to one of the integrators nag_rk_interval or nag_rk_step, the last call to the integrator returned with an error or a second call to this procedure without an intermediate call to an integrator.

5 Examples of Usage

Complete examples of the use of this procedure appear in Examples 1, 2 and 3 of this module document.

6 Further Comments

When a secondary integration has taken place, that is when global error assessment has been specified using global_err = .true. in a previous call to nag_rk_setup, then the approximate extra number of evaluations of f required is given by $2 \times$ num_steps_ok $\times$ step_cost for method $= 2$ or 3 and $3 \times$ num_steps_ok $\times$ step_cost for method $= 1$.

Procedure: nag_rk_global_err

1 Description

nag_rk_global_err provides details about global error assessment computed during an integration.

After a call to nag_rk_interval or nag_rk_step, this procedure can be called for information about error assessment if this assessment was specified in the setup procedure nag_rk_setup. After computing an approximate solution y in the primary integration, a more accurate 'true' solution $\hat{y}$ is computed in a secondary integration. The error is computed as specified in nag_rk_setup for local error control. At each step in the primary integration, an average magnitude η_i of component y_i is computed, and the estimate of the relative global error in the component is

$$\frac{|y_i - \hat{y}_i|}{\max(\eta_i, \texttt{thresh}(i))}.$$

It is difficult to estimate reliably the global error at a single point. For this reason the root-mean-square of the estimated relative global error in each solution component is computed. This average is taken over all steps from the beginning of the integration through to the current integration point. If all has gone well, the average errors reported will be comparable to the required accuracy (i.e., tol in procedure nag_rk_setup). The maximum error seen in any component in the integration so far and the point where the maximum error first occurred are also reported.

Note: all the output arguments of this procedure are optional. However, at least one output argument must be present in every call statement.

2 Usage

```
USE nag_ivp_ode_rk
CALL nag_rk_global_err(comm  [, optional arguments])
```

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '$\mathbf{x}(n)$' is used in the argument descriptions to specify that the array x must have exactly n elements.

The procedure derives the value of the following problem parameter from the argument comm.

$n \geq 1$ — the number of equations

3.1 Mandatory Argument

comm — type(nag_rk_comm_*wp*), intent(inout)

Input: information required by the procedure in order to evaluate the global error.

Output: information necessary for a subsequent call to any of the procedures in the module.

3.2 Optional Arguments

Note. Optional arguments must be supplied by keyword, not by position. The order in which they are described below may differ from the order in which they occur in the argument list.

err_max — real(kind=*wp*), intent(out), optional

Output: the new maximum approximate relative global error observed over all solution components and all steps.

t_err_max — real(kind=wp), intent(out), optional

Output: the first value of the independent variable where an approximate relative true error attains the maximum value, `err_max`.

rms_err(n) — real(kind=wp), intent(out), optional

Output: `rms_err`(i) contains the root-mean-square of the relative global error in the primary integration for the ith solution component, for $i = 1, 2, \ldots, n$. The average is taken over all steps from the beginning of the integration to the current integration point.

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document `nag_error_handling` (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to `nag_set_error` before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
302	An array argument has an invalid shape.
305	Invalid absence of an optional argument.
306	Invalid sequence of calls. For example, without a previous call to one of the integrators `nag_rk_interval` or `nag_rk_step`, the last call to the integrator returned with an error, the last call to the integrator did not take any successful step or a second call to this procedure without an intermediate call to an integrator.

5 Examples of Usage

A complete example of the use of this procedure appears in Example 2 of this module document.

6 Further Comments

If the integration has proceeded well and the problem is smooth enough, stable and not too difficult then the values returned in the arguments `rms_err` and `err_max` should be comparable to the value of `tol` specified in the previous call to `nag_rk_setup`.

Procedure: nag_rk_step

1 Description

nag_rk_step is a procedure for solving the initial value problem for a system of first-order ordinary differential equations one step at a time, using a Runge–Kutta method. It is designed to be used in complex tasks when solving systems of ordinary differential equations. You must first call this nag_rk_setup with call_step = .true. to specify the problem and how it is to be solved. Thereafter you call nag_rk_step repeatedly to take one integration step at a time from t_start in the direction of t_end (as specified in nag_rk_setup), with the size of each step being selected internally. In this manner nag_rk_step returns an approximation to the solution and its derivative at successive points. Between steps you may perform complex tasks, such as locating a root of a function of the solution. If nag_rk_step encounters some difficulty in taking a step, the integration is not advanced and the procedure returns with the same solution as returned on the previous successful step. nag_rk_step tries to advance the integration as far as possible subject to passing the test on the local error and not going beyond t_end. In the call to nag_rk_setup you can specify the first step size for nag_rk_step to attempt. Alternatively nag_rk_step will compute an appropriate first step. Thereafter nag_rk_step estimates an appropriate step size for its next step. This value and other details of the integration can be obtained after any call to nag_rk_step by a call to the procedure nag_rk_info. The local error is controlled at every step as specified in nag_rk_setup. If you wish to assess the global error, you must set global_err = .true. in the call to nag_rk_setup. This assessment can be obtained after any call to nag_rk_step by a call to the procedure nag_rk_global_err.

