NAG C Library Function Document

nag_pde_parab_1d_euler_hll (d03pwc)

1 Purpose

nag_pde_parab_1d_euler_hll (d03pwc) calculates a numerical flux function using a modified HLL (Harten-Lax-van Leer) Approximate Riemann Solver for the Euler equations in conservative form. It is designed primarily for use with the upwind discretisation schemes nag_pde_parab_1d_cd (d03pfc), nag_pde_parab_1d_cd_ode (d03plc) or nag_pde_parab_1d_cd_ode_remesh (d03psc), but may also be applicable to other conservative upwind schemes requiring numerical flux functions.

2 Specification

```c
void nag_pde_parab_1d_euler_hll (const double uleft[], const double uright[],
double gamma, double flux[], Nag_D03_Save *saved, NagError *fail)
```

3 Description

nag_pde_parab_1d_euler_hll (d03pwc) calculates a numerical flux function at a single spatial point using a modified HLL (Harten-Lax-van Leer) Approximate Riemann Solver (see Toro (1992), Toro (1996) and Toro et al. (1994)) for the Euler equations (for a perfect gas) in conservative form. The user 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_parab_1d_cd (d03pfc), nag_pde_parab_1d_cd_ode (d03plc) and nag_pde_parab_1d_cd_ode_remesh (d03psc), 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 the user may call nag_pde_parab_1d_euler_hll (d03pwc).

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

\[
\frac{\partial U}{\partial t} + \frac{\partial F}{\partial x} = 0,
\]

with

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

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

\[
p = (\gamma - 1) \left( e - \frac{m^2}{2} \right),
\]

where \( u = m/\rho \) is the velocity.

The function calculates an approximation to 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 \( \omega(y/t) \) of the Riemann problem defined by

\[
\frac{\partial U}{\partial t} + \frac{\partial F}{\partial y} = 0,
\]

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.
4 References

Toro E F (1992) The weighted average flux method applied to the Euler equations *Phil. Trans. R. Soc. Lond.* A341 499–530


5 Parameters

1:  *uleft*[3] – const double  
    *Input*
    
    *On entry*: *uleft*[i − 1] must contain the left value of the component \( U_i \) for \( i = 1, 2, 3 \). That is, *uleft*[0] must contain the left value of \( \rho \), *uleft*[1] must contain the left value of \( m \) and *uleft*[2] must contain the left value of \( e \).

2:  *uright*[3] – const double  
    *Input*
    
    *On entry*: *uright*[i − 1] must contain the right value of the component \( U_i \) for \( i = 1, 2, 3 \). That is, *uright*[0] must contain the right value of \( \rho \), *uright*[1] must contain the right value of \( m \) and *uright*[2] must contain the right value of \( e \).

3:  *gamma* – double  
    *Input*
    
    *On entry*: the ratio of specific heats \( \gamma \).  
    *Constraint*: \( \gamma > 0.0 \).

4:  *flux*[3] – double  
    *Output*
    
    *On exit*: *flux*[i − 1] contains the numerical flux component \( \hat{F}_i \) for \( i = 1, 2, 3 \).

5:  *saved* – Nag_D03_Save *  
    *Input/Output*
    
    *Note*: *saved* is a NAG defined structure. See Section 2.2.1.1 of the Essential Introduction.
    
    *On entry*: data concerning the computation required by nag_pde_parab_1d_euler_hll (d03pwc) and passed through to *numflx* from one of the integrator functions nag_pde_parab_1d_cd (d03pfc), nag_pde_parab_1d_cd_ode (d03pde), or nag_pde_parab_1d_cd_ode_remesh (d03psc).
    
    *On exit*: modified data required by the integrator function.

6:  *fail* – NagError *  
    *Input/Output*
    
    The NAG error parameter (see the Essential Introduction).

