nag_ode_ivp_rk_setup (d02pvc)

1.   Purpose

nag_ode_ivp_rk_setup (d02pvc) is a setup function which must be called prior to the first call of either of the integration functions nag_ode_ivp_rk_range (d02pcc) and nag_ode_ivp_rk_onestep (d02pdc).

2.   Specification

```
#include <nag.h>
#include <nagd02.h>

void nag_ode_ivp_rk_setup(Integer neq, double tstart, double ystart[],
          double tend, double tol, double thres[],
          Nag_RK_method method, Nag_RK_task task, Nag_ErrorAssess errass,
          double hstart, Nag_ODE_RK *opt, NagError * fail)
```

3.   Description

This function and its associated functions (nag_ode_ivp_rk_range (d02pcc), nag_ode_ivp_rk_onestep (d02pdc), nag_ode_ivp_rk_reset_tend (d02pwc), nag_ode_ivp_rk_interp (d02pxc), nag_ode_ivp_rk_errass (d02pzc)) solve the initial value problem for a first order system of ordinary differential equations. The functions, based on Runge-Kutta methods and derived from RKSUITE (Brankin *et al*, 1991) integrate

$$y' = f(t,y) \qquad \text{given} \qquad y(t_0) = y_0$$

where y is the vector of **neq** solution components and t is the independent variable.

The integration proceeds by steps from the initial point t_0 towards the final point t_f. An approximate solution y is computed at each step. For each component $y_i, i = 1, 2, \ldots,$**neq** the error made in the step, i.e., the local error, is estimated. The step size is chosen automatically so that the integration will proceed efficiently while keeping this local error estimate smaller than a tolerance that you specify by means of parameters **tol** and **thres**.

nag_ode_ivp_rk_range (d02pcc) can be used to solve the 'usual task', namely integrating the system of differential equations to obtain answers at points you specify. nag_ode_ivp_rk_onestep (d02pdc) is used for all more 'complicated tasks'.

You should consider carefully how you want the local error to be controlled. Essentially the code uses relative local error control, with **tol** being the desired relative accuracy. For reliable computation, the code must work with approximate solutions that have some correct digits, so there is an upper bound on the value you can specify for **tol**. It is impossible to compute a numerical solution that is more accurate than the correctly rounded value of the true solution, so you are not allowed to specify a **tol** that is too small for the precision you are using. The magnitude of the local error in y_i on any step will not be greater than **tol** $\times max(\sigma_i,$ **thres**$[i-1])$ where σ_i is an average magnitude of y_i over the step. If **thres**$[i-1]$ is smaller than the current value of σ_i, this is a relative error test and **tol** indicates how many significant digits you want in y_i. If **thres**$[i-1]$ is larger than the current value of σ_i, this is an absolute error test with tolerance **tol** $\times$ **thres**$[i-1]$. Relative error control is the recommended mode of operation, but pure relative error control, **thres**$[i-1] = 0.0$, is not permitted. See Section 8 for further information about error control.

nag_ode_ivp_rk_range (d02pcc) and nag_ode_ivp_rk_onestep (d02pdc) control local error rather than the true (global) error, the difference between the numerical and true solution. Control of the local error controls the true error indirectly. Roughly speaking, the code produces a solution that satisfies the differential equation with a discrepancy bounded in magnitude by the error tolerance. What this implies about how close the numerical solution is 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. To judge the accuracy of the numerical solution, 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. Alternatively, a global error assessment can be computed automatically using the parameter **errass**. Because indirect control of the true error by controlling the local error is generally satisfactory and because both ways of assessing true errors cost twice, or more, the cost of the integration itself, such assessments are used mostly for spot checks, selecting appropriate tolerances for local error control, and exploratory computations.

nag_ode_ivp_rk_range (d02pcc) and nag_ode_ivp_rk_onestep (d02pdc) each implement three Runge-Kutta formula pairs, and you must select one for the integration. The best choice for **method** depends on the problem. The order of accuracy is 3,5,8, corresponding to **Nag_RK_2_3**, **Nag_RK_4_5** and **Nag_RK_7_8** respectively. As a rule, the lower the value of **tol**, the higher the order of accuracy of the **method**. If the components **thres** are small enough that you are effectively specifying relative error control, experience suggests

tol   efficient **method**

$10^{-2} - 10^{-4}$	**Nag_RK_2_3**
$10^{-3} - 10^{-6}$	**Nag_RK_4_5**
$10^{-5} -$	**Nag_RK_7_8**

