Godunov Methods and Riemann Solvers

Among the most interesting and difficult problems in computational fluid dynamics is the simulation of discontinuities like *shock fronts*. Simple finite difference schemes cannot handle this type of singular behavior.

Following the work of Godunov, *Mat. Sb.* **47**, 271 (1959), which was based on his Ph.D. thesis, many effective *shock-capturing schemes* were developed for applications in astrophysics and the aerospace industry.

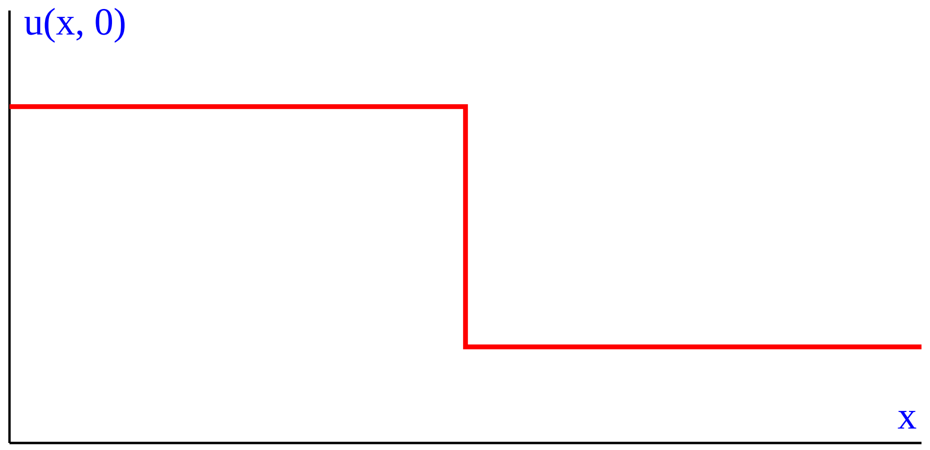
The linear advection equation

Consider the simple linear equation

$$\frac{\partial}{\partial t}u(x,t) + c\frac{\partial}{\partial x}u(x,t) = 0 ,$$

where c is a constant with dimensions of speed. Given an initial profile $u(x,0) = \xi(x)$, the solution of this equation is easily seen to be $u(x,t) = \xi(x - ct)$, i.e., a waveform which moves at constant speed $dx/dt = c$ without changing its shape.

The Riemann problem



A simple form of initial condition is a step function or piece-wise constant value for $u(x, 0)$, for example as shown in the figure. This type of initial condition defines a *Riemann problem*. Physically, this initial condition represents a *shock front* which moves with constant speed c without changing its shape.

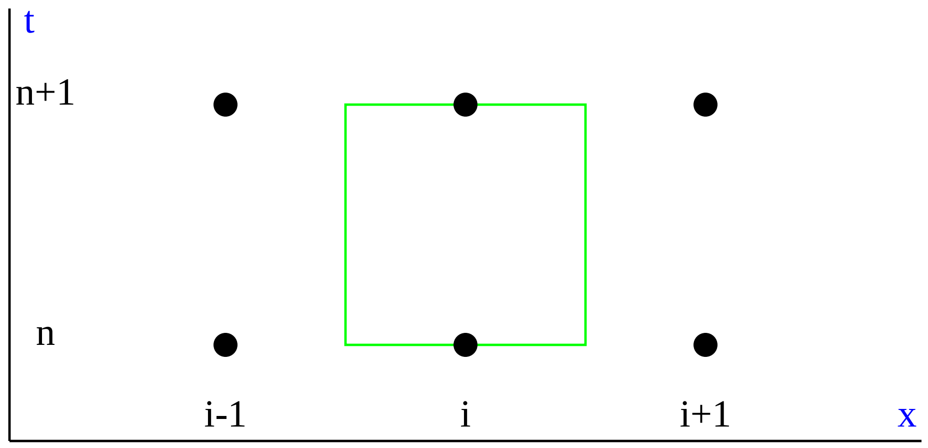
Even though this is such a simple problem with a simple solution, it is very difficult to simulate numerically. The reason for this is that the *derivative* $\partial u/\partial x$ is infinite at the discontinuity: mathematically it is a *delta function*. Most finite difference schemes assume that the solution is smooth, i.e., the derivatives are bounded, so that a Taylor series expansion in the spatial step size h is valid. When this assumption is violated by a discontinuity, a first order scheme tends to smear out the discontinuity, and including higher orders results in unstable oscillations of the solution at the position of the discontinuity.