6 Error Indicators and Warnings

**NE_REAL**

- Right pressure value \( pr < 0.0 \): \( pr = \langle \text{value} \rangle \).
- Left pressure value \( pl < 0.0 \): \( pl = \langle \text{value} \rangle \).
- On entry, \( *uright*[0] < 0.0 \): \( *uright*[0] = \langle \text{value} \rangle \).
- On entry, \( *uleft*[0] < 0.0 \): \( *uleft*[0] = \langle \text{value} \rangle \).
- On entry, \( *gamma* = \langle \text{value} \rangle \).
- *Constraint*: \( *gamma* > 0.0 \).

**NE_ALLOC_FAIL**

Memory allocation failed.
NE_BAD_PARAM
On entry, parameter (value) had an illegal value.

NE_INTERNAL_ERROR
An internal error has occurred in this function. Check the function call and any array sizes. If the
call is correct then please consult NAG for assistance.

7 Accuracy
The function performs an exact calculation of the HLL numerical flux function, and so the result will be
accurate to machine precision.

8 Further Comments
The function must only be used to calculate the numerical flux for the Euler equations in exactly the form
given by (2), with \( u_{\text{left}}[i-1] \) and \( u_{\text{right}}[i-1] \) containing the left and right values of \( \rho, m \) and \( e \) for
\( i = 1, 2, 3 \) respectively. The time taken is independent of the input parameters.

9 Example
This example uses nag_pde_parab_1d_cd_ode (d03pke) and nag_pde_parab_1d_euler_hll (d03pwc) 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
\[
\begin{align*}
\rho(x, 0) &= 5.99924, & u(x, 0) &= 19.5975, & p(x, 0) &= 460.894, & \text{for } x < 0.5, \\
\rho(x, 0) &= 5.99242, & u(x, 0) &= -6.19633, & p(x, 0) &= 46.095, & \text{for } x > 0.5.
\end{align*}
\]
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 (traveling 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
/* nag_pde_parab_1d_euler_hll (d03pwc) Example Program. */
* Copyright 2001 Numerical Algorithms Group.
* * Mark 7, 2001.
*/
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <nag.h>
#include <nag_stdlib.h>
#include <nagd03.h>
#include <nagx01.h>

/* Structure to communicate with user-supplied function arguments */
struct user {
    double elo, ero, rlo, rro, ulo, uro, gamma;
};
static void bndary(Integer, Integer, double, const double[],
    const double[], Integer, const double[],
    const double[], Integer, double[], Integer *,
    Nag_Comm *);
static void numflx(Integer, double, double, Integer, const double[],
    const double[], const double[], double[], Integer *,
    double[], Integer *,
}
int main(void)
{
    const Integer npde=3, npts=141, ncode=0, nxi=0, neqn=npde*npts+ncode,
    lisave=neqn+24, intpts=9, nwkres=npde*(2*npts+3*npde+32)+7*npts+4,
    lenode=9*neqn+50, mlu=3*npde-1, lrsave=(3*mlu+1)*neqn+nwkres+lenode;
    double d, p, tout, ts, v;
    Integer exit_status, i, ind, itask, itol, itrace, k;
    double *algopt=0, *atol=0, *rtol=0, *rsave, *u=0,
    *ue=0, *x=0, *xi=0;
    Integer *isave;
    NagError fail;
    Nag_Comm comm;
    Nag_D03_Save saved;
    struct user data;

    INIT_FAIL(fail);
    exit_status = 0;

    /* Allocate memory */
    if ( !(algopt = NAG_ALLOC(30, double)) ||
        !(atol = NAG_ALLOC(1, double)) ||
        !(rtol = NAG_ALLOC(1, double)) ||
        !(u = NAG_ALLOC(npde*npts, double)) ||
        !(ue = NAG_ALLOC(npde*intpts, double)) ||
        !(rsave = NAG_ALLOC(lrsave, double)) ||
        !(x = NAG_ALLOC(npts, double)) ||
        !(xi = NAG_ALLOC(1, double)) ||
        !(isave = NAG_ALLOC(lisave, Integer)) )
    {
        Vprintf("Allocation failure\n");
        exit_status = 1;
        goto END;
    }

    Vprintf("d03pwc Example Program Results\n");

