NAG Toolbox for MATLAB

d02pv

1 Purpose

d02pv is a setup function which must be called prior to the first call of either of the integration functions d02pc and d02pd.

2 Syntax

```
[work, ifail] = d02pv(tstart, ystart, tend, tol, thres, method, task,
errass, lenwrk, 'neq', neq, 'hstart', hstart)
```

3 Description

d02pv and its associated functions (d02pc, d02pd, d02pw, d02px, d02py, d02pz) 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 (see Brankin *et al.* 1991), integrate

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

where y is the vector of n 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, for $i = 1, 2, \ldots, n$, 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**.

d02pc can be used to solve the 'usual task', namely integrating the system of differential equations to obtain answers at points you specify. d02pd 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 **tol** 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** $\times \max(\mu_i, \mathbf{thres}(i))$ where μ_i is an average magnitude of y_i over the step. If **thres**(i) is smaller than the current value of μ_i, this is a relative error test and **tol** indicates how many significant digits you want in y_i. If **thres**(i) is larger than the current value of μ_i, this is an absolute error test with tolerance **tol** $\times$ **thres**(i). Relative error control is the recommended mode of operation, but pure relative error control, **thres**$(i) = 0.0$, is not permitted. See Section 8 for further information about error control.

d02pc and d02pd 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.

d02pc and d02pd 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 and 8 respectively. As a rule, the smaller **tol** is, the larger you should take the value of **method**. If the components **thres** are small enough that you are effectively specifying relative error control, experience suggests

tol	efficient **method**
$10^{-2} - 10^{-4}$	1
$10^{-3} - 10^{-6}$	2
$10^{-5} -$	3

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 value for **thres**(i), 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 value for **method** to compute significant digits, but **method** $= 1$ is a particularly poor choice in this situation. All three methods are inefficient when the problem is 'stiff'. If it is only mildly stiff, you can solve it with acceptable efficiency with **method** $= 1$, but if it is moderately or very stiff, a code designed specifically for such problems will be much more efficient. The higher the order, i.e., the larger the value of **method**, the more smoothness is required of the solution in order 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 d02pz. 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** $= 2$ or 3, and three times with **method** $= 1$.

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 parameter **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 via the parameter **hstart**.

4 References

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

5 Parameters

5.1 Compulsory Input Parameters

1: **tstart – double scalar**

The initial value of the independent variable, t_0.

2: **ystart(neq) – double array**

y_0, the initial values of the solution, y_i, for $i = 1, 2, \ldots n$, at t_0.

3: **tend – double scalar**

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.

4: **tol – double scalar**

A relative error tolerance.

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

5: **thres(neq) – double array**

A vector of thresholds.

Constraint: **thres**$(i) \geq \sqrt{\sigma}$, where σ is approximately the smallest possible machine number that can be reciprocated without overflow (see x02am).

6: **method – int32 scalar**

The Runge–Kutta method to be used.

method $= 1$

A 2(3) pair is used.

method $= 2$

A 4(5) pair is used.

method $= 3$

A 7(8) pair is used.

Constraint: $1 \leq$ **method** ≤ 3.

7: **task – string**

Determines whether the usual integration task is to be performed using d02pc or a more complicated task is to be performed using d02pd.

task = 'U'

d02pc is to be used for the integration.

task = 'C'

d02pd is to be used for the integration.

Constraint: **task** = 'U' or 'C'.

8: **errass – logical scalar**

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

9: **lenwrk – int32 scalar**

(**lenwrk** $\geq 32 \times$ **neq** is always sufficient.)

Constraints:

if **task** = 'U' and **errass** = **false**,
if **method** = 1, **lenwrk** $\geq 10 \times$ **neq**;
if **method** = 2, **lenwrk** $\geq 20 \times$ **neq**;
if **method** = 3, **lenwrk** $\geq 16 \times$ **neq**;

if **task** = 'U' and **errass** = **true**,
if **method** = 1, **lenwrk** $\geq 17 \times$ **neq**;
if **method** = 2, **lenwrk** $\geq 32 \times$ **neq**;
if **method** = 3, **lenwrk** $\geq 21 \times$ **neq**;

if **task** = 'C' and **errass** = **false**,
if **method** = 1, **lenwrk** $\geq 10 \times$ **neq**;
if **method** = 2, **lenwrk** $\geq 14 \times$ **neq**;
if **method** = 3, **lenwrk** $\geq 16 \times$ **neq**;

if **task** = 'C' and **errass** = **true**,
if **method** = 1, **lenwrk** $\geq 15 \times$ **neq**;
if **method** = 2, **lenwrk** $\geq 26 \times$ **neq**;
if **method** = 3, **lenwrk** $\geq 21 \times$ **neq**.

5.2 Optional Input Parameters

1: **neq – int32 scalar**

Default: The dimension of the arrays **ystart**, **thres**. (An error is raised if these dimensions are not equal.)

n, the number of ordinary differential equations in the system to be solved by the integration function.

Constraint: **neq** ≥ 1.

2: **hstart – double scalar**

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.

Default: 0.0

5.3 Input Parameters Omitted from the MATLAB Interface

None.

5.4 Output Parameters

1: **work**(**lenwrk**) **– double array**

Contains information for use by d02pc or d02pd. This **must** be the same array as supplied to d02pc or d02pd. The contents of this array must remain unchanged between calls.

2: **ifail – int32 scalar**

0 unless the function detects an error (see Section 6).

6 Error Indicators and Warnings

Errors or warnings detected by the function:

ifail $= 1$

On entry, **neq** < 1,
or **tend** is too close to **tstart**,
or **tol** > 0.01 or **tol** $< 10.0 \times$ ***machine precision***,
or **thres**$(i) < \sqrt{\sigma}$, where σ is approximately the smallest possible machine number that can be reciprocated without overflow (see x02am),
or **method** $\neq 1$, 2 or 3,
or **task** $\neq$ 'U' or 'C',
or **lenwrk** is too small.

7 Accuracy

Not applicable.

8 Further Comments

If **task** $=$ 'C' then 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 d02pw.

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) 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) 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 d02pc with nominal values of **thres**. As d02pc steps from t_0 towards t_f for each $i = 1, 2, \ldots, n$ it forms **ymax**(i), 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) to be $10.0 \times$ ***machine precision*** times the final value of **ymax**(i).

9 Example

d02pc_f.m

```
function [yp] = f(t, y)
  yp = zeros(2, 1);
  yp(1) = y(2);
  yp(2) = -y(1);
```

```
tstart = 0;
ystart = [0;
     1];
tend = 6.283185307179586;
tol = 0.001;
thres = [1e-08;
     1e-08];
method = int32(1);
task = 'Usual Task';
errass = false;
lenwrk = int32(64);
neq = int32(2);
twant = 0.7853981633974483;
ygot = [0; 0];
```

```
ymax = [0; 0];
[work, ifail] = ...
       d02pv(tstart, ystart, tend, tol, thres, method, task, errass,
lenwrk);
[tgot, ygotOut, ypgot, ymaxOut, workOut, ifail] = ...
    d02pc('d02pc_f', neq, twant, ygot, ymax, work)
```

```
tgot =
    0.7854
ygotOut =
    0.7069
    0.7068
ypgot =
    0.7058
   -0.7084
ymaxOut =
    0.7069
    1.0000
workOut =
     array elided
ifail =
           0
```