

# NAG 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 discretization 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

```
#include <nag.h>
#include <nagd03.h>

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. 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_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 you may call `nag_pde_parab_1d_euler_hll` (d03pwc).

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

$$\frac{\partial U}{\partial t} + \frac{\partial F}{\partial x} = 0, \quad (1)$$

with

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

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

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

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  $U(y, t) = \omega(y/t)$  of the Riemann problem defined by

$$\frac{\partial U}{\partial t} + \frac{\partial F}{\partial y} = 0, \quad (4)$$

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.

## 4 References

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

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

Toro E F, Spruce M and Spears W (1994) Restoration of the contact surface in the HLL Riemann solver *J. Shock Waves* **4** 25–34

## 5 Arguments

- 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  $e$ .  
*Constraints:*  
 $\mathbf{uleft}[0] \geq 0.0$ ;  
Left pressure,  $pl \geq 0.0$ , where  $pl$  is calculated using (3).
- 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  $e$ .  
*Constraints:*  
 $\mathbf{uright}[0] \geq 0.0$ ;  
Right pressure,  $pr \geq 0.0$ , where  $pr$  is calculated using (3).
- 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 \* *Communication Structure*  
**saved** may contain data concerning the computation required by nag\_pde\_parab\_1d\_euler\_hll (d03pwc) as passed through to **numflx** from one of the integrator functions nag\_pde\_parab\_1d\_cd (d03pfc), nag\_pde\_parab\_1d\_cd\_ode (d03plc) or nag\_pde\_parab\_1d\_cd\_ode\_remesh (d03psc). You should not change the components of **saved**.
- 6: **fail** – NagError \* *Input/Output*  
The NAG error argument (see Section 2.7 in How to Use the NAG Library and its Documentation).

## 6 Error Indicators and Warnings

### NE\_ALLOC\_FAIL

Dynamic memory allocation failed.

See Section 2.3.1.2 in How to Use the NAG Library and its Documentation for further information.

### NE\_BAD\_PARAM

On entry, argument  $\langle value \rangle$  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 contact NAG for assistance.

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

See Section 2.7.6 in How to Use the NAG Library and its Documentation for further information.

### NE\_NO\_LICENCE

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

See Section 2.7.5 in How to Use the NAG Library and its Documentation for further information.

### NE\_REAL

Left pressure value  $pl < 0.0$ :  $pl = \langle value \rangle$ .

On entry, **gamma** =  $\langle value \rangle$ .

Constraint: **gamma**  $> 0.0$ .

On entry, **uleft**[0] =  $\langle value \rangle$ .

Constraint: **uleft**[0]  $\geq 0.0$ .

On entry, **uright**[0] =  $\langle value \rangle$ .

Constraint: **uright**[0]  $\geq 0.0$ .

Right pressure value  $pr < 0.0$ :  $pr = \langle value \rangle$ .

## 7 Accuracy

nag\_pde\_parab\_1d\_euler\_hll (d03pwc) performs an exact calculation of the HLL (Harten–Lax–van Leer) numerical flux function, and so the result will be accurate to *machine precision*.

## 8 Parallelism and Performance

nag\_pde\_parab\_1d\_euler\_hll (d03pwc) is not threaded in any implementation.

## 9 Further Comments

nag\_pde\_parab\_1d\_euler\_hll (d03pwc) must only be used to calculate the numerical flux for the Euler equations in exactly the form given by (2), with **uleft**[ $i - 1$ ] and **uright**[ $i - 1$ ] 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 arguments.

## 10 Example

This example uses nag\_pde\_parab\_1d\_cd\_ode (d03plc) 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

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

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.

## 10.1 Program Text

```
/* nag_pde_parab_1d_euler_hll (d03pwc) Example Program.
 *
 * NAGPRODCODE Version.
 *
 * Copyright 2016 Numerical Algorithms Group.
 *
 * Mark 26, 2016.
 */

#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;
};

#ifdef __cplusplus
extern "C"
{
#endif
    static void NAG_CALL bndary(Integer, Integer, double, const double[],
                                const double[], Integer, const double[],
                                const double[], Integer, double[], Integer *,
                                Nag_Comm *);

    static void NAG_CALL numflx(Integer, double, double, Integer,
                                const double[], const double[], const double[],
                                double[], Integer *, Nag_Comm *,
                                Nag_D03_Save *);
#ifdef __cplusplus
}
#endif