2 Usage

```
USE nag_ivp_ode_rk

CALL nag_rk_step(f, t_now, y_now, yp_now, comm  [, optional arguments])
```

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as 'x(n)' is used in the argument descriptions to specify that the array x must have exactly n elements.

This procedure derives the value of the following problem parameter from the shape of the supplied arrays.

$n \geq 1$ — the number of equations

3.1 Mandatory Arguments

f — function

The function f, supplied by the user must evaluate the functions f_i (that is the first derivatives y_i') for the given values of the arguments t and y_i.

The specification of f is:

```
function f(t, y)

real(kind=wp), intent(in) ::  t
```
Input: the current value of the independent variable, t.

```
real(kind=wp), intent(in) ::  y(:)
```
Shape: `y` has shape (n).

Input: the current values of the dependent variables, y_i, for $i = 1, 2, \ldots, n$.

```
real(kind=wp) ::  f(SIZE(y))
```
Result: the values of the first derivatives, y_i', for $i = 1, 2, \ldots, n$.

t_now — real(kind=*wp*), intent(out)

Output: the value of the independent variable, t, where a solution has been computed.

y_now(n) — real(kind=*wp*), intent(out)

Output: an approximation to the solution at `t_now`. The local error of the step to `t_now` was no greater than permitted by the specified tolerances (see `nag_rk_setup`).

yp_now(n) — real(kind=*wp*), intent(out)

Output: an approximation to the derivative of the solution at `t_now`.

comm — type(nag_rk_comm_*wp*), intent(inout)

Input: information required by the procedure in order to perform the integration.

Output: information necessary for a subsequent call to any of the procedures in the module.

3.2 Optional Argument

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document `nag_error_handling` (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to `nag_set_error` before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
302	An array argument has an invalid shape.
306	Invalid sequence of calls. For example the last call to this procedure returned with an error and you are calling it again with the same argument `comm`.

Failures (error%level = 2):

error%code	**Description**
201	It does not appear possible to achieve the accuracy specified by `tol` and `thresh` in the call to `nag_rk_setup`. You cannot continue integrating this problem. A larger value for `method`, if possible, will permit greater accuracy. To increase `method` and/or continue with larger values of `tol` and/or `thresh`, restart the integration from `t_now`, `y_now` by a call to `nag_rk_setup`.
202	The global error assessment may not be reliable beyond the current integration point `t_now`. This error can only occur if `global_err` = `.true.` in the call to `nag_rk_setup`. It may occur because either too little or too much accuracy has been requested or because f is not smooth enough beyond `t_now` for the current solution `y_now`. The integration cannot be continued. This return does not mean that you cannot integrate beyond `t_now`, rather that you cannot do it with `global_err` = `.true.`. However, it may also indicate problems with the primary integration.
203	This procedure is being used inefficiently because the step size has been reduced drastically many times to get answers at many points `t_end`. If you really need the solution at this many points, you should use `nag_rk_interp` to get the answers inexpensively. If you need to change from `method` = 3 to do this, restart the integration from `t_now`, `y_now` by a call to `nag_rk_setup`. If you wish to continue as before, call `nag_rk_step` again. The internal monitor of this kind of inefficiency will be reset automatically so that the integration can proceed.
204	A considerable amount of work has been expended in the (primary) integration. This is measured by counting the number of calls to the procedure `f`. At least 5000 calls have been made since the last time this counter was reset. Calls to `f` in a secondary integration for global error assessment (when `global_err` = `.true.` in the initialization call to `nag_rk_setup`) are not counted in this total. The integration was interrupted. If you wish to continue on towards `t_end`, just call `nag_rk_step` again. The counter measuring work will be reset to zero automatically.
205	It appears that this problem is 'stiff'. The methods implemented in this procedure can solve such problems, but they are inefficient. The integration was interrupted. If you want to continue on towards `t_end`, just call this procedure again. The internal stiffness monitor will be reset automatically.

5 Examples of Usage

Complete examples of the use of this procedure appear in Examples 2 and 3 of this module document.

