NAG Toolbox

nag_pde_1d_parab_fd_interp (d03pz)

1 Purpose

nag_pde_1d_parab_fd_interp (d03pz) interpolates in the spatial coordinate the solution and derivative of a system of partial differential equations (PDEs). The solution must first be computed using one of the finite difference schemes nag_pde_1d_parab_fd (d03pc), nag_pde_1d_parab_dae_fd (d03ph) or nag_pde_1d_parab_remesh_fd (d03pp), or one of the Keller box schemes nag_pde_1d_parab_keller (d03pe), nag_pde_1d_parab_dae_keller (d03pk) or nag_pde_1d_parab_remesh_keller (d03pr).

2 Syntax

```
[up, ifail] = nag_pde_1d_parab_fd_interp(m, u, x, xp, itype, 'npde', npde, 'npts', npts, 'intpts', intpts)
[up, ifail] = d03pz(m, u, x, xp, itype, 'npde', npde, 'npts', npts, 'intpts', intpts)
```

3 Description

nag_pde_1d_parab_fd_interp (d03pz) is an interpolation function for evaluating the solution of a system of partial differential equations (PDEs), at a set of user-specified points. The solution of the system of equations (possibly with coupled ordinary differential equations) must be computed using a finite difference scheme or a Keller box scheme on a set of mesh points. nag_pde_1d_parab_fd_interp (d03pz) can then be employed to compute the solution at a set of points anywhere in the range of the mesh. It can also evaluate the first spatial derivative of the solution. It uses linear interpolation for approximating the solution.

4 References

None.

5 Parameters

Note: the arguments **x**, **m**, **u**, **npts** and **npde** must be supplied unchanged from the PDE function.

5.1 Compulsory Input Parameters

1: **m** – INTEGER

The coordinate system used. If the call to nag_pde_1d_parab_fd_interp (d03pz) follows one of the finite difference functions then **m** must be the same argument **m** as used in that call. For the Keller box scheme only Cartesian coordinate systems are valid and so **m must** be set to zero. No check will be made by nag_pde_1d_parab_fd_interp (d03pz) in this case.

m = 0
Indicates Cartesian coordinates.

m = 1
Indicates cylindrical polar coordinates.

m = 2
Indicates spherical polar coordinates.

Constraints:

$0 \leq \mathbf{m} \leq 2$ following a finite difference function;
$\mathbf{m} = 0$ following a Keller box scheme function.

2: **u**(**npde**, **npts**) – REAL (KIND=nag_wp) array

The PDE part of the original solution returned in the argument **u** by the PDE function.

Constraint: **npde** ≥ 1.

3: **x**(**npts**) – REAL (KIND=nag_wp) array

$\mathbf{x}(i)$, for $i = 1, 2, \ldots,$ **npts**, must contain the mesh points as used by the PDE function.

4: **xp**(**intpts**) – REAL (KIND=nag_wp) array

$\mathbf{xp}(i)$, for $i = 1, 2, \ldots,$ **intpts**, must contain the spatial interpolation points.

Constraint: $\mathbf{x}(1) \leq \mathbf{xp}(1) < \mathbf{xp}(2) < \cdots < \mathbf{xp}(\mathbf{intpts}) \leq \mathbf{x}(\mathbf{npts})$.

5: **itype** – INTEGER

Specifies the interpolation to be performed.

itype $= 1$
The solutions at the interpolation points are computed.

itype $= 2$
Both the solutions and their first derivatives at the interpolation points are computed.

Constraint: **itype** $= 1$ or 2.

5.2 Optional Input Parameters

1: **npde** – INTEGER

Default: the first dimension of the array **u**.

The number of PDEs.

Constraint: **npde** ≥ 1.

2: **npts** – INTEGER

Default: the dimension of the array **x** and the second dimension of the array **u**. (An error is raised if these dimensions are not equal.)

The number of mesh points.

Constraint: **npts** ≥ 3.

3: **intpts** – INTEGER

Default: the dimension of the array **xp**.

The number of interpolation points.

Constraint: **intpts** ≥ 1.

5.3 Output Parameters

1: **up**(**npde**, **intpts**, **itype**) – REAL (KIND=nag_wp) array

If **itype** $= 1$, $\mathbf{up}(i, j, 1)$, contains the value of the solution $U_i(x_j, t_{\text{out}})$, at the interpolation points $x_j = \mathbf{xp}(j)$, for $j = 1, 2, \ldots,$ **intpts** and $i = 1, 2, \ldots,$ **npde**.