    /* Problem parameters */
    data.gamma = 1.4;
    data.rlo = 5.99924;
    data.rro = 5.99242;
    data.ulo = 5.99924*19.5975;
    data.uro = -5.99242*6.19633;
    data.elo = 460.894/(data.gamma-1.0) + 0.5*data.rlo*19.5975*19.5975;
    data.ero = 46.095 /(data.gamma-1.0) + 0.5*data.rro*6.19633*6.19633;
    comm.p = (Pointer)

    /* Initialise mesh */
    for (i = 0; i < npts; ++i) x[i] = i/(npts-1.0);

    /* Initial values */
    for (i = 1; i <= npts; ++i)
    {
        if (x[i-1] < 0.5)
        {
            U(1, i) = data.rlo;
            U(2, i) = data.ulo;
            U(3, i) = data.elo;
        } else if (x[i-1] == 0.5) {

        }
\( U(1, i) = 0.5*(\text{data.rlo} + \text{data.rro}); \)

\( U(2, i) = 0.5*(\text{data.ulo} + \text{data.uro}); \)

\( U(3, i) = 0.5*(\text{data.elo} + \text{data.ero}); \)

else {
   \( U(1, i) = \text{data.rro}; \)
   \( U(2, i) = \text{data.uro}; \)
   \( U(3, i) = \text{data.ero}; \)
}

itrace = 0;
itol = 1;
atol[0] = 0.005;
rtol[0] = 5e-4;
xi[0] = 0.0;
ind = 0;
itask = 1;

for (i = 0; i < 30; ++i) algopt[i] = 0.0;

/* Theta integration */
algopt[0] = 2.0;
algopt[5] = 2.0;
algopt[6] = 2.0;

/* Max. time step */
algopt[12] = 0.005;
ts = 0.0;
tout = 0.035;
d03plc(npde, &ts, tout, d03plp, numflx, bndary, u, npts, 
   x, ncode, d03pek, nxi, xi, neqn, rtol, atol, itol, 
   Nag_TwoNorm, Nag_LinAlgBand, algopt, rsave, lrsave, 
   lsave, itask, itrace, 0, &ind, &comm, &saved, 
   &fail);

if (fail.code != NE_NOERROR)
   { 
      Vprintf("Error from d03plc.
          \%s
", fail.message);
   exit_status = 1;
   goto END;
}

Vprintf(" t = \%6.3f\n\n", ts);
Vprintf(" x APPROX d EXACT d APPROX v EXACT v");
Vprintf(" APPROX p EXACT p\n");

/* Read exact data at output points */
for (i = 1; i <= intpts; ++i)
   { 
      Vscanf("%lf", &UE(1,i));
      Vscanf("%lf", &UE(2,i));
      Vscanf("%lf", &UE(3,i));
   }

/* Calculate density, velocity and pressure */
k = 0;
for (i = 15; i <= 127; i += 14)
   { 
      ++k;
      d = U(1, i);
      v = U(2, i)/d;
      p = d*(data.gamma-1.0)*(U(3, i)/d - 0.5*v*v);
      Vprintf(" %8.2e", x[i-1]);
      Vprintf(" %8.2e", d);
      Vprintf(" %8.2e", UE(1,k));
   }
Vprintf(" %10.4e", v);
Vprintf(" %10.4e", UE(2,k));
Vprintf(" %10.4e", p);
Vprintf(" %10.4e\n", UE(3,k));

Vprintf("\n");
Vprintf(" Number of integration steps in time = %6ld\n", isave[0]);
Vprintf(" Number of function evaluations = %6ld\n", isave[1]);
Vprintf(" Number of Jacobian evaluations =%6ld\n", isave[2]);
Vprintf(" Number of iterations = %6ld\n\n", isave[4]);