Assume that all relevant arguments have been declared correctly as described in Section 3, and that input and input/output arguments have been appropriately initialized.
The following code fragment illustrates the use of this procedure:

```
...
CALL nag_rk_setup( t_start,y_start,t_end,tol,thresh,comm,call_step=.TRUE. )
DO
   CALL nag_rk_step( f,t_now,y_now,yp_now,comm )
   PRINT*, t_now, y_now
          ...
   IF (t_now == t_end) EXIT
END DO
        ...
```

6 Further Comments

If you want answers at specific points there are two ways to proceed.

1. The more efficient way is to step beyond the point where a solution is desired, and then call `nag_rk_interp` to get an answer there. Within the span of the current step, you can get all the answers you want at very little cost by repeated calls to `nag_rk_interp`. This is very valuable when you want to find 'special events', e.g.,, where a particular solution component vanishes. You cannot proceed in this way with `method` = 3.

2. The other way to get an answer at a specific point is to set `t_end` to this value and integrate to `t_end`. This procedure will not step beyond `t_end`, so when a step would carry it beyond, it will reduce the step size so as to produce an answer at `t_end` exactly. After getting an answer there (`t_now` = `t_end`), you can reset `t_end` to the next point where you want an answer and repeat. `t_end` could be reset by a call to `nag_rk_setup`, but you should not do this. You should instead use `nag_rk_reset_end` because it is both easier to use and much more efficient. This way of getting answers at specific points can be used with any of the available methods, but it is the only way with `method` = 3. This can be inefficient. Should this be the case, the code will bring the matter to your attention.

6.1 Accuracy

The accuracy of the solution is determined by the arguments `tol` and `thresh` in a previous call to `nag_rk_setup` (see Section 6 of the procedure document for `nag_rk_setup` for further details and advice). 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 properties of the differential system.

If `nag_rk_step` returns with `error%code` = 201 and the accuracy specified by `tol` and `thresh` is really required, then you should consider whether there is a more fundamental difficulty. For example, the solution may contain a singularity. In such a region the solution components will usually be of a large magnitude. Successive output values of `y_now` should be monitored with the aim of trapping the solution before the singularity. In any case numerical solution cannot be continued through a singularity, and analytical treatment may be necessary.

Performance statistics are available after return from this procedure (if `error%level` ≤ 2) by a call to the procedure `nag_rk_info`. If `global_err = .true.` in the call to `nag_rk_setup`, global error assessment is available after any return from `nag_rk_step` (if `error%level` ≤ 2) by a call to the procedure `nag_rk_global_err`.

After a failure with `error%code` = 201 or 202 the diagnostic procedures `nag_rk_info` and `nag_rk_global_err` may be called only once.

If this procedure returns with `error%code` = 205, then it is advisable to change to another code more suited to the solution of stiff problems.

Procedure: nag_rk_interp

1 Description

`nag_rk_interp` is a procedure to compute an approximate solution of a system of ordinary differential equations using interpolation anywhere within an integration step taken by `nag_rk_step`. The procedure is only capable of performing the interpolation for `method` = 1 and 2. The accuracy of the computed values will be similar to that computed by `nag_rk_step`.

Note: all the output arguments of this procedure are optional. However, at least one output argument must be present in every call statement.

2 Usage

```
USE nag_ivp_ode_rk

CALL nag_rk_interp(f, t_want, comm  [, optional arguments])
```

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '$\mathbf{x}(n)$' is used in the argument descriptions to specify that the array `x` must have exactly n elements.

This procedure derives the values of the following problem parameters from the shape of the supplied arrays.

$1 \leq l \leq n$ — the number of required components of the solution

$1 \leq m \leq n$ — the number of required components of the first derivative

The parameter n is obtained from the argument `comm`.

3.1 Mandatory Arguments

f — function

The function `f`, supplied by the user must evaluate the functions f_i (that is the first derivatives y_i') for the given values of the arguments t and y_i.

The specification of `f` is:

```
function f(t, y)

real(kind=wp), intent(in) ::  t
```
Input: the current value of the independent variable, t.

```
real(kind=wp), intent(in) ::  y(:)
```
Shape: `y` has shape (n).

Input: the current values of the dependent variables, y_i, for $i = 1, 2, \ldots, n$.

```
real(kind=wp) ::  f(SIZE(y))
```
Result: the values of the first derivatives, y_i', for $i = 1, 2, \ldots, n$.

t_want — real(kind=*wp*), intent(in)

Input: the value of the independent variable, t, where a solution is required.

comm — type(nag_rk_comm_*wp*), intent(inout)

Input: information required by the procedure in order to perform the interpolation.

Output: information necessary for a subsequent call to any of the procedures in the module.

3.2 Optional Arguments

Note. Optional arguments must be supplied by keyword, not by position. The order in which they are described below may differ from the order in which they occur in the argument list.

y_want(l) — real(kind=*wp*), intent(out), optional

Output: an approximation to the first l components of the solution at `t_want`.

yp_want(m) — real(kind=*wp*), intent(out), optional

Output: an approximation to the first m components of the first derivative at `t_want`.