Integral form of the conservation law

To solve this problem, Godunov used the *conservation form* of the advection equation

$$\frac{\partial}{\partial t}u(x,t) + \frac{\partial}{\partial x}f(x,t) = 0 ,$$

where $f(x, t) = cu(x, t)$ is the *flux* of the field $u(x, t)$.



The figure shows a few lattice sites on the space-time grid $x = ih$, $t = n\tau$ that will be used to solve the problem numerically. If we consider the pair (f, u) to be a vector function in the (x, t) plane, then the conservation equation

$$\partial_x f + \partial_t u = \nabla \cdot \begin{pmatrix} r \\ u \end{pmatrix}$$

is the divergence of the vector. Let us integrate this divergence over the rectangular region shown in the figure and use Gauss' integral formula to convert it to a line integral around the perimeter:

$$\int \nabla \cdot \begin{pmatrix} r \\ u \end{pmatrix} \, \mathrm{d}x \, \mathrm{d}t = \oint \begin{pmatrix} r \\ u \end{pmatrix} \cdot \hat{\mathbf{n}} \, \mathrm{d}\ell = 0 \, ,$$

where the integrand in the line integral is the normal component of the vector field on the perimeter of the rectangle. Let's define the integral averages of $u(x, t)$ on the top and bottom sides of the rectangle

$$u_i^{n+1} = \frac{1}{h} \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} u(x, t_{n+1}) \, \mathrm{d}x \qquad u_i^n = \frac{1}{h} \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} u(x, t_n) \, \mathrm{d}x \, ,$$

and the time integral averages of the flux along the left and right sides of the rectangle

$$f_{i-\frac{1}{2}} = \frac{1}{\tau} \int_{t_n}^{t_{n+1}} f(u(x_{i-\frac{1}{2}}, t)) \, \mathrm{d}t \, , \qquad f_{i+\frac{1}{2}} = \frac{1}{\tau} \int_{t_n}^{t_{n+1}} f(u(x_{i+\frac{1}{2}}, t)) \, \mathrm{d}t \, .$$

The line integral can be written

$$\left(u_i^{n+1} - u_i^n\right) \mathrm{d}x + \left(f_{i+\frac{1}{2}} - f_{i-\frac{1}{2}}\right) \mathrm{d}t = 0 \, .$$

Now, if we interpret u_i^n as the value of the solution at grid point i at time step n, then the value of the solution at grid point i at the *next* time step $n + 1$ is given by the formula

$$u_i^{n+1} = u_i^n - \frac{\tau}{h} \left(f_{i+\frac{1}{2}} - f_{i-\frac{1}{2}}\right) \, .$$

What remains is to specify the conserved *half-step fluxes* $f_{i\pm\frac{1}{2}}$.

Godunov's upwind scheme for half-step fluxes

Godunov's suggestion for determining the half-step fluxes was to solve a pair of Riemann problems. For example, to determine $f_{i+\frac{1}{2}}$, consider the Riemann problem on the interval $x_{i-\frac{1}{2}} < x < x_{i+\frac{3}{2}}$ for which $x_{i+\frac{1}{2}}$ is the center point

$$u(x, t_n) = \begin{cases} u_i^n & \text{if } x < x_{i+\frac{1}{2}} \\ u_{i+1}^n & \text{if } x > x_{i+\frac{1}{2}} \end{cases}$$

If the solution of this Riemann problem is denoted $u_{i+\frac{1}{2}}(x,t)$, then the *Godunov flux* is taken to be

$$f_{i+\frac{1}{2}} = f\left(u_{i+\frac{1}{2}}(x_{i+\frac{1}{2}}, t_n)\right) .$$

