NAG Toolbox for MATLAB

d02pd

1 Purpose

d02pd is a one-step function for solving the initial value problem for a first-order system of ordinary differential equations using Runge–Kutta methods.

2 Syntax

```
[tnow, ynow, ypnow, work, ifail] = d02pd(f, neq, work)
```

3 Description

d02pd and its associated functions (d02pv, 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.

d02pd is designed to be used in complicated tasks when solving systems of ordinary differential equations. You must first call d02pv to specify the problem and how it is to be solved. Thereafter you (repeatedly) call d02pd to take one integration step at a time from **tstart** in the direction of **tend** (as specified in d02pv). In this manner d02pd returns an approximation to the solution **ynow** and its derivative **ypnow** at successive points **tnow**. If d02pd encounters some difficulty in taking a step, the integration is not advanced and the function returns with the same values of **tnow**, **ynow** and **ypnow** as returned on the previous successful step. d02pd tries to advance the integration as far as possible subject to passing the test on the local error and not going past **tend**. In the call to d02pv you can specify either the first step size for d02pd to attempt or that it compute automatically an appropriate value. Thereafter d02pd estimates an appropriate step size for its next step. This value and other details of the integration can be obtained after any call to d02pd by a call to d02py. The local error is controlled at every step as specified in d02pv. If you wish to assess the true error, you must set **errass** $=$ **true** in the call to d02pv. This assessment can be obtained after any call to d02pd by a call to d02pz.

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

(i) The more efficient way is to step past the point where a solution is desired, and then call d02px 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 d02px. This is very valuable when you want to find where something happens, e.g., where a particular solution component vanishes. You cannot proceed in this way with **method** $= 3$.

(ii) The other way to get an answer at a specific point is to set **tend** to this value and integrate to **tend**. d02pd will not step past **tend**, so when a step would carry it past, it will reduce the step size so as to produce an answer at **tend** exactly. After getting an answer there (**tnow** $=$ **tend**), you can reset **tend** to the next point where you want an answer, and repeat. **tend** could be reset by a call to d02pv, but you should not do this. You should use d02pw instead 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** $= 3$. It can be inefficient. Should this be the case, the code will bring the matter to your attention.

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: **f – string containing name of m-file**

f must evaluate the functions f_i (that is the first derivatives y_i') for given values of the arguments t, y_i.

Its specification is:

```
[yp] = f(t, y)
```

Input Parameters

1: **t – double scalar**

t, the current value of the independent variable.

2: **y**(n) **– double array**

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

Output Parameters

1: **yp**(n) **– double array**

The values of f_i, for $i = 1, 2, \ldots, n$.

2: **neq – int32 scalar**

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

Constraint: **neq** ≥ 1.

3: **work**($*$) **– double array**

Note: the dimension of the array **work** must be at least **lenwrk** (see d02pv).

This **must** be the same array as supplied to d02pv. It **must** remain unchanged between calls.

5.2 Optional Input Parameters

None.

5.3 Input Parameters Omitted from the MATLAB Interface

None.

5.4 Output Parameters

1: **tnow – double scalar**

The value of the independent variable t at which a solution has been computed.

2: **ynow**($*$) **– double array**

Note: the dimension of the array **ynow** must be at least n.

An approximation to the solution at **tnow**. The local error of the step to **tnow** was no greater than permitted by the specified tolerances (see d02pv).

3: **ypnow**$(*)$ **– double array**

Note: the dimension of the array **ypnow** must be at least n.

An approximation to the derivative of the solution at **tnow**.

4: **work**$(*)$ **– double array**

Note: the dimension of the array **work** must be at least **lenwrk** (see d02pv).

Information about the integration for use on subsequent calls to d02pd or other associated functions.

5: **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, an invalid call to d02pd was made, for example without a previous call to the setup function d02pv. You cannot continue integrating the problem.

ifail = 2

d02pd is being used inefficiently because the step size has been reduced drastically many times to obtain answers at many points **tend**. If you really need the solution at this many points, you should use d02px to obtain the answers inexpensively. If you need to change from **method** = 3 to do this, restart the integration from **tnow**, **ynow** by a call to d02pv. If you wish to continue as before, call d02pd again. The monitor of this kind of inefficiency will be reset automatically so that the integration can proceed.

ifail = 3

A considerable amount of work has been expended in the (primary) integration. This is measured by counting the number of calls to the user-supplied (sub)program **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 **errass** = **true** in the call to d02pv) are not counted in this total. The integration was interrupted. If you wish to continue on towards **tend**, just call d02pd again. The counter measuring work will be reset to zero automatically.

ifail = 4

It appears that this problem is stiff. The methods implemented in d02pd can solve such problems, but they are inefficient. You should change to another code based on methods appropriate for stiff problems. The integration was interrupted. If you want to continue on towards **tend**, just call d02pd again. The stiffness monitor will be reset automatically.

ifail = 5

It does not appear possible to achieve the accuracy specified by **tol** and **thres** in the call to d02pv with the precision available on the computer being used and with this value of **method**. You cannot continue integrating this problem. A larger value for **method**, if possible, will permit greater accuracy with this precision. To increase **method** and/or continue with larger values of **tol** and/or **thres**, restart the integration from **tnow**, **ynow** by a call to d02pv.

ifail = 6

(This error exit can only occur if **errass** = **true** in the call to d02pv.) The global error assessment may not be reliable beyond the current integration point **tnow**. This may occur because either too little or too much accuracy has been requested or because $f(t,y)$ is not smooth enough for values of t just beyond **tnow** and current values of the solution y. The integration cannot be continued. This return does not mean that you cannot integrate past **tnow**, rather that you cannot do it with **errass** = **true**. However, it may also indicate problems with the primary integration.

7 Accuracy

The accuracy of integration is determined by the parameters **tol** and **thres** in a prior call to d02pv. Note that only the local error at each step is controlled by these parameters. 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.

8 Further Comments

If d02pd returns with **ifail** = 5 and the accuracy specified by **tol** and **thres** 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 large in magnitude. Successive output values of **ynow** should be monitored with the aim of trapping the solution before the singularity. In any case numerical integration cannot be continued through a singularity, and analytical treatment may be necessary.

Performance statistics are available after any return from d02pd (except when **ifail** = 1) by a call to d02py. If **errass** = **true** in the call to d02pv, global error assessment is available after any return from d02pd (except when **ifail** = 1) by a call to d02pz.

After a failure with **ifail** = 5 or 6 the diagnostic functions d02py and d02pz may be called only once.

If d02pd returns with **ifail** = 4 then it is advisable to change to another code more suited to the solution of stiff problems. d02pd will not return with **ifail** = 4 if the problem is actually stiff but it is estimated that integration can be completed using less function evaluations than already computed.

9 Example

```
d02pd_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 = 0.0001;
thres = [1e-08; 1e-08];
method = int32(2);
task = 'Complex Task';
errass = false;
lenwrk = int32(64);
neq = int32(2);
[work, ifail] = ...
       d02pv(tstart, ystart, tend, tol, thres, method, task, errass,
lenwrk);
[tnow, ynow, ypnow, workOut, ifail] = d02pd('d02pd_f', neq, work)
```

```
tnow =
    0.7854
ynow =
    0.7071
    0.7071
ypnow =
    0.7071
   -0.7071
```

```
workOut =
     array elided
ifail =
           0
```