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document `nag_error_handling` (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to `nag_set_error` before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
302	An array argument has an invalid shape.
305	Invalid absence of an optional argument.
306	Invalid sequence of calls.
	For example, without a previous call to the integrator `nag_rk_step`, the last call to the integrator returned with an error, or the last call to the integrator `nag_rk_step` was being used with `method` = 3.

5 Examples of Usage

A complete example of the use of this procedure appears in Example 2 of this module document.

6 Further Comments

This procedure cannot be used in conjunction with `nag_rk_interval`.

Procedure: nag_rk_reset_end

1 Description

`nag_rk_reset_end` is used to reset the final value of the independent variable, t_e, when the integration is already underway in a step by step manner using `nag_rk_step`. It can also be used to extend or reduce the range of integration. The new value must be beyond the current value of the independent variable (as returned in `t_now` by `nag_rk_step`) in the current direction of integration. It is much more efficient to use this procedure for this purpose than to use `nag_rk_setup`, which involves the overhead of a complete restart of the integration.

If you want to change the direction of integration then you must restart by a call to `nag_rk_setup`.

2 Usage

```
USE nag_ivp_ode_rk

CALL nag_rk_reset_end(t_end_new, comm  [, optional arguments])
```

3 Arguments

3.1 Mandatory Arguments

t_end_new — real(kind=*wp*), intent(in)

Input: the new value of the independent variable, t_e.

Constraints: sign(`t_end_new` − `t_now`) = sign(`t_end` − `t_start`), where `t_start` and `t_end` are as supplied in the previous call to `nag_rk_setup` and `t_now` is returned by the preceding call to `nag_rk_step`. Note that `t_end_new` must be distinguishable from `t_now` for the method and the precision of the machine being used.

comm — type(nag_rk_comm_*wp*), intent(inout)

Input: information required by the procedure in order to reset the integration end point.

Output: information necessary for a subsequent call to any of the procedures in the module.

3.2 Optional Argument

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document `nag_error_handling` (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to `nag_set_error` before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
306	Invalid sequence of calls. For example, without a previous call to the integrator `nag_rk_step` or the last call to the integrator returned with an error.

5 Examples of Usage

A complete example of the use of this procedure appears in Example 3 of this module document.

6 Further Comments

This procedure cannot be used in conjunction with `nag_rk_interval`.

Derived Type: nag_rk_comm_*wp*

Note. The names of derived types containing real/complex components are precision dependent. For double precision the name of this type is `nag_rk_comm_dp`. For single precision the name is `nag_rk_comm_sp`. Please read the Users' Note for your implementation to check which precisions are available.

1 Description

A structure of the type `nag_rk_comm_`*wp* is used to communicate information between the various procedures contained in this module.

Some procedures in the module allocate storage to the pointer components of the structure. For details of the amount of storage allocated see the description of the argument `comm` in the procedure documents for `nag_rk_setup` and `nag_rk_interp`.

If you wish to deallocate the storage when the integration is complete and the structure is no longer required, you must call the generic deallocation procedure `nag_deallocate`, which is described in the module document `nag_lib_support` (1.1).

The components of this type are private.

2 Type Definition

```
type nag_rk_comm_wp
  private
  .
  .
  .
end type nag_rk_comm_wp
```

3 Components

In order to reduce the risk of accidental data corruption the components of this type are private and may not be accessed directly.

Example 1: Integration over an interval

We solve the equation

$$y'' = -y$$

over the range $[0,\ 2\pi]$, with initial conditions $y(0) = 0,\ y'(0) = 1$. This is reposed as

$$y_1' = y_2, \quad y_2' = -y_1$$

with initial conditions $y_1(0) = 0.0$ and $y_2(0) = 1.0$. We use relative error control with threshold values of 10^{-8} for each solution component and compute the solution at intervals of length $\pi/4$ across the range. We use the default Runge–Kutta method (i.e., `method` $= 2$) with `tol` $= 10^{-3}$ and `tol` $= 10^{-4}$ in turn so that we may compare the solutions.