If **itype** $= 2$, $\mathbf{up}(i, j, 1)$ contains $U_i(x_j, t_{\text{out}})$ and $\mathbf{up}(i, j, 2)$ contains $\dfrac{\partial U_i}{\partial x}$ at these points.

2: **ifail** – INTEGER

ifail $= 0$ unless the function detects an error (see Section 5).

6 Error Indicators and Warnings

Errors or warnings detected by the function:

ifail $= 1$

On entry, **itype** $\neq 1$ or 2,
or **intpts** < 1,
or **npde** < 1,
or **npts** < 3,
or **m** $\neq 0$, 1 or 2,
or the mesh points $\mathbf{x}(i)$, for $i = 1, 2, \ldots,$ **npts**, are not in strictly increasing order.

ifail $= 2$

On entry, the interpolation points $\mathbf{xp}(i)$, for $i = 1, 2, \ldots,$ **intpts**, are not in strictly increasing order.

ifail $= 3$

You are attempting extrapolation, that is, one of the interpolation points $\mathbf{xp}(i)$, for some i, lies outside the interval $[\mathbf{x}(1), \mathbf{x}(\mathbf{npts})]$. Extrapolation is not permitted.

ifail $= -99$

An unexpected error has been triggered by this routine. Please contact NAG.

ifail $= -399$

Your licence key may have expired or may not have been installed correctly.

ifail $= -999$

Dynamic memory allocation failed.

7 Accuracy

See the PDE function documents.

8 Further Comments

None.

9 Example

See Section 10 in nag_pde_1d_parab_fd (d03pc), nag_pde_1d_parab_remesh_fd (d03pp) and nag_pde_1d_parab_remesh_keller (d03pr).

9.1 Program Text

```
function d03pz_example

fprintf('d03pz example results\n\n');

%  Solution of an elliptic-parabolic pair of PDEs.
global alpha;

% Set values for problem parameters.
npde = 2;
```

```
% Number of points on calculation mesh, and on interpolated mesh.
npts = 20;
intpts = 6;

itype  = nag_int(1);
neqn   = npde*npts;
lisave = neqn + 24;
nwk    = (10 + 6*npde)*neqn;
lrsave = nwk + (21 + 3*npde)*npde + 7*npts + 54;

% Define some arrays.
rsave  = zeros(lrsave, 1);
u      = zeros(npde, npts);
uinterp = zeros(npde, intpts, itype);
x      = zeros(npts, 1);
isave = zeros(lisave, 1, nag_int_name);
cwsav = {'', '', '', '', '', '', '', '', '', ''};
lwsav = false(100, 1);
iwsav = zeros(505, 1, nag_int_name);
rwsav = zeros(1100, 1);

% Set up the points on the interpolation grid.
xinterp = [0.0 0.4 0.6 0.8 0.9 1.0];

acc = 1.0e-3;
alpha = 1;

itrace = nag_int(0);
itask  = nag_int(1);
% Use cylindrical polar coordinates.
m = nag_int(1);

% We run through the calculation twice; once to output the interpolated
% results, and once to store the results for plotting.
niter = [5, 28];

% Prepare to store plotting results.
tsav = zeros(niter(2), 1);
usav = zeros(2, niter(2), npts);
isav = 0;

for icalc = 1:2

  % Set spatial mesh points.
  piby2 = pi/2;
  hx = piby2/(npts-1);
  x = sin(0:hx:piby2);

  % Set initial conditions.
  ts   = 0;
  tout = 1e-5;

  % Set the initial values.
  [u] = uinit(x, npts);

  % Start the integration in time.
  ind = nag_int(0);

  % Counter for saved results.
  isav = 0;

  % Loop over endpoints for the integration.
  % Set itask = 1: normal computation of output values at t = tout.
  for iter = 1:niter(icalc)

    %Set the endpoint.
    if icalc == 1
      tout = 10.0*tout;
    else
      if iter < 10
```