For the linear advection equation, the solution of this Riemann problem is trivial

$$u_{i+\frac{1}{2}}(x,t) = \begin{cases} u_i^n & \text{if } c > 0 \\ u_{i+1}^n & \text{if } c < 0 \end{cases}$$

and hence

$$f_{i+\frac{1}{2}} = \begin{cases} cu_i^n & \text{if } c > 0 \\ cu_{i+1}^n & \text{if } c < 0 \end{cases}$$

This gives an *upwind scheme* because if $c > 0$ the waveform moves to the right and the left initial value u_i^n covers the right boundary of the rectangular region; whereas if $c < 0$ the waveform moves to the left and the right initial value u_{i+1}^n covers the right boundary.

Substituting the Godunov flux values into the conservative update formula, we obtain the discrete solution

$$u_i^{n+1} = u_i^n - \frac{\tau}{h}\left(f_{i+\frac{1}{2}} - f_{i-\frac{1}{2}}\right) = u_i^n - \begin{cases} \lambda_{\text{CFL}}\left(u_i^n - u_{i-1}^n\right) & \text{if } c > 0 \\ \lambda_{\text{CFL}}\left(u_{i+1}^n - u_i^n\right) & \text{if } c < 0 \end{cases}$$

where the *Courant-Friedrichs-Lewy* number

$$\lambda_{\text{CFL}} = \frac{c\tau}{h} \; .$$

This general Godunov approach can be applied to more complicated problems. For example, in Burgers' equation

$$f = \frac{1}{2}u^2 \; ,$$

and the current value of the solution, rather than the constant wave speed c determines the choice of u_i^n or u_{i+1}^n in the equations above.

In the case of the 1-D Euler equations of gas dynamics, the solution and flux each have 3 components

$$\mathbf{U} = \begin{pmatrix} \rho \\ \rho u \\ e \end{pmatrix} \; , \qquad \mathbf{F} = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ u(e+p) \end{pmatrix} \; .$$

Instead of a single wave speed, the solution to the Riemann problem involves finding the eigenvalues of a 3×3 matrix: the solution involves several regions separated by left- and right-moving shock fronts and a contact discontinuity, instead of just a single shock front; the correct region must be chosen for to compute the Godunov flux.