1 Program Text

Note. The listing of the example program presented below is double precision. Single precision users are referred to Section 5.2 of the Essential Introduction for further information.

```
MODULE ivp_ode_rk_ex01_mod

  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)

CONTAINS

  FUNCTION f(t,y)

    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Intrinsic Functions ..
    INTRINSIC SIZE
    ! .. Scalar Arguments ..
    REAL (wp), INTENT (IN) :: t
    ! .. Array Arguments ..
    REAL (wp), INTENT (IN) :: y(:)
    ! .. Function Return Value ..
    REAL (wp) :: f(SIZE(y))
    ! .. Executable Statements ..

    f(1) = y(2)
    f(2) = -y(1)

  END FUNCTION f

END MODULE ivp_ode_rk_ex01_mod

PROGRAM nag_ivp_ode_rk_ex01

  ! Example Program Text for nag_ivp_ode_rk
  ! NAG fl90, Release 3. NAG Copyright 1997.

  ! .. Use Statements ..
  USE nag_ivp_ode_rk, ONLY : nag_rk_comm_wp => nag_rk_comm_dp, &
   nag_rk_setup, nag_rk_interval, nag_rk_info, nag_deallocate
  USE nag_math_constants, ONLY : nag_pi
  USE nag_examples_io, ONLY : nag_std_out
```

```
    USE ivp_ode_rk_ex01_mod, ONLY : wp, f
    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Parameters ..
    INTEGER, PARAMETER :: n = 2
    ! .. Local Scalars ..
    INTEGER :: i, j, nout, num_fun_eval
    REAL (wp) :: pi, tinc, tol, t_end, t_got, t_start, t_want
    TYPE (nag_rk_comm_wp) :: comm
    ! .. Local Arrays ..
    REAL (wp) :: thresh(n), yp_got(n), y_got(n), y_start(n)
    ! .. Executable Statements ..

    WRITE (nag_std_out,*) 'Example Program Results for nag_ivp_ode_rk_ex01'

    pi = nag_pi(0.0_wp)

    t_start = 0.0_wp
    t_end = 2.0_wp*pi
    y_start = (/ 0.0_wp, 1.0_wp/)
    thresh = 1.0E-10_wp

    nout = 8
    tinc = (t_end-t_start)/nout

    DO i = 1, 2
      IF (i==1) tol = 1.0E-3_wp
      IF (i==2) tol = 1.0E-4_wp

      CALL nag_rk_setup(t_start,y_start,t_end,tol,thresh,comm)

      WRITE (nag_std_out,'(/1X,A,1PE8.1)') 'Calculation with tol = ', tol
      WRITE (nag_std_out,*) '   t           y1           y2'
      WRITE (nag_std_out,'(1x,F6.3,2(4X,F9.5))') t_start, y_start(:)

      DO j = nout - 1, 0, -1
        t_want = t_end - j*tinc

        CALL nag_rk_interval(f,t_want,t_got,y_got,yp_got,comm)

        WRITE (nag_std_out,'(1X,F6.3,2(4X,F9.5))') t_got, y_got(:)
      END DO

      CALL nag_rk_info(comm,num_fun_eval=num_fun_eval)

      WRITE (nag_std_out,'(1X,A,I10)') &
       'The cost of integration in evaluations of f = ', num_fun_eval
    END DO

    CALL nag_deallocate(comm)    ! Free structure allocated by NAG fl90

  END PROGRAM nag_ivp_ode_rk_ex01
```

2 Program Data

None.

3 Program Results

```
Example Program Results for nag_ivp_ode_rk_ex01

Calculation with tol =  1.0E-03
   t           y1           y2
 0.000      0.00000      1.00000
 0.785      0.70711      0.70710
 1.571      0.99996     -0.00008
 2.356      0.70696     -0.70715
 3.142     -0.00022     -0.99991
 3.927     -0.70723     -0.70679
 4.712     -0.99986      0.00034
 5.498     -0.70675      0.70725
 6.283      0.00037      0.99984
The cost of integration in evaluations of f =         68

Calculation with tol =  1.0E-04
   t           y1           y2
 0.000      0.00000      1.00000
 0.785      0.70711      0.70710
 1.571      1.00000     -0.00001
 2.356      0.70710     -0.70711
 3.142     -0.00001     -0.99999
 3.927     -0.70711     -0.70709
 4.712     -0.99998      0.00002
 5.498     -0.70707      0.70711
 6.283      0.00003      0.99998
The cost of integration in evaluations of f =        102
```

Example 2: Interpolation across an interval with global error assessment

We solve the equation

$$y'' = -y$$

over the range $[0,\ 2\pi]$, with initial conditions $y(0) = 0,\ y'(0) = 1$. This is reposed as

$$y_1' = y_2, \quad y_2' = -y_1$$

with initial conditions $y_1(0) = 0.0$ and $y_2(0) = 1.0$. We use relative error control with threshold values of 10^{-8} for each solution component. `nag_rk_step` is used to integrate the problem one step at a time and `nag_rk_interp` is used to compute the first component of the solution and its derivative at intervals of length $\pi/8$ across the range whenever these points lie in one of those integration steps. We use the default Runge–Kutta method (`method` = 2) with `tol` = 10^{-3} and `tol` = 10^{-4} in turn so that we may compare the solutions. With each value of the above tolerances a global error assessment is performed and is reported using `nag_rk_global_err`.

