NAG Toolbox

nag_pde_1d_parab_euler_exact (d03px)

1 Purpose

nag_pde_1d_parab_euler_exact (d03px) calculates a numerical flux function using an Exact Riemann Solver for the Euler equations in conservative form. It is designed primarily for use with the upwind discretization schemes nag_pde_1d_parab_convdiff (d03pf), nag_pde_1d_parab_convdiff_dae (d03pl) or nag_pde_1d_parab_convdiff_remesh (d03ps), but may also be applicable to other conservative upwind schemes requiring numerical flux functions.

2 Syntax

```
[flux, ifail] = nag_pde_1d_parab_euler_exact(uleft, uright, gamma, tol, niter)
[flux, ifail] = d03px(uleft, uright, gamma, tol, niter)
```

3 Description

nag_pde_1d_parab_euler_exact (d03px) calculates a numerical flux function at a single spatial point using an Exact Riemann Solver (see Toro (1996) and Toro (1989)) for the Euler equations (for a perfect gas) in conservative form. You must supply the *left* and *right* solution values at the point where the numerical flux is required, i.e., the initial left and right states of the Riemann problem defined below. In nag_pde_1d_parab_convdiff (d03pf), nag_pde_1d_parab_convdiff_dae (d03pl) and nag_pde_1d_parab_convdiff_remesh (d03ps), the left and right solution values are derived automatically from the solution values at adjacent spatial points and supplied to the function argument **numflx** from which you may call nag_pde_1d_parab_euler_exact (d03px).

The Euler equations for a perfect gas in conservative form are:

$$\frac{\partial U}{\partial t}+\frac{\partial F}{\partial x}=0, \tag{1}$$

with

$$U=\begin{bmatrix}\rho\\ m\\ e\end{bmatrix} \quad\text{and}\quad F=\begin{bmatrix}m\\ \frac{m^2}{\rho}+(\gamma-1)\left(e-\frac{m^2}{2\rho}\right)\\ \frac{me}{\rho}+\frac{m}{\rho}(\gamma-1)\left(e-\frac{m^2}{2\rho}\right)\end{bmatrix}, \tag{2}$$

where ρ is the density, m is the momentum, e is the specific total energy and γ is the (constant) ratio of specific heats. The pressure p is given by

$$p=(\gamma-1)\left(e-\frac{\rho u^2}{2}\right), \tag{3}$$

where $u=m/\rho$ is the velocity.

The function calculates the numerical flux function $F(U_L,U_R)=F(U^*(U_L,U_R))$, where $U=U_L$ and $U=U_R$ are the left and right solution values, and $U^*(U_L,U_R)$ is the intermediate state $\omega(0)$ arising from the similarity solution $U(y,t)=\omega(y/t)$ of the Riemann problem defined by

$$\frac{\partial U}{\partial t}+\frac{\partial F}{\partial y}=0, \tag{4}$$

with U and F as in (2), and initial piecewise constant values $U=U_L$ for $y<0$ and $U=U_R$ for $y>0$. The spatial domain is $-\infty<y<\infty$, where $y=0$ is the point at which the numerical flux is required.

The algorithm is termed an Exact Riemann Solver although it does in fact calculate an approximate solution to a true Riemann problem, as opposed to an Approximate Riemann Solver which involves

some form of alternative modelling of the Riemann problem. The approximation part of the Exact Riemann Solver is a Newton–Raphson iterative procedure to calculate the pressure, and you must supply a tolerance **tol** and a maximum number of iterations **niter**. Default values for these arguments can be chosen.

A solution cannot be found by this function if there is a vacuum state in the Riemann problem (loosely characterised by zero density), or if such a state is generated by the interaction of two non-vacuum data states. In this case a Riemann solver which can handle vacuum states has to be used (see Toro (1996)).