END:
if (algopt) NAG_FREE(algopt);
if (atol) NAG_FREE(atol);
if (rtol) NAG_FREE(rtol);
if (u) NAG_FREE(u);
if (ue) NAG_FREE(ue);
if (rsave) NAG_FREE(rsave);
if (x) NAG_FREE(x);
if (xi) NAG_FREE(xi);
if (isave) NAG_FREE(isave);

return exit_status;
}

static void bndary(Integer npde, Integer npts, double t, const double x[],
const double u[], Integer ncode, const double v[],
const double vdot[], Integer ibnd, double g[],
Integer *ires, Nag_Comm *comm)
{
struct user *data = (struct user *)comm->p;

if (ibnd == 0)
{
  g[0] = U(1, 1) - data->rlo;
  g[1] = U(2, 1) - data->ulo;
  g[2] = U(3, 1) - data->elo;
} else {
  g[0] = U(1, npts) - data->rro;
  g[1] = U(2, npts) - data->uro;
  g[2] = U(3, npts) - data->ero;
}
return;
}

static void numflx(Integer npde, double t, double x, Integer ncode,
const double v[], const double uleft[],
const double uright[], double flux[], Integer *ires,
Nag_Comm *comm, Nag_D03_Save *saved)
{
struct user *data = (struct user *)comm->p;
NagError fail;
INIT_FAIL(fail);
d03pwc(uleft, uright, data->gamma, flux, saved, &fail);
if (fail.code != NE_NOERROR)
{
  Vprintf("Error from d03pwc.\n\n", fail.message);
}
return;
9.2 Program Data

d03pwc Example Program Data
0.5999E+01 0.1960E+02 0.4609E+03
0.5999E+01 0.1960E+02 0.4609E+03
0.5999E+01 0.1960E+02 0.4609E+03
0.5999E+01 0.1960E+02 0.4609E+03
0.5999E+01 0.1960E+02 0.4609E+03
0.1428E+02 0.8690E+01 0.1692E+04
0.1428E+02 0.8690E+01 0.1692E+04
0.1428E+02 0.8690E+01 0.1692E+04
0.3104E+02 0.8690E+01 0.1692E+04

9.3 Program Results

d03pwc Example Program Results

\[
t = 0.035
\]

\[
x \quad \text{APPROX } d \quad \text{EXACT } d \quad \text{APPROX } v \quad \text{EXACT } v \quad \text{APPROX } p \quad \text{EXACT } p
\]
\[
1.00e-01 \quad 5.9992e+00 \quad 5.9990e+00 \quad 1.9598e+01 \quad 1.9600e+01 \quad 4.6089e+02 \quad 4.6090e+02
\]
\[
2.00e-01 \quad 5.9992e+00 \quad 5.9990e+00 \quad 1.9598e+01 \quad 1.9600e+01 \quad 4.6089e+02 \quad 4.6090e+02
\]
\[
3.00e-01 \quad 5.9992e+00 \quad 5.9990e+00 \quad 1.9598e+01 \quad 1.9600e+01 \quad 4.6089e+02 \quad 4.6090e+02
\]
\[
4.00e-01 \quad 5.9992e+00 \quad 5.9990e+00 \quad 1.9598e+01 \quad 1.9600e+01 \quad 4.6089e+02 \quad 4.6090e+02
\]
\[
5.00e-01 \quad 5.9992e+00 \quad 5.9990e+00 \quad 1.9598e+01 \quad 1.9600e+01 \quad 4.6089e+02 \quad 4.6090e+02
\]
\[
6.00e-01 \quad 1.4221e+01 \quad 1.4280e+01 \quad 8.6581e+00 \quad 8.6900e+00 \quad 1.6872e+03 \quad 1.6920e+03
\]
\[
7.00e-01 \quad 1.4255e+01 \quad 1.4280e+01 \quad 8.6697e+00 \quad 8.6900e+00 \quad 1.6881e+03 \quad 1.6920e+03
\]
\[
8.00e-01 \quad 1.9444e+01 \quad 1.4280e+01 \quad 8.6783e+00 \quad 8.6900e+00 \quad 1.6905e+03 \quad 1.6920e+03
\]
\[
9.00e-01 \quad 3.1002e+01 \quad 3.1040e+01 \quad 8.6765e+00 \quad 8.6900e+00 \quad 1.6868e+03 \quad 1.6920e+03
\]

Number of integration steps in time = 699
Number of function evaluations = 1714
Number of Jacobian evaluations = 1
Number of iterations = 2