Program to solve Sod's shock tube problem

The program `shocktube.cpp` simulates Sod's shock tube problem using various schemes: 1. Roe's Riemann solver, 2. a two-step Lax-Wendroff scheme, 3. a first order upwind Godunov scheme, and 4. a simple first order Lax-Friedrichs scheme.

```
// Program to solve Sod's shock tube problem

#include <cmath>
#include <cstdlib>
```

```
#include <iostream>
#include <cstdio>
#include <cstring>

using namespace std;

#include <GL/glut.h>
#include "RoeSolver.h"          // Roe's Riemann solver for Euler equations
#include "Riemann.h"            // Laney's upwind Godunov Riemann solver

double L = 1;                   // length of shock tube
double gama = 1.4;              // ratio of specific heats
int N = 200;                    // number of grid points

double **U;                     // solution with 3 components
double **newU;                  // new solution
double **F;                     // flux with 3 components
double *vol;                    // for Roe solver

double h;                       // lattice spacing
double tau;                     // time step
double CFL = 0.9;               // Courant-Friedrichs-Lewy number
int step;

void allocate() {
    static int oldN = 0;
    if (N != oldN) {
        if (U != 0) {
            for (int j = 0; j < oldN; j++) {
                delete [] U[j]; delete newU[j]; delete [] F[j];
            }
            delete [] U; delete [] newU; delete [] F; delete [] vol;
        }
        oldN = N;
        U = new double* [N];
        newU = new double* [N];
        F = new double* [N];
        vol = new double [N];
        for (int j = 0; j < N; j++) {
            U[j] = new double [3];
            newU[j] = new double [3];
            F[j] = new double [3];
        }
    }
}

double cMax() {
    double uMax = 0;
    for (int i = 0; i < N; i++) {
        if (U[i][0] == 0)
            continue;
        double rho = U[i][0];
        double u = U[i][1] / rho;
        double p = (U[i][2] - rho * u * u / 2) * (gama - 1);
        double c = sqrt(gama * abs(p) / rho);
        if (uMax < c + abs(u))
            uMax = c + abs(u);
    }
    return uMax;
}

void initialize() {

    allocate();
    h = L / (N - 1);
    for (int j = 0; j < N; j++) {
        double rho = 1, p = 1, u = 0;
        if (j > N / 2)
            rho = 0.125, p = 0.1;
        double e = p / (gama - 1) + rho * u * u / 2;
        U[j][0] = rho;
        U[j][1] = rho * u;
        U[j][2] = e;
        vol[j] = 1;
    }
    tau = CFL * h / cMax();
    step = 0;
}

void boundaryConditions(double **U) {

    // reflection boundary conditions at the tube ends
    U[0][0] = U[1][0];
    U[0][1] = -U[1][1];
    U[0][2] = U[1][2];
    U[N - 1][0] = U[N - 2][0];
    U[N - 1][1] = -U[N - 2][1];
    U[N - 1][2] = U[N - 2][2];
}

void LaxWendroffStep() {

    // compute flux F from U
    for (int j = 0; j < N; j++) {
        double rho = U[j][0];
        double m = U[j][1];
        double e = U[j][2];
        double p = (gama - 1) * (e - m * m / rho / 2);
        F[j][0] = m;
        F[j][1] = m * m / rho + p;
        F[j][2] = m / rho * (e + p);
    }

    // half step
    for (int j = 1; j < N - 1; j++)
        for (int i = 0; i < 3; i++)
            newU[j][i] = (U[j + 1][i] + U[j][i]) / 2 -
                         tau / 2 / h * (F[j + 1][i] - F[j][i]);
    boundaryConditions(newU);

    // compute flux at half steps
    for (int j = 0; j < N; j++) {
        double rho = newU[j][0];
        double m = newU[j][1];
        double e = newU[j][2];
        double p = (gama - 1) * (e - m * m / rho / 2);
        F[j][0] = m;
        F[j][1] = m * m / rho + p;
        F[j][2] = m / rho * (e + p);
    }

    // step using half step flux
    for (int j = 1; j < N - 1; j++)
        for (int i = 0; i < 3; i++)
            newU[j][i] = U[j][i] - tau / h * (F[j][i] - F[j - 1][i]);

    // update U from newU
    for (int j = 1; j < N - 1; j++)
        for (int i = 0; i < 3; i++)
            U[j][i] = newU[j][i];
}

void LaxFriedrichsStep() {

    // compute flux F from U
    for (int j = 0; j < N; j++) {
        double rho = U[j][0];
        double m = U[j][1];
        double e = U[j][2];
        double p = (gama - 1) * (e - m * m / rho / 2);
        F[j][0] = m;
        F[j][1] = m * m / rho + p;
        F[j][2] = m / rho * (e + p);
    }
    // Lax-Friedrichs step
    for (int j = 1; j < N - 1; j++)