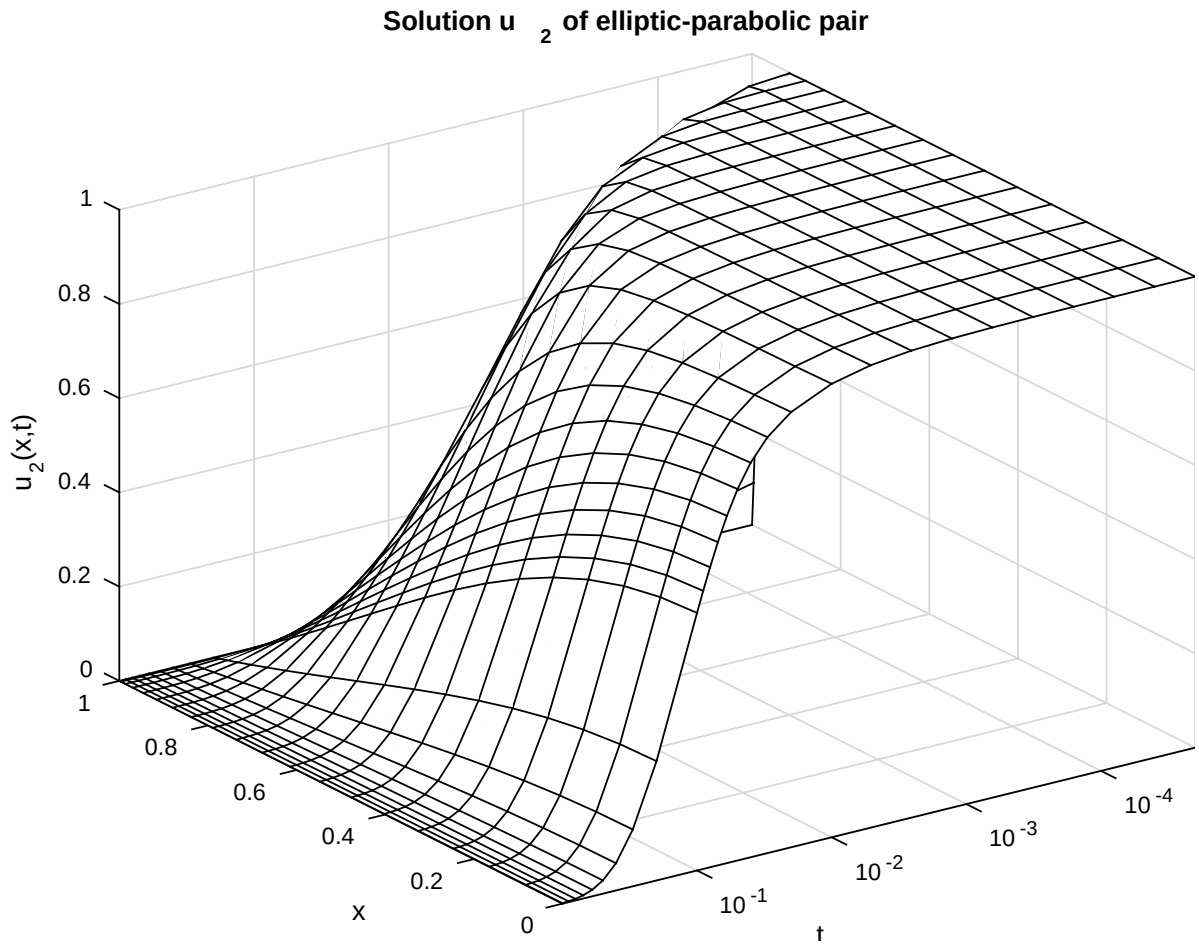
```
        tout = 2*tout;
      else
        if iter == 10
          tout = 0.01;
        else
          if iter < 20
            tout = tout + 0.01;
          else
            tout = tout + 0.1;
          end
        end
      end
    end

    % The first time this is called, ind is 0, which (re)starts the
    % integration in time.  On exit, ind is set to 1; using this value
    % on a subsequent call continues the integration.  This means that
    % only tout and ifail should be reset between calls.
    [ts, u, rsave, isave, ind, user, cwsav, lwsav, iwsav, rwsav, ifail] = ...
    d03pc(...
           m, ts, tout, @pdedef, @bndary, u, x, acc, rsave, isave, itask, ...
          itrace, ind, cwsav, lwsav, iwsav, rwsav);

    if icalc == 1
      % Output interpolation points first time through.
      if iter == 1
        fprintf([' accuracy requirement = %12.5e\n', ...
                 ' parameter alpha =  %12.3e\n'], acc, alpha);
        fprintf(' t / x       ');
        fprintf('%8.4f', xinterp(1:intpts));
        fprintf('\n\n');
      end

      % Call d03pz to do interpolation of results onto coarser grid.
      [uinterp, ifail] = d03pz( ...
                               m, u, x, xinterp, itype);

      % Output interpolated results for this time step.
      fprintf('%7.4f  u(1)', tout);
      fprintf('%8.4f', uinterp(1,1:intpts,1));
      fprintf('\n%13s','u(2)');
      fprintf('%8.4f', uinterp(2,1:intpts,1));
      fprintf('\n\n');
    else
      % Save this timestep, and this set of results.
      isav = isav+1;
      tsav(isav) = ts;
      usav(1:2,isav,1:npts) = u(1:2,1:npts);
    end
  end

  if icalc == 1
    % Output some statistics.
    fprintf([' Number of integration steps in time = %6d\n', ...
             ' Number of function evaluations      = %6d\n', ...
             ' Number of Jacobian evaluations      = %6d\n', ...
             ' Number of iterations                = %6d\n'], ...
            isave(1), isave(2), isave(3), isave(5));
  else
    % Plot results.
    fig1 = figure;
    plot_results(x, tsav, squeeze(usav(1,:,:)), 'u_1');
    fig2 = figure;
    plot_results(x, tsav, squeeze(usav(2,:,:)), 'u_2');
  end
end

function [p, q, r, ires, user] = pdedef(npde, t, x, u, ux, ires, user)
  % Evaluate Pij, Qi and Ri which define the system of PDEs.
  global alpha;
```