1 Program Text

Note. The listing of the example program presented below is double precision. Single precision users are referred to Section 5.2 of the Essential Introduction for further information.

```
MODULE ivp_ode_rk_ex02_mod

  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)

CONTAINS

  FUNCTION f(t,y)

    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Intrinsic Functions ..
    INTRINSIC SIZE
    ! .. Scalar Arguments ..
    REAL (wp), INTENT (IN) :: t
    ! .. Array Arguments ..
    REAL (wp), INTENT (IN) :: y(:)
    ! .. Function Return Value ..
    REAL (wp) :: f(SIZE(y))
    ! .. Executable Statements ..

    f(1) = y(2)
    f(2) = -y(1)

  END FUNCTION f

END MODULE ivp_ode_rk_ex02_mod

PROGRAM nag_ivp_ode_rk_ex02

  ! Example Program Text for nag_ivp_ode_rk
  ! NAG fl90, Release 3. NAG Copyright 1997.
```

```
      ! .. Use Statements ..
      USE nag_ivp_ode_rk, ONLY : nag_rk_comm_wp => nag_rk_comm_dp, &
       nag_rk_setup, nag_rk_step, nag_rk_info, nag_rk_interp, &
       nag_rk_global_err, nag_deallocate, nag_error, nag_set_error
      USE nag_math_constants, ONLY : nag_pi
      USE nag_examples_io, ONLY : nag_std_out
      USE ivp_ode_rk_ex02_mod, ONLY : wp, f
      ! .. Implicit None Statement ..
      IMPLICIT NONE
      ! .. Parameters ..
      INTEGER, PARAMETER :: n = 2
      ! .. Local Scalars ..
      INTEGER :: i, j, nout, num_fun_eval
      REAL (wp) :: err_max, pi, tinc, tol, t_end, t_err_max, t_now, t_start, &
       t_want
      LOGICAL :: call_step, global_err
      TYPE (nag_rk_comm_wp) :: comm
      TYPE (nag_error) :: error_rec
      ! .. Local Arrays ..
      REAL (wp) :: rms_err(n), thresh(n), yp_now(n), yp_want(1), y_now(n), &
       y_start(n), y_want(1)
      ! .. Executable Statements ..

      WRITE (nag_std_out,*) 'Example Program Results for nag_ivp_ode_rk_ex02'

      CALL nag_set_error(error_rec)

      pi = nag_pi(0.0_wp)

      t_start = 0.0_wp
      t_end = 2.0_wp*pi
      y_start = (/ 0.0_wp, 1.0_wp/)
      thresh = 1.0E-8_wp
      call_step = .TRUE.
      global_err = .TRUE.

      nout = 16
      tinc = (t_end-t_start)/nout

      DO i = 1, 2
        IF (i==1) tol = 1.0E-3_wp
        IF (i==2) tol = 1.0E-4_wp

        CALL nag_rk_setup(t_start,y_start,t_end,tol,thresh,comm, &
         call_step=call_step,global_err=global_err)

        WRITE (nag_std_out,'(/1X,A,1PE8.1)') 'Calculation with tol = ', tol
        WRITE (nag_std_out,*) '    t         y1        y2'
        WRITE (nag_std_out,'(1X,F7.4,2(3X,F7.4))') t_start, y_start(:)

        j = nout - 1
        t_want = t_end - j*tinc

inner:  DO

          CALL nag_rk_step(f,t_now,y_now,yp_now,comm,error=error_rec)

          IF (error_rec%code==0) THEN

interp:     DO
              IF (t_want>t_now) EXIT interp
              CALL nag_rk_interp(f,t_want,comm,y_want=y_want,yp_want=yp_want)

              WRITE (nag_std_out,'(1X,F7.4,2(3X,F7.4))') t_want, y_want(1), &
               yp_want(1)
              j = j - 1
              t_want = t_end - j*tinc
            END DO interp

            IF (t_now>=t_end) EXIT inner

          END IF

        END DO inner

        CALL nag_rk_global_err(comm,rms_err=rms_err,err_max=err_max, &
         t_err_max=t_err_max)

        WRITE (nag_std_out,*)
        WRITE (nag_std_out,'(1X,A/1X,2(2X,E9.2))') &
         'Componentwise error assessment:', rms_err(:)
        WRITE (nag_std_out,'(1X,A,E9.2,A,F7.4)') &
         'Worst global error observed was ', err_max, &
         ' - it occurred at  t = ', t_err_max

        CALL nag_rk_info(comm,num_fun_eval=num_fun_eval)

        WRITE (nag_std_out,*)
        WRITE (nag_std_out,'(1X,A,I10)') &
         'The cost of integration in evaluations of f = ', num_fun_eval
      END DO

      CALL nag_deallocate(comm)    ! Free structure allocated by NAG fl90

    END PROGRAM nag_ivp_ode_rk_ex02
```

2 Program Data

None.