        for (int i = 0; i < 3; i++)
            newU[j][i] = (U[j + 1][i] + U[j - 1][i]) / 2 -
                         tau / h * (F[j + 1][i] - F[j - 1][i]);
    boundaryConditions(newU);

    // update U from newU
    for (int j = 1; j < N - 1; j++)
        for (int i = 0; i < 3; i++)
            U[j][i] = newU[j][i];
}

void upwindGodunovStep() {

    // find fluxes using Riemann solver
    for (int j = 0; j < N - 1; j++)
        Riemann(U[j], U[j + 1], F[j]);

    // update U
    for (int j = 1; j < N - 1; j++)
        for (int i = 0; i < 3; i++)
            U[j][i] -= tau / h * (F[j][i] - F[j - 1][i]);
}

void RoeStep() {

    // compute fluxes at cell boundaries
    int icntl;
    RoeSolve(h, tau, gama, vol, U, F, N - 2, icntl);

    // update U
    for (int j = 1; j < N - 1; j++)
        for (int i = 0; i < 3; i++)
            U[j][i] -= tau / h * (F[j + 1][i] - F[j][i]);
}

double nu = 0.0;

void LapidusViscosity() {

    // store Delta_U values in newU
    for (int j = 1; j < N; j++)
        for (int i = 0; i < 3; i++)
            newU[j][i] = U[j][i] - U[j - 1][i];

    // multiply Delta_U by |Delta_U|
    for (int j = 1; j < N; j++)
        for (int i = 0; i < 3; i++)
            newU[j][i] *= abs(newU[j][i]);

    // add artificial viscosity
    for (int j = 2; j < N; j++)
        for (int i = 0; i < 3; i++)
            U[j][i] += nu * tau / h * (newU[j][i] - newU[j - 1][i]);
}

void (*stepAlgorithm)() = RoeStep;
```

```
void redraw();

void takeStep() {
    boundaryConditions(U);
    tau = CFL * h / cMax();
    stepAlgorithm();
    LapidusViscosity();
    redraw();
    ++step;
}

int mainWindow, controlWindow, plotWindow[4];
int margin = 10, controlHeight = 30;
int buttons = 4;
int algorithm = 0;
char algorithmName[][20] = {"Roe Solver", "Lax Wendroff",
                            "Upwind Godunov", "Lax Friedrichs"};
double yMin[] = {-1, -1, -0.2, -0.2};
double yMax[] = {2, 1, 3, 1.2};

void redraw() {
    for (int i = 0; i < 4; i++) {
        glutSetWindow(plotWindow[i]);
        glutPostRedisplay();
    }
}

void reshape(int w, int h) {
    glViewport(0, 0, w, h);
```

```
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        if (glutGetWindow() == plotWindow[0])
            gluOrtho2D(0, 1, yMin[0], yMax[0]);
        else if (glutGetWindow() == plotWindow[1])
            gluOrtho2D(0, 1, yMin[1], yMax[1]);
        else if (glutGetWindow() == plotWindow[2])
            gluOrtho2D(0, 1, yMin[2], yMax[2]);
        else if (glutGetWindow() == plotWindow[3])
            gluOrtho2D(0, 1, yMin[3], yMax[3]);
        else
            gluOrtho2D(0, w, 0, h);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
    }

    void display() {

        glClear(GL_COLOR_BUFFER_BIT);

        glColor3ub(0, 0, 0);
        glBegin(GL_LINES);
            glVertex2d(0, 0);
            glVertex2d(1, 0);
        glEnd();

        int plot = glutGetWindow();
        if (plot == plotWindow[0])
            glColor3ub(255, 0, 0);
        if (plot == plotWindow[1])
            glColor3ub(0, 255, 0);
        if (plot == plotWindow[2])
            glColor3ub(0, 0, 255);
        if (plot == plotWindow[3])
            glColor3ub(255, 0, 255);

        double avg = 0;
        glBegin(GL_LINE_STRIP);
            for (int j = 0; j < N; j++) {
                double y;
                if (plot == plotWindow[0])
                    y = U[j][0];
                if (plot == plotWindow[1])
                    y = U[j][1] / U[j][0];
                if (plot == plotWindow[2])
                    y = U[j][2];
                if (plot == plotWindow[3])
                    y = (U[j][2] - U[j][1] * U[j][1] / U[j][0] / 2) * (gama - 1);
                glVertex2d(j * h, y);
                avg += y;
            }
        glEnd();

        if (avg != 0.0)
            avg /= N;
        for (int i = 0; i < 4; i++) {
            if (plot == plotWindow[i]) {
                glRasterPos2d(0.05, yMin[i] + 0.92 * (yMax[i] - yMin[i]));
                char plotName[][20] = {"Density", "Velocity",
                                       "Energy", "Pressure"};
                char str[50];
                sprintf(str, "<%s> = %.4g", plotName[i], avg);
                for (int j = 0; j < strlen(str); j++)
                    glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, str[j]);
            }
        }
        glColor3ub(0, 0, 0);