#define UE(I, J) ue[npde*((J) - 1) + (I) - 1]
#define U(I, J)  u[npde*((J) - 1) + (I) - 1]

int main(void)
{
    const Integer npde = 3, npts = 141, ncode = 0, nxi = 0;
    const Integer neqn = npde * npts + ncode, lisave = neqn + 24, intpts = 9;
    const Integer nwkres =
        npde * (2 * npts + 3 * npde + 32) + 7 * npts + 4, lenode =
        9 * neqn + 50;
    const Integer 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 *algotp = 0, *atol = 0, *rtol = 0, *rsave = 0, *u = 0, *ue = 0;
    double *x = 0, *xi = 0;
```

```

Integer *isave = 0;
NagError fail;
Nag_Comm comm;
Nag_D03_Save saved;
struct user data;

INIT_FAIL(fail);

exit_status = 0;

/* Allocate memory */

if (!(algot = 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)))
{
    printf("Allocation failure\n");
    exit_status = 1;
    goto END;
}

printf("nag_pde_parab_1d_euler_hll (d03pwc) Example Program Results\n");

/* Skip heading in data file */

#ifdef _WIN32
    scanf_s("%*[\n] ");
#else
    scanf("%*[\n] ");
#endif

/* 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) &data;

/* Initialize 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 * (data.rlo + data.rro);
        U(2, i) = 0.5 * (data.ulo + data.uro);
        U(3, i) = 0.5 * (data.elo + data.ero);
    }
    else {
        U(1, i) = data.rro;
        U(2, i) = data.uro;
        U(3, i) = 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;

/* nag_pde_parab_1d_cd_ode (d03plc).
 * General system of convection-diffusion PDEs with source
 * terms in conservative form, coupled DAEs, method of
 * lines, upwind scheme using numerical flux function based
 * on Riemann solver, one space variable
 */
nag_pde_parab_1d_cd_ode(npde, &ts, tout, NULLFN, numflx, bndary, u, npts,
                        x, ncode, NULLFN, nxi, xi, neqn, rtol, atol, itol,
                        Nag_TwoNorm, Nag_LinAlgBand, algopt, rsave, lrsave,
                        isave, lisave, itask, itrace, 0, &ind, &comm,
                        &saved, &fail);

if (fail.code != NE_NOERROR) {
    printf("Error from nag_pde_parab_1d_cd_ode (d03plc).\n%s\n",
           fail.message);
    exit_status = 1;
    goto END;
}

printf("\n t = %6.3f\n\n", ts);
printf("%15s%18s%22s\n", "d", "v", "p");
printf("%3s%10s%9s%9s%11s%11s\n", "x", "Approx", "Exact",
    "Approx", "Exact", "Approx", "Exact");

/* Read exact data at output points */

for (i = 1; i <= intpts; ++i) {
#ifdef _WIN32
    scanf_s("%lf", &UE(1, i));
#else
    scanf("%lf", &UE(1, i));
#endif
#ifdef _WIN32
    scanf_s("%lf", &UE(2, i));
#else
    scanf("%lf", &UE(2, i));
#endif
#ifdef _WIN32
    scanf_s("%lf", &UE(3, i));
#else
    scanf("%lf", &UE(3, i));
#endif
}

/* 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);
    printf("%4.1f", x[i - 1]);
    printf("%9.4f", d);
    printf("%9.4f", UE(1, k));
    printf("%9.4f", v);
    printf("%9.4f", UE(2, k));
    printf("%13.4e", p);
    printf("%13.4e\n", UE(3, k));
}

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

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

return exit_status;
}

static void NAG_CALL 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 NAG_CALL numflx(Integer npde, double t, double x, Integer ncode,
                           const double v[], const double uleft[],
                           const double uringht[], double flux[],
                           Integer *ires, Nag_Comm *comm,
                           Nag_D03_Save *saved)
{
    struct user *data = (struct user *) comm->p;
    NagError fail;