3 Program Results

```
 Example Program Results for nag_ivp_ode_rk_ex02

 Calculation with tol =  1.0E-03
     t         y1        y2
  0.0000    0.0000    1.0000
  0.3927    0.3827    0.9239
  0.7854    0.7071    0.7071
  1.1781    0.9239    0.3826
  1.5708    1.0000   -0.0001
  1.9635    0.9238   -0.3828
  2.3562    0.7070   -0.7073
  2.7489    0.3825   -0.9240
  3.1416   -0.0002   -0.9999
  3.5343   -0.3829   -0.9238
  3.9270   -0.7072   -0.7069
  4.3197   -0.9239   -0.3823
  4.7124   -0.9999    0.0004
  5.1051   -0.9236    0.3830
  5.4978   -0.7068    0.7073
  5.8905   -0.3823    0.9239
  6.2832    0.0004    0.9998

 Componentwise error assessment:
    0.44E-03   0.31E-03
 Worst global error observed was  0.90E-03 - it occurred at  t =  6.2832

 The cost of integration in evaluations of f =        152

 Calculation with tol =  1.0E-04
     t         y1        y2
  0.0000    0.0000    1.0000
  0.3927    0.3827    0.9239
  0.7854    0.7071    0.7071
  1.1781    0.9239    0.3827
  1.5708    1.0000    0.0000
  1.9635    0.9239   -0.3827
  2.3562    0.7071   -0.7071
  2.7489    0.3827   -0.9239
  3.1416    0.0000   -1.0000
  3.5343   -0.3827   -0.9239
  3.9270   -0.7071   -0.7071
  4.3197   -0.9239   -0.3827
  4.7124   -1.0000    0.0000
  5.1051   -0.9238    0.3827
  5.4978   -0.7071    0.7071
  5.8905   -0.3826    0.9239
  6.2832    0.0000    1.0000

 Componentwise error assessment:
    0.49E-04   0.17E-04
 Worst global error observed was  0.13E-03 - it occurred at  t =  6.2832

 The cost of integration in evaluations of f =        231
```

Example 3: Resetting the end point of integration

We integrate a two body problem. The equations for the coordinates $(x(t), y(t))$ of one body as functions of time t in a suitable frame of reference are

$$x'' = -x/r^3, \quad y'' = -y/r^3, \quad r = \sqrt{x^2 + y^2}.$$

The initial conditions

$$x(0) = 1 - \alpha, \quad x'(0) = 0, \quad y(0) = 0, \quad y'(0) = \sqrt{\frac{1+\alpha}{1-\alpha}},$$

lead to elliptic motion with $0 < \alpha < 1$. We select $\alpha = 0.7$ and reduce the two second-order differential equations to the following system of first-order differential equations:

$$y_1' = y_3, \quad y_2' = y_4, \quad y_3' = -y_1/r^3, \quad y_4' = -y_2/r^3$$

over the range $[0, 6\pi]$. We use relative error control with threshold values of 10^{-10} for each solution component and compute the solution at intervals of length π across the range using `nag_rk_reset_end` to reset the end of the integration range. We use a high-order Runge–Kutta method (`method` $= 3$) with `tol` $= 10^{-4}$ and `tol` $= 4.0 \times 10^{-5}$ in turn so that we may compare the solutions.