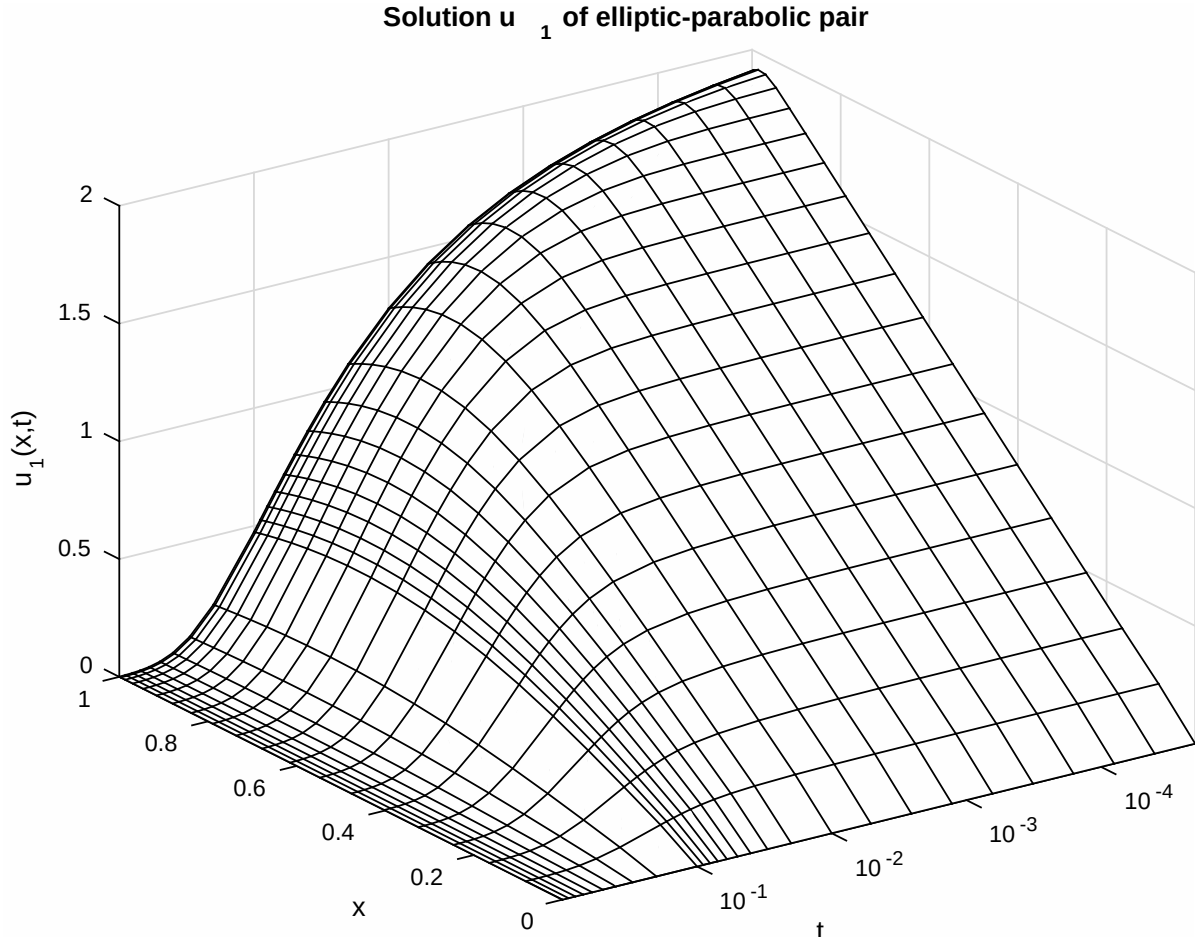
```
  p = zeros(npde,npde);
  q = zeros(npde,1);
  r = zeros(npde,1);
  q(1) = 4*alpha*(u(2) + x*ux(2));
  r(1) = x*ux(1);
  r(2) = ux(2) - u(1)*u(2);
  p(2,2) = 1 - x*x;

function [beta, gamma, ires, user] = bndary(npde, t, u, ux, ibnd, ires, user)
  % Evaluate beta and gamma to define the boundary conditions.

  if (ibnd == 0)
  beta(1) = 0;
  beta(2) = 1;
  gamma(1) = u(1);
  gamma(2) = -u(1)*u(2);
  else
  beta(1) = 1;
  beta(2) = 0;
  gamma(1) = -u(1);
  gamma(2) = u(2);
  end

function [u] = uinit(x, npts)
  % Set initial values for solution.
  global alpha;

  for i = 1:npts
    u(1,i) = 2*alpha*x(i);
    u(2,i) = 1;
  end

function plot_results(x, t, u, ident)

  % Plot array as a mesh.
  mesh(x, t, u);
  set(gca, 'YScale', 'log');
  set(gca, 'YTick', [0.00001 0.0001 0.001 0.01 0.1 1]);
  set(gca, 'YMinorGrid', 'off');
  set(gca, 'YMinorTick', 'off');

  % Label the axes, and set the title.
  xlabel('x');
  ylabel('t');
  zlabel([ident,'(x,t)']);
  title({['Solution ',ident,' of elliptic-parabolic pair'], ...
         'using method of lines and BDF'});
  title(['Solution ',ident,' of elliptic-parabolic pair']);

  % Set the axes limits tight to the x and y range.
  axis([x(1) x(end) t(1) t(end)]);

  % Set the view to something nice (determined empirically).
  view(-125, 30);
```