The overlap in the ranges of tolerances appropriate for a given **method** merely reflects the dependence of efficiency on the problem being solved. Making **tol** smaller will normally make the integration more expensive. However, in the range of tolerances appropriate to a **method**, the increase in cost is modest. There are situations for which one **method**, or even this kind of code, is a poor choice. You should not specify a very small **thres**$[i-1]$, when the ith solution component might vanish. In particular, you should not do this when $y_i = 0.0$. If you do, the code will have to work hard with any **method** to compute significant digits, but **method** = **Nag_RK_2_3** is a particularly poor choice in this situation. All three **method**s are inefficient when the problem is 'stiff'. If it is only mildly stiff, you can solve it with acceptable efficiency with **method** = **Nag_RK_2_3**, but if it is moderately or very stiff, a code designed specifically for such problems will be much more efficient. The higher the order of accuracy of the **method**, the more smoothness is required of the solution for the **method** to be efficient.

When assessment of the true (global) error is requested, this error assessment is updated at each step. Its value can be obtained at any time by a call to nag_ode_ivp_rk_errass (d02pzc). 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 insufficiently 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 this is roughly twice the cost of the integration itself with **method** = **Nag_RK_4_5** or **Nag_RK_7_8** and three times with **method** = **Nag_RK_2_3**.

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 you set the variable **hstart** to 0.0. 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, provide it in the variable **hstart**.

4.   Parameters

neq
> Input: the number of ordinary differential equations in the system.
> Constraint: **neq** ≥ 1.

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

ystart[neq]
> Input: y_0, the initial values of the solution, y_i for $i = 1, 2, \ldots,$**neq**, at t_0.

tend

    Input: the final value of the independent variable, t_f, at which the solution is required. **tstart** and **tend** together determine the direction of integration.

    Constraint: **tend** must be distinguishable from **tstart** for the method and the precision of the machine being used.

tol

    Input: a relative error tolerance.

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

thres[neq]

    Input: a vector of thresholds.

    Constraint: **thres**$[i-1] \ge \sqrt{\epsilon}$, where ϵ is the smallest possible machine number (see X02AMC).

method

    Input: the Runge-Kutta method to be used.

    If **method** = **Nag_RK_2_3** then a 2(3) pair is used.

    If **method** = **Nag_RK_4_5** then a 4(5) pair is used.

    If **method** = **Nag_RK_7_8** then a 7(8) pair is used.

    Constraint: **method** = **Nag_RK_2_3** or **Nag_RK_4_5** or **Nag_RK_7_8**.

task

    Input: determines whether the usual integration task is to be performed using nag_ode_ivp_rk_range (d02pcc) or a more complicated task is to be performed using nag_ode_ivp_rk_onestep (d02pdc).

    If **task** = **Nag_RK_range** then nag_ode_ivp_rk_range (d02pcc) is to be used for the integration.

    If **task** = **Nag_RK_onestep** then nag_ode_ivp_rk_onestep (d02pdc) is to be used for the integration.

    Constraint: **task** = **Nag_RK_range** or **Nag_RK_onestep**

errass

    Input: specifies whether a global error assessment is to be computed with the main integration. **errass** = **Nag_ErrorAssess_on** specifies that it is.

    Constraint: **errass** = **Nag_ErrorAssess_on** or **Nag_ErrorAssess_off**

hstart

    Input: a value for the size of the first step in the integration to be attempted. The absolute value of **hstart** is used with the direction being determined by **tstart** and **tend**. The actual first step taken by the integrator may be different to **hstart** if the underlying algorithm determines that **hstart** is unsuitable. If **hstart** = 0.0 then the size of the first step is computed automatically.