        glutSwapBuffers();
    }

    void mouse(int button, int state, int x, int y) {

        static bool running = true;

        if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN) {
            if (running) {
                glutIdleFunc(NULL);
                running = false;
            } else {
                glutIdleFunc(takeStep);
                running = true;
            }
            redraw();
        }
    }

    void mouseControl(int button, int state, int x, int y) {
        if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN) {
            int w = glutGet(GLUT_WINDOW_WIDTH);
            algorithm = buttons * x / w;
            switch (algorithm) {
            case 0:
                stepAlgorithm = RoeStep;
                initialize();
                break;
            case 1:
                stepAlgorithm = LaxWendroffStep;
                initialize();
                break;
            case 2:
                stepAlgorithm = upwindGodunovStep;
                initialize();
                break;
            case 3:
                stepAlgorithm = LaxFriedrichsStep;
                initialize();
                break;
            default:
                break;
            }
            glutPostRedisplay();
            redraw();
        }
    }
```

```
void displayControl() {
    glClear(GL_COLOR_BUFFER_BIT);
    int w = glutGet(GLUT_WINDOW_WIDTH);
    int h = glutGet(GLUT_WINDOW_HEIGHT);
    double dx = w / buttons;
    for (int b = 0; b < buttons; b++) {
        if (b == algorithm)
            glColor3ub(255, 0, 0);
        else
            glColor3ub(0, 255, 0);
        glRectd(b * dx, 0, (b + 1) * dx, h);
        glColor3ub(0, 0, 0);
        glRasterPos2d((b + 0.2) * dx, 0.3 * h);
        char *str = algorithmName[b];
        for (int j = 0; j < strlen(str); j++)
            glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, str[j]);
    }
    glColor3ub(0, 0, 255);
    double d = 0.1 * h;
    glRectd(0, 0, w, d);
    glRectd(0, h - d, w, h);
    for (int b = 0; b <= buttons; b++) {
        double x = b * dx - d / 2;
        if (b == 0) x += d / 2;
        if (b == buttons) x -= d / 2;
        glRectd(x, 0, x + d, h);
    }
    glutSwapBuffers();
}
```

```
void makeSubWindows() {

    int w = glutGet(GLUT_WINDOW_WIDTH);
    int h = glutGet(GLUT_WINDOW_HEIGHT);
    int dx = (w - 3 * margin) / 2;
    int dy = (h - 4 * margin - controlHeight) / 2;
    for (int i = 0; i < 2; i++)
    for (int j = 0; j < 2; j++) {
        int x0 = margin * (1 + i) + i * dx;
        int y0 = margin * (1 + j) + j * dy;
        int n = 2 * i + j;
        glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
        plotWindow[n] = glutCreateSubWindow(mainWindow, x0, y0, dx, dy);
        glClearColor(1.0, 1.0, 0, 0);
        glShadeModel(GL_FLAT);
        glutDisplayFunc(display);
        glutReshapeFunc(reshape);
        glutMouseFunc(mouse);
    }
    controlWindow = glutCreateSubWindow(mainWindow, margin,
                                        h - margin - controlHeight,
                                        2 * dx + margin, controlHeight);
    glClearColor(0.0, 0.0, 1.0, 0.0);
    glutDisplayFunc(displayControl);
    glutReshapeFunc(reshape);
    glutMouseFunc(mouseControl);
}
```

```
void displayMain() {
    glClear(GL_COLOR_BUFFER_BIT);

    glutSwapBuffers();
}

void reshapeMain(int w, int h) {
    reshape(w, h);
    int dx = (w - 3 * margin) / 2;
    int dy = (h - 4 * margin - controlHeight) / 2;
    for (int i = 0; i < 2; i++)
    for (int j = 0; j < 2; j++) {
        glutSetWindow(plotWindow[2 * i + j]);
        glutPositionWindow(margin * (1 + i) + i * dx,
                           margin * (1 + j) + j * dy);
        glutReshapeWindow(dx, dy);
    }
    glutSetWindow(controlWindow);
    glutPositionWindow(margin, h - margin - controlHeight);
    glutReshapeWindow(w - 2 * margin, controlHeight);
}

int main(int argc, char *argv[]) {
    glutInit(&argc, argv);
    initialize();
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(800, 600);
    glutInitWindowPosition(100, 100);
    mainWindow = glutCreateWindow("Sod's shock tube problem");
```

```
        glutDisplayFunc(displayMain);
        glutReshapeFunc(reshapeMain);
        glutIdleFunc(takeStep);
        glClearColor(0.0, 0.0, 0.0, 0.0);
        glShadeModel(GL_FLAT);
        makeSubWindows();
        glutMainLoop();
    }
```