4 References

Toro E F (1989) A weighted average flux method for hyperbolic conservation laws *Proc. Roy. Soc. Lond.* **A423** 401–418

Toro E F (1996) *Riemann Solvers and Upwind Methods for Fluid Dynamics* Springer–Verlag

5 Parameters

5.1 Compulsory Input Parameters

1: **uleft**(**3**) – REAL (KIND=nag_wp) array

uleft(i) must contain the left value of the component U_i, for $i = 1, 2, 3$. That is, **uleft**(1) must contain the left value of ρ, **uleft**(2) must contain the left value of m and **uleft**(3) must contain the left value of e.

2: **uright**(**3**) – REAL (KIND=nag_wp) array

uright(i) must contain the right value of the component U_i, for $i = 1, 2, 3$. That is, **uright**(1) must contain the right value of ρ, **uright**(2) must contain the right value of m and **uright**(3) must contain the right value of e.

3: **gamma** – REAL (KIND=nag_wp)

The ratio of specific heats, γ.

Constraint: **gamma** > 0.0.

4: **tol** – REAL (KIND=nag_wp)

The tolerance to be used in the Newton–Raphson procedure to calculate the pressure. If **tol** is set to zero then the default value of 1.0×10^{-6} is used.

Constraint: **tol** ≥ 0.0.

5: **niter** – INTEGER

The maximum number of Newton–Raphson iterations allowed. If **niter** is set to zero then the default value of 20 is used.

Constraint: **niter** ≥ 0.

5.2 Optional Input Parameters

None.

5.3 Output Parameters

1: **flux**(**3**) – REAL (KIND=nag_wp) array

flux(i) contains the numerical flux component $\hat{F}_i$, for $i = 1, 2, 3$.

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, **gamma** ≤ 0.0,
or **tol** < 0.0,
or **niter** < 0.

ifail $= 2$

On entry, the left and/or right density or derived pressure value is less than 0.0.

ifail $= 3$

A vacuum condition has been detected therefore a solution cannot be found using this function. You are advised to check your problem formulation.

ifail $= 4$

The internal Newton–Raphson iterative procedure used to solve for the pressure has failed to converge. The value of **tol** or **niter** may be too small, but if the problem persists try an Approximate Riemann Solver (nag_pde_1d_parab_euler_roe (d03pu), nag_pde_1d_parab_euler_osher (d03pv) or nag_pde_1d_parab_euler_hll (d03pw)).

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

The algorithm is exact apart from the calculation of the pressure which uses a Newton–Raphson iterative procedure, the accuracy of which is controlled by the argument **tol**. In some cases the initial guess for the Newton–Raphson procedure is exact and no further iterations are required.

8 Further Comments

nag_pde_1d_parab_euler_exact (d03px) must only be used to calculate the numerical flux for the Euler equations in exactly the form given by (2), with **uleft**(i) and **uright**(i) containing the left and right values of ρ, m and e, for $i = 1, 2, 3$, respectively.

For some problems the function may fail or be highly inefficient in comparison with an Approximate Riemann Solver (e.g., nag_pde_1d_parab_euler_roe (d03pu), nag_pde_1d_parab_euler_osher (d03pv) or nag_pde_1d_parab_euler_hll (d03pw)). Hence it is advisable to try more than one Riemann solver and to compare the performance and the results.

The time taken by the function is independent of all input arguments other than **tol**.

9 Example

This example uses nag_pde_1d_parab_convdiff_dae (d03pl) and nag_pde_1d_parab_euler_exact (d03px) to solve the Euler equations in the domain $0 \leq x \leq 1$ for $0 < t \leq 0.035$ with initial conditions for the primitive variables $\rho(x,t)$, $u(x,t)$ and $p(x,t)$ given by

$$\begin{array}{llll} \rho(x,0) = 5.99924, & u(x,0) = 19.5975, & p(x,0) = 460.894, & \text{for } x < 0.5, \\ \rho(x,0) = 5.99242, & u(x,0) = -6.19633, & p(x,0) = 46.095, & \text{for } x > 0.5. \end{array}$$

This test problem is taken from Toro (1996) and its solution represents the collision of two strong shocks travelling in opposite directions, consisting of a left facing shock (travelling slowly to the right), a right travelling contact discontinuity and a right travelling shock wave. There is an exact solution to this problem (see Toro (1996)) but the calculation is lengthy and has therefore been omitted.