    Suggested value: **hstart** = 0.0.

opt

    Output: the structure of type Nag_ODE_RK initialised to appropriate values and to be passed unchanged to the integration routines nag_ode_ivp_rk_range (d02pcc) or nag_ode_ivp_rk_onestep (d02pdc). Memory will have been allocated by nag_ode_ivp_rk_setup. This memory is used by nag_ode_ivp_rk_range (d02pcc), nag_ode_ivp_rk_onestep (d02pdc), nag_ode_ivp_rk_interp (d02pxc), nag_ode_ivp_rk_reset_tend (d02pwc) and nag_ode_ivp_rk_errass (d02pzc). The library function nag_ode_ivp_rk_free (d02ppc) is provided so that this memory can be freed by the user when the integration is complete or the setup function nag_ode_ivp_rk_setup (d02pvc) is to be re-entered.

fail

    The NAG error parameter, see the Essential Introduction to the NAG C Library.

5. Error Indications and Warnings

NE_BAD_PARAM

    On entry parameter **task** had an illegal value.

    On entry parameter **errass** had an illegal value.

    On entry parameter **method** had an illegal value.

NE_INT_ARG_LT

On entry, **neq** must not be less than 1: **neq** = ⟨*value*⟩.

NE_2_REAL_ARG_EQ

On entry **tstart** = ⟨*value*⟩ while **tend** = ⟨*value*⟩. These parameters must satisfy **tstart** ≠ **tend**

NE_REAL_RANGE_CONS

On entry **tol** = ⟨*value*⟩ and 10***machine precision** = ⟨*value*⟩. The parameter **tol** must satisfy 10***machine precision** ≤ **tol** ≤ 0.01

NE_REAL_ARRAY_INPUT

On entry **thres**[⟨*value*⟩] = ⟨*value*⟩
Constraint: **thres**[⟨*value*⟩] ≥ ⟨*value*⟩

NE_2_REAL_ARG_TOO_CLOSE

On entry **tend** = ⟨*value*⟩ while **tstart** = ⟨*value*⟩. These parameters must satisfy $abs(\textbf{tend}-\textbf{tstart}) \geq$ ⟨*value*⟩.

NE_ALLOC_FAIL

Memory allocation failed.

6. Further Comments

The value of the parameter **tend** may be reset during the integration without the overhead associated with a complete restart; this can be achieved by a call to nag_ode_ivp_rk_reset_tend (d02pwc).

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 **thres**[$i-1$] to this threshold. In this way you avoid the cost of computing significant digits in y_i when only the fact that it is smaller than the threshold is of interest. This matter is important when y_i vanishes, and in particular, when the initial value **ystart**[$i-1$] vanishes. An appropriate threshold depends on the general size of y_i in the course of the integration. Physical reasoning may help you select suitable threshold values. If you do not know what to expect of y, you can find out by a preliminary integration using nag_ode_ivp_rk_range (d02pcc) with nominal values of **thres**. As nag_ode_ivp_rk_range (d02pcc) steps from t_0 towards t_f for each $i = 1, 2, \ldots,$**neq** it forms **ymax**[$i-1$], the largest magnitude of y_i computed at any step in the integration so far. Using this you can determine more appropriate values for **thres** for an accurate integration. You might, for example, take **thres**[$i-1$] to be 10.0 × **machine precision** times the final value of **ymax**[$i-1$].

6.1. Accuracy

Not applicable.

6.2. References

Brankin R W, Gladwell I and Shampine L F (1991) *RKSUITE: a suite of Runge-Kutta codes for the initial value problem for ODEs* SoftReport 91-S1, Department of Mathematics, Southern Methodist University, Dallas, TX 75275, U.S.A.

7. See Also

nag_ode_ivp_rk_range (d02pcc)
nag_ode_ivp_rk_onestep (d02pdc)
nag_ode_ivp_rk_free (d02ppc)
nag_ode_ivp_rk_reset_tend (d02pwc)
nag_ode_ivp_rk_interp (d02pxc)
nag_ode_ivp_rk_errass (d02pzc)

8. Example

See example programs for nag_ode_ivp_rk_range (d02pcc), nag_ode_ivp_rk_onestep (d02pdc), nag_ode_ivp_rk_interp (d02pxc), nag_ode_ivp_rk_reset_tend (d02pwc) and nag_ode_ivp_rk_errass (d02pzc).