1 Program Text

Note. The listing of the example program presented below is double precision. Single precision users are referred to Section 5.2 of the Essential Introduction for further information.

```
MODULE ivp_ode_rk_ex03_mod

  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)

CONTAINS

  FUNCTION f(t,y)

    ! .. Implicit None Statement ..
    IMPLICIT NONE
    ! .. Intrinsic Functions ..
    INTRINSIC SIZE, SQRT
    ! .. Scalar Arguments ..
    REAL (wp), INTENT (IN) :: t
    ! .. Array Arguments ..
    REAL (wp), INTENT (IN) :: y(:)
    ! .. Function Return Value ..
    REAL (wp) :: f(SIZE(y))
    ! .. Local Scalars ..
    REAL (wp) :: r3
    ! .. Executable Statements ..

    r3 = (SQRT(y(1)**2+y(2)**2))**3
    f(1) = y(3)
    f(2) = y(4)
    f(3) = -y(1)/r3
    f(4) = -y(2)/r3

  END FUNCTION f
```

```
    END MODULE ivp_ode_rk_ex03_mod

    PROGRAM nag_ivp_ode_rk_ex03

      ! Example Program Text for nag_ivp_ode_rk
      ! NAG fl90, Release 3. NAG Copyright 1997.

      ! .. Use Statements ..
      USE nag_ivp_ode_rk, ONLY : nag_rk_comm_wp => nag_rk_comm_dp, &
       nag_rk_setup, nag_rk_step, nag_rk_info, nag_rk_reset_end, &
       nag_deallocate
      USE nag_math_constants, ONLY : nag_pi
      USE nag_examples_io, ONLY : nag_std_out
      USE ivp_ode_rk_ex03_mod, ONLY : wp, f
      ! .. Implicit None Statement ..
      IMPLICIT NONE
      ! .. Intrinsic Functions ..
      INTRINSIC SQRT
      ! .. Parameters ..
      INTEGER, PARAMETER :: n = 4
      ! .. Local Scalars ..
      INTEGER :: i, j, method, nout, num_fun_eval
      REAL (wp) :: alpha, pi, tinc, tol, t_end, t_final, t_now, t_start
      LOGICAL :: call_step
      TYPE (nag_rk_comm_wp) :: comm
      ! .. Local Arrays ..
      REAL (wp) :: thresh(n), yp_now(n), y_now(n), y_start(n)
      ! .. Executable Statements ..

      WRITE (nag_std_out,*) 'Example Program Results for nag_ivp_ode_rk_ex03'

      pi = nag_pi(0.0_wp)
      alpha = 0.7_wp

      t_end = 2.0_wp*pi
      t_final = 6.0_wp*pi
      thresh = 1.0E-10_wp
      t_start = 0.0_wp
      y_start = (/ 1.0_wp - alpha, 0.0_wp, 0.0_wp, SQRT((1.0_wp+alpha)/(1.0_wp &
       -alpha)) /)
      method = 3
      call_step = .TRUE.

      nout = 6
      tinc = t_final/nout

      DO i = 1, 2
        IF (i==1) tol = 1.0E-4_wp
        IF (i==2) tol = 4.0E-5_wp
        j = nout - 1
        t_end = t_final - j*tinc

        CALL nag_rk_setup(t_start,y_start,t_end,tol,thresh,comm,method=method, &
         call_step=call_step)

        WRITE (nag_std_out,'(/1X,A,1PE8.1)') 'Calculation with tol = ', tol
        WRITE (nag_std_out,*) &
         '   t         y1         y2         y3         y4'
        WRITE (nag_std_out,'(1X,F6.3,4(4X,F7.4))') t_start, y_start(:)

inner:  DO
```

```
          CALL nag_rk_step(f,t_now,y_now,yp_now,comm)

          IF (t_now>=t_end) THEN
            WRITE (nag_std_out,'(1X,F6.3,4(4X,F7.4))') t_now, y_now(:)
            IF (t_now>=t_final) EXIT inner
            j = j - 1
            t_end = t_final - j*tinc

            CALL nag_rk_reset_end(t_end,comm)

          END IF

        END DO inner

        CALL nag_rk_info(comm,num_fun_eval=num_fun_eval)

        WRITE (nag_std_out,'(1X,A,I10)') &
         'The cost of integration in evaluations of f = ', num_fun_eval
      END DO

      CALL nag_deallocate(comm)    ! Free structure allocated by NAG fl90

    END PROGRAM nag_ivp_ode_rk_ex03
```

2 Program Data

None.

3 Program Results

```
 Example Program Results for nag_ivp_ode_rk_ex03

 Calculation with tol =  1.0E-04
    t         y1         y2         y3         y4
  0.000     0.3000     0.0000     0.0000     2.3805
  3.142    -1.7000     0.0000     0.0000    -0.4201
  6.283     0.3000     0.0000     0.0001     2.3805
  9.425    -1.7000     0.0000     0.0000    -0.4201
 12.566     0.3000    -0.0003     0.0016     2.3805
 15.708    -1.7001     0.0001    -0.0001    -0.4201
 18.850     0.3000    -0.0010     0.0045     2.3805
 The cost of integration in evaluations of f =        571

 Calculation with tol =  4.0E-05
    t         y1         y2         y3         y4
  0.000     0.3000     0.0000     0.0000     2.3805
  3.142    -1.7000     0.0000     0.0000    -0.4201
  6.283     0.3000     0.0000     0.0001     2.3805
  9.425    -1.7000     0.0000     0.0000    -0.4201
 12.566     0.3000    -0.0002     0.0008     2.3805
 15.708    -1.7000     0.0001     0.0000    -0.4201
 18.850     0.3000    -0.0005     0.0021     2.3805
 The cost of integration in evaluations of f =        670
```

References

[1] 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

[2] Brankin R W, Gladwell I and Shampine L F (1991) RKSUITE: A suite of Runge–Kutta codes for the initial value problems for ODEs *SoftReport 91-S1* Southern Methodist University, Dallas, TX, USA