9.1 Program Text

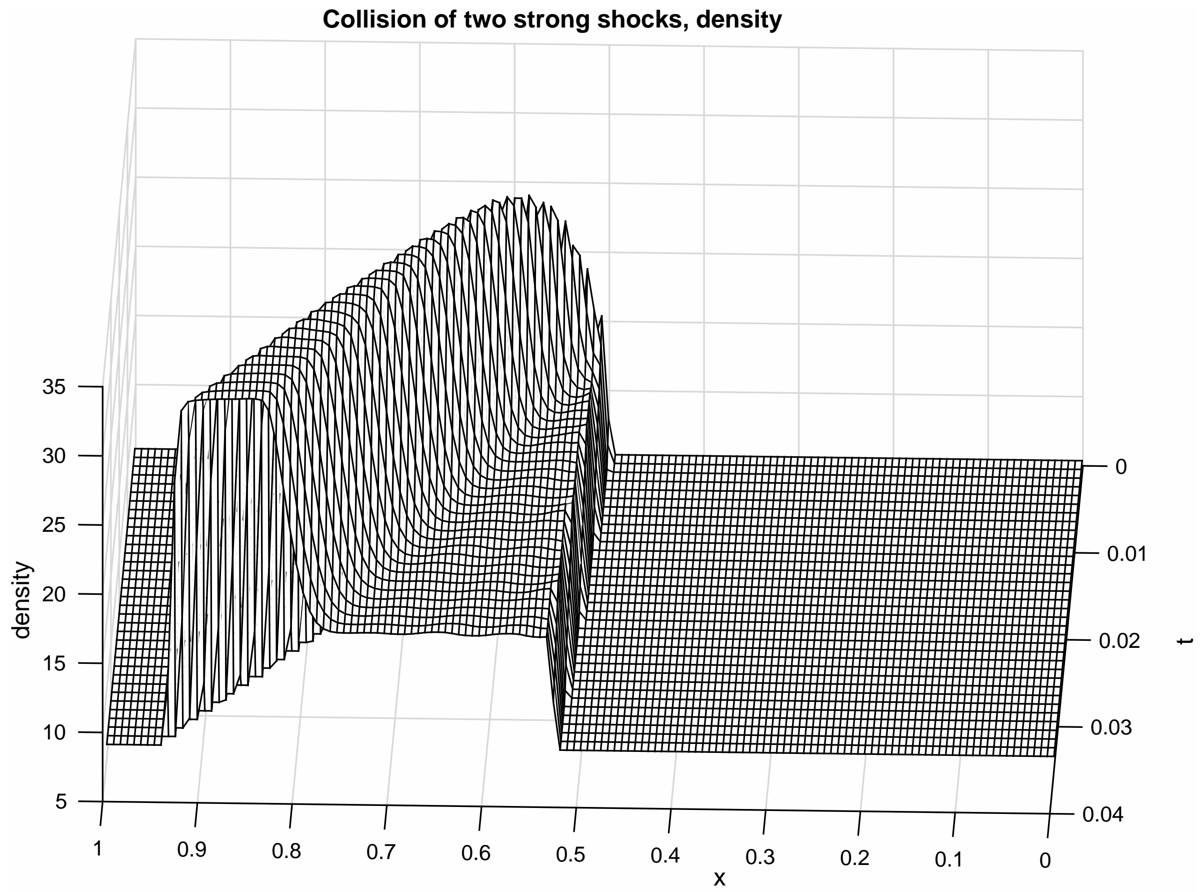
```
     function d03px_example

fprintf('d03px example results\n\n');

global gamma rl0 rr0 ul0 ur0 el0 er0;

% Problem parameters
alpha_l = 460.894;
alpha_r = 46.095;
beta_l  = 19.5975;
beta_r  =  6.19633;
gamma  =   1.4;
rl0    =   5.99924;
rr0    =   5.99242;
ul0    = 117.5701059;
ur0    = -37.1310118186;
el0    = alpha_l/(gamma-1) + rl0*beta_l^2/2;
er0    = alpha_r/(gamma-1) + rr0*beta_r^2/2;

npde  = nag_int(3);
npts  = nag_int(141);
ncode = nag_int(0);
nxi   = nag_int(0);
neqn  = npde*npts+ncode;
ts    = 0;
xi    = [];
itol = nag_int(1);
atol = [0.005];
rtol = [0.0005];
norm_p = '2';
laopt = 'B';
algopt = zeros(30,1);
algopt(1) = 2;
algopt(6) = 2;
algopt(7) = 2;
algopt(13) = 0.005;
rsave  = zeros(21000, 1);
isave  = zeros(25700, 1, nag_int_name);
itask  = nag_int(1);
itrace = nag_int(0);
ind    = nag_int(0);

% Initial mesh and solution
dx = 1/(double(npts)-1);
x  = [0:dx:1];
u = uvinit(x);

u1sol = zeros(35,npts);
u2sol = zeros(35,npts);
u3sol = zeros(35,npts);
xsol = zeros(35,npts);
tsol = zeros(35,npts);
```