Translation of Laney's Riemann solver

The file `Riemann.h` contains an exact Riemann solver routine translated from a fortran subroutine on Laney's web site.

```
    // Translation of Laney's Riemann solver

    #ifndef RIEMANN_H_INCLUDED
    #define RIEMANN_H_INCLUDED

    inline double fg(double x) {
        const double gamma = 1.4;
        const double g2 = (gamma + 1) / (2 * gamma);
        return (x-1) / sqrt(g2 * (x - 1) + 1);
    }

    void Riemann(double *U1, double *U4, double *F) {

        const double gamma = 1.4;
        const double g1 = (gamma - 1) / (2 * gamma);
```

```
const double g2 = (gamma + 1) / (2 * gamma);
const double g3 = (gamma + 1) / (gamma - 1);
const double tol = 1e-10;

// compute primitive variables
double rho1 = U1[0];
double u1 = U1[1] / rho1;
double p1 = (U1[2] - rho1 * u1 * u1 / 2) * (gamma - 1);
double rho4 = U4[0];
double u4 = U4[1] / rho4;
double p4 = (U4[2] - rho4 * u4 * u4 / 2) * (gamma - 1);

// switch states if necessary so high pressure is on left
bool revflag = false;
if (p4 < p1) {
    double swap = p1; p1 = p4; p4 = swap;
    swap = u1; u1 = -u4; u4 = -swap;
    swap = rho1; rho1 = rho4; rho4 = swap;
    revflag = true;
}

double a1 = sqrt(gamma * p1 / rho1);
double a4 = sqrt(gamma * p4 / rho4);
double p = pow(p4/p1, g1);
double du = u4 - u1;

// apply the secant method
// initial guesses
double x = 0.05 * p4 / p1;
double y = 0.5 * p4 / p1;
double fx = p - pow(x, g1) / (1 + g1 * (gamma * du - a1 * fg(x)) / a4);
double fy = p - pow(y, g1) / (1 + g1 * (gamma * du - a1 * fg(y)) / a4);
bool converge = false;

for (int i = 1; i <= 20; i++) {

    double z = y - fy * (y - x) / (fy - fx);
    double fz = p - pow(z, g1) / (1 + g1 * (gamma * du - a1 * fg(z)) / a4);

    if (abs(fz) < tol && abs(z - y) < tol) {
        converge = true;
        break;
    }

    x = y;
    fx = fy;
    y = z;
    fy = fz;
}

if (!converge)
    cerr << "Warning:  secant failed to converge in Riemann" << endl;

// Compute shock
double p2 = p1 * x;
double u2 = u1 + a1 * fg(x) / gamma;
//     u2 = u4 + 2.*a4*(1.-(x**g1)/p)/(gamma-1.)
double a2 = a1 * sqrt(x * (g3 + x) / (1 + g3 * x));
```

```
    double rho2 = gamma * p2 / (a2 * a2);
    double s1 = u1 + a1 * sqrt(g2 *(x - 1) + 1);
    //     s1 = (rho1*u1 - rho2*u2)/(rho1-rho2)

    // Compute contact
    double p3 = p2;
    double u3 = u2;
    double a3 = a4 + 0.5 * (gamma - 1) * (u4 - u3);
    double s2 = u2;
    double rho3 = gamma * p3/(a3 * a3);

    // Compute expansion
    double s3 = u3 - a3;
    double s4 = u4 - a4;

    // Compute fluxes
    double f1, f2, f3, a, u, rho;
    if (revflag) {
        if (s4 > 0) {
            f1 = -rho4 * u4;
            f2 = rho4 * u4 * u4 + p4;
            f3 = -0.5 * rho4 * u4 * u4 * u4
                 - rho4 * a4 * a4 * u4 / (gamma - 1);
        } else if (s3 > 0) {
            u = (-(gamma-1.)*u4+2.*a4)/(gamma+1.);
            a = u;
            p = p4*pow(a/a4, 2.*gamma/(gamma-1.));
            if (a < 0 || p < 0) {