    INIT_FAIL(fail);
    /* nag_pde_parab_1d_euler_hll (d03pwc).
     * Modified HLL Riemann solver for Euler equations in
     * conservative form, for use with nag_pde_parab_1d_cd
     * (d03pfc), nag_pde_parab_1d_cd_ode (d03plc) and

```

```

    * nag_pde_parab_1d_cd_ode_remesh (d03psc)
    */
    nag_pde_parab_1d_euler_hll(uleft, uright, data->gamma, flux, saved, &fail);

    if (fail.code != NE_NOERROR) {
        printf("Error from nag_pde_parab_1d_euler_hll (d03pwc).\n%s\n",
            fail.message);
    }

    return;
}

```

## 10.2 Program Data

nag\_pde\_parab\_1d\_euler\_hll (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

```

## 10.3 Program Results

nag\_pde\_parab\_1d\_euler\_hll (d03pwc) Example Program Results

t = 0.035

	d		v		p	
x	Approx	Exact	Approx	Exact	Approx	Exact
0.1	5.9992	5.9990	19.5975	19.6000	4.6089e+02	4.6090e+02
0.2	5.9992	5.9990	19.5975	19.6000	4.6089e+02	4.6090e+02
0.3	5.9992	5.9990	19.5975	19.6000	4.6089e+02	4.6090e+02
0.4	5.9992	5.9990	19.5975	19.6000	4.6089e+02	4.6090e+02
0.5	5.9992	5.9990	19.5975	19.6000	4.6089e+02	4.6090e+02
0.6	14.2215	14.2800	8.6581	8.6900	1.6872e+03	1.6920e+03
0.7	14.2553	14.2800	8.6697	8.6900	1.6881e+03	1.6920e+03
0.8	19.4441	14.2800	8.6783	8.6900	1.6905e+03	1.6920e+03
0.9	31.0016	31.0400	8.6765	8.6900	1.6868e+03	1.6920e+03

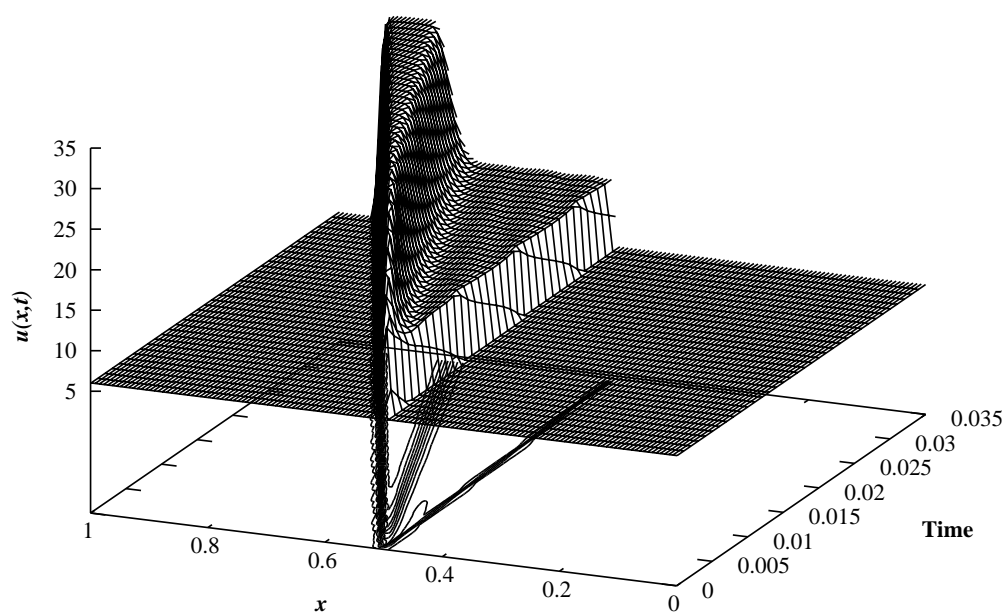
```

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

```



**Example Program**  
Euler Equation Solution Showing Collision of Two Strong Shocks  
DENSITY



Euler Equation Solution Showing Collision of Two Strong Shocks  
VELOCITY