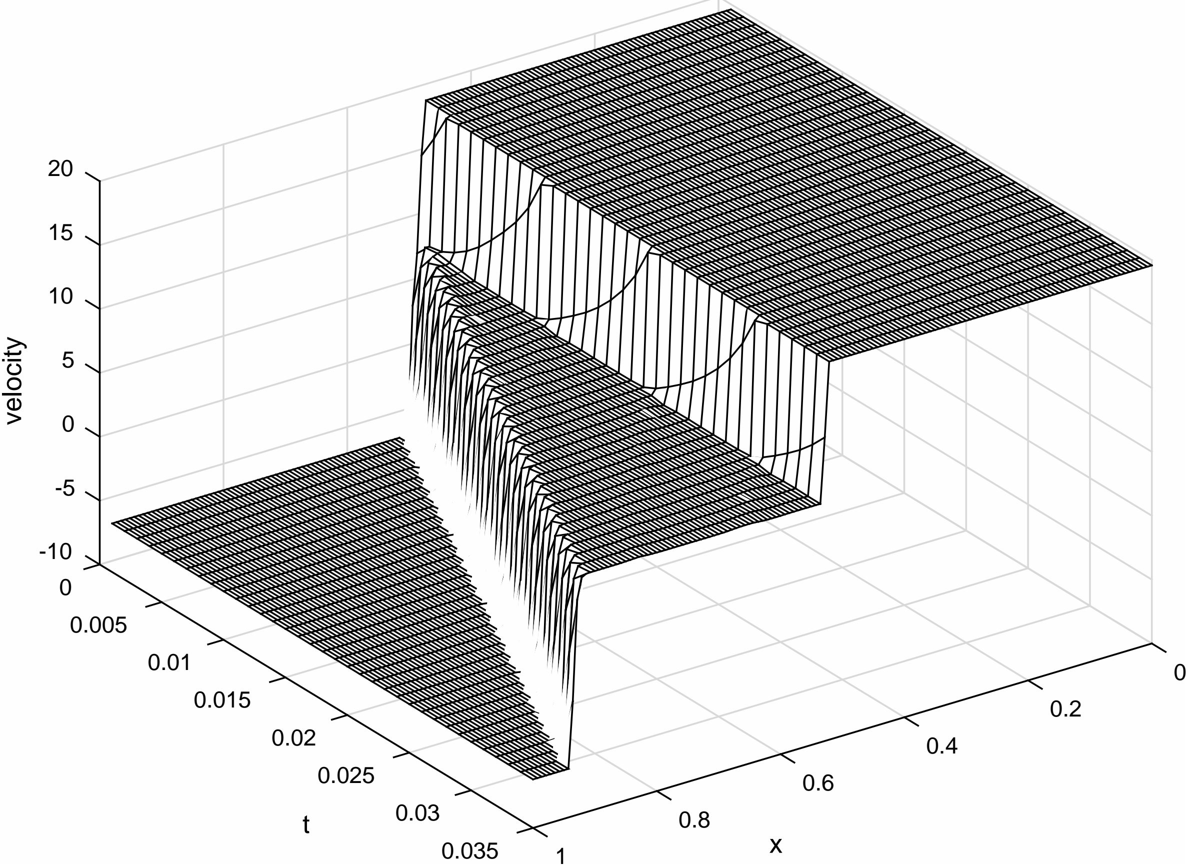
```
for j=1:35
  tout = 0.001*j;
  [ts, u, rsave, isave, ind, ifail] = ...
  d03pl( ...
         npde, ts, tout, 'd03plp', @numflx, @bndary, u, x, ncode, ...
         'd03pek', xi, rtol, atol, itol, norm_p, laopt, ...
         algopt, rsave, isave, itask, itrace, ind,'nxi',nxi);

  xsol(j,:) = x;
  tsol(j,:) = ts;
  u1sol(j,:) = u(1,:);
  u2sol(j,:) = u(2,:)./u(1,:);
  u3sol(j,:) = 0.4*u(1,:).*(u(3,:)./u(1,:)-u2sol(j,:).^2/2);

end

nsteps = 50*((isave(1)+25)/50);
nfuncs = 50*((isave(2)+25)/50);
njacs = isave(3);
niters = isave(5);
fprintf('\n Number of time steps           (nearest 50) = %6d\n',nsteps);
fprintf(' Number of function evaluations (nearest 50) = %6d\n',nfuncs);
fprintf(' Number of Jacobian evaluations (nearest  1) = %6d\n',njacs);
fprintf(' Number of iterations           (nearest  1) = %6d\n',niters);

fig1=figure;
mesh(xsol,tsol,u1sol);
title('Collision of two strong shocks, density');
xlabel('x');
ylabel('t');
zlabel('density');
view(182,40);

fig2=figure;
mesh(xsol,tsol,u2sol);
title('Collision of two strong shocks, velocity');
xlabel('x');
ylabel('t');
zlabel('velocity');
view(145,40);

fig3=figure;
mesh(xsol,tsol,u3sol);
title('Collision of two strong shocks, pressure');
xlabel('x');
ylabel('t');
zlabel('pressure');
view(-174,50);

function [g, iresout] = bndary(npde, npts, t, x, u, ncode, ...
                               v, vdot, ibnd, ires)

  global rl0 rr0 ul0 ur0 el0 er0;

  if (ibnd == 0)
    g(1) = u(1,1) - rl0;
    g(2) = u(2,1) - ul0;
    g(3) = u(3,1) - el0;
  else
    g(1) = u(1,npts) - rr0;
    g(2) = u(2,npts) - ur0;
    g(3) = u(3,npts) - er0;
  end
  iresout = ires;

function [flux, ires] = numflx(npde, t, x, ncode, v, uleft, uright, ires)

  global gamma;

  % Exact Reimann solver.
```