9.2 Program Results

```
     d03pz example results

 accuracy requirement =  1.00000e-03
 parameter alpha =     1.000e+00
 t / x         0.0000  0.4000  0.6000  0.8000  0.9000  1.0000

 0.0001  u(1)  0.0000  0.8008  1.1988  1.5990  1.7958  1.8485
         u(2)  0.9997  0.9995  0.9994  0.9988  0.9663 -0.0000

 0.0010  u(1)  0.0000  0.7982  1.1940  1.5841  1.7179  1.6734
         u(2)  0.9969  0.9952  0.9937  0.9484  0.6385 -0.0000

 0.0100  u(1)  0.0000  0.7676  1.1239  1.3547  1.3635  1.2830
```

```
         u(2)  0.9627  0.9495  0.8754  0.5537  0.2908 -0.0000

 0.1000  u(1)  0.0000  0.3908  0.5007  0.5297  0.5120  0.4744
         u(2)  0.5468  0.4299  0.2995  0.1479  0.0724 -0.0000

 1.0000  u(1)  0.0000  0.0007  0.0008  0.0008  0.0008  0.0007
         u(2)  0.0010  0.0007  0.0005  0.0002  0.0001 -0.0000

 Number of integration steps in time =     78
 Number of function evaluations      =    378
 Number of Jacobian evaluations      =     25
 Number of iterations                =    190
```




Solution u 1 of elliptic-parabolic pair
u1(x,t)
2
1.5
1
0.5
0
1
0.8
0.6
0.4
0.2
0
x
10 -1
10 -2
10 -3
10 -4
t