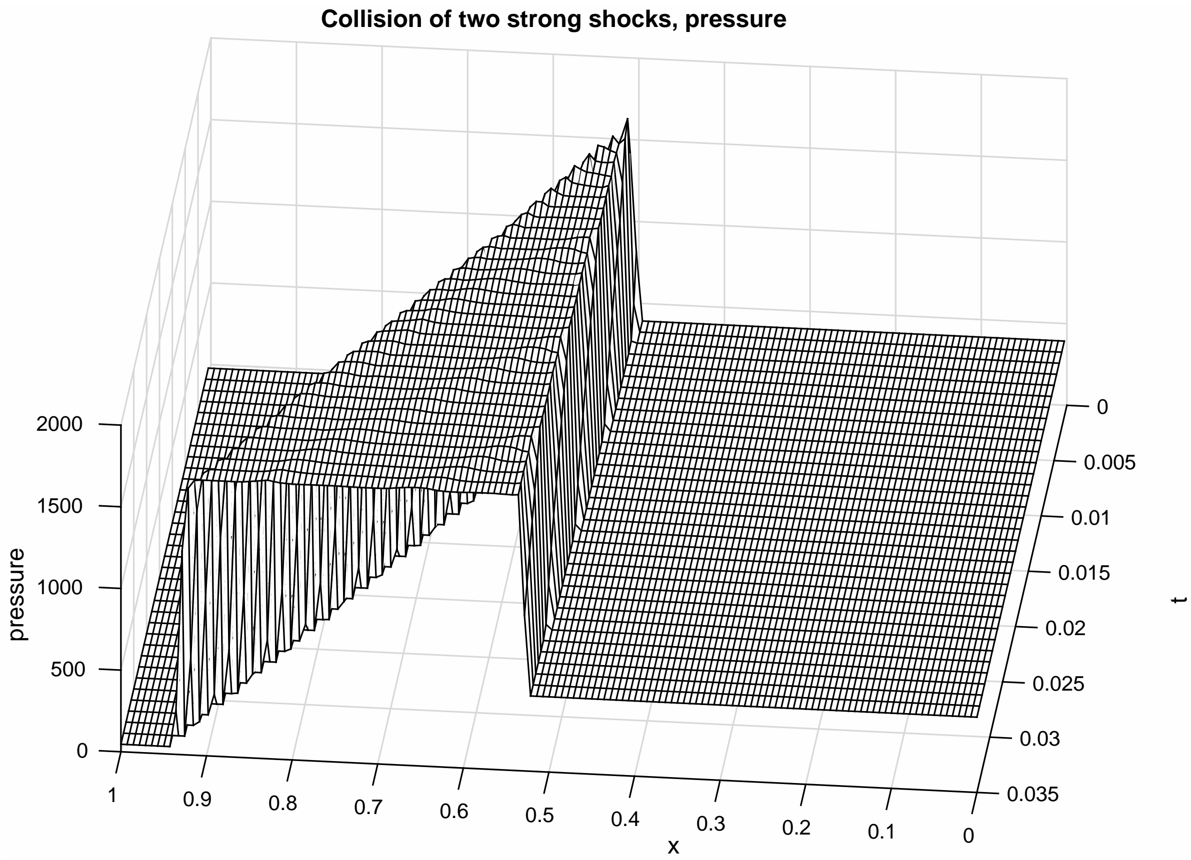
```
  tol = 0;
  niter  = nag_int(0);
  [flux, ifail] = d03px( ...
                         uleft, uright, gamma, tol, niter);

function [u] = uvinit(x)

  global rl0 rr0 ul0 ur0 el0 er0;

  n = size(x,2);
  u = zeros(3,n);
  for i = 1:n
    if x(i)<1/2
      u(1,i) = rl0;
      u(2,i) = ul0;
      u(3,i) = el0;
    elseif x(i)== 1/2
      u(1,i) = (rl0+rr0)/2;
      u(2,i) = (ul0+ur0)/2;
      u(3,i) = (el0+er0)/2;
    else
      u(1,i) = rr0;
      u(2,i) = ur0;
      u(3,i) = er0;
    end
  end
```

9.2 Program Results

```
     d03px example results

 Number of time steps           (nearest 50) =    800
 Number of function evaluations (nearest 50) =   1950
 Number of Jacobian evaluations (nearest  1) =      1
 Number of iterations           (nearest  1) =      2
```


Collision of two strong shocks, density
density
35
30
25
20
15
10
5
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
x
0
0.01
0.02
0.03
0.04
t


Collision of two strong shocks, velocity
velocity
20
15
10
5
0
-5
-10
t
0
0.005
0.01
0.015
0.02
0.025
0.03
0.035
x
1
0.8
0.6
0.4
0.2
0


Collision of two strong shocks, pressure
pressure
2000
1500
1000
500
0
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
x
t
0
0.005
0.01
0.015
0.02
0.025
0.03
0.035