                cerr << "Negative a or p in Riemann" << endl;
            }
            rho = gamma*p/(a*a);
            f1 = -rho*u;
            f2 = rho*u*u + p ;
            f3 = -.5*rho*u*u*u - rho*a*a*u/(gamma-1.);
        } else if (s2 > 0) {
            f1 = -rho3*u3;
            f2 = rho3*u3*u3 + p3;
            f3 =  -.5*rho3*u3*u3*u3 - rho3*a3*a3*u3/(gamma-1.);
        } else if (s1 > 0) {
            f1 = -rho2*u2;
            f2 = rho2*u2*u2 + p2;
            f3 = -.5*rho2*u2*u2*u2 - rho2*a2*a2*u2/(gamma-1.);
        } else {
            f1 = -rho1*u1;
            f2 = rho1*u1*u1 + p1;
            f3 = -.5*rho1*u1*u1*u1 - rho1*a1*a1*u1/(gamma-1.);
        }
    } else {
        if(s4 > 0) {
            f1 = rho4*u4;
            f2 = rho4*u4*u4 + p4;
            f3 = .5*rho4*u4*u4*u4 + rho4*a4*a4*u4/(gamma-1.);
        } else if (s3 > 0) {
            u = ((gamma-1.)*u4+2.*a4)/(gamma+1.);
            a = u;
            p = p4*pow(a/a4, 2.*gamma/(gamma-1.));
            if (a < 0 || p < 0) {
                cerr << "Negative a or p in Riemann" << endl;
            }
            rho = gamma*p/(a*a);
            f1 = rho*u;
            f2 = rho*u*u + p;
            f3 = .5*rho*u*u*u + rho*a*a*u/(gamma-1.);
        } else if (s2 > 0) {
            f1 = rho3*u3;
            f2 = rho3*u3*u3 + p3;
            f3 =  .5*rho3*u3*u3*u3 + rho3*a3*a3*u3/(gamma-1.);
        } else if (s1 > 0) {
            f1 = rho2*u2;
            f2 = rho2*u2*u2 + p2;
            f3 = .5*rho2*u2*u2*u2 + rho2*a2*a2*u2/(gamma-1.);
        } else {
            f1 = rho1*u1;
            f2 = rho1*u1*u1 + p1;
            f3 = .5*rho1*u1*u1*u1 + rho1*a1*a1*u1/(gamma-1.);
        }
    }
    F[0] = f1;
    F[1] = f2;
    F[2] = f3;
}

#endif /* RIEMANN_H_INCLUDED */
```

Translation of G. Mellema's Roe Solver `roesol.f`

The file `RoeSolver.h` contains a translation from Fortran of the approximate Riemann solver for the Euler equations on Mellema's web site.

```
// Translation of G. Mellema's Roe Solver {\tt roesol.f}

#ifndef ROESOLVER_H_INCLUDED
#define ROESOLVER_H_INCLUDED

#include <algorithm>

void RoeSolve(double dr,                // spatial step
              double dt,                // time step
              double gamma,             // adiabatic index
              double *vol,              // volume factor for 3-D problem
              double **state,           // (rho, rho*u, e)  -- input
              double **flux,            // flux at cell boundaries -- output
              int meshr,                // number of interior points
              int& icntl                // diagnostic -- bad if != 0
             )
{
    const double tiny = 1e-30;
    const double sbpar1 = 2.0;
    const double sbpar2 = 2.0;

    // allocate temporary arrays
    double **fludif = new double* [meshr+2];
    double *rsumr   = new double [meshr+2];
    double *utilde  = new double [meshr+2];
    double *htilde  = new double [meshr+2];
    double *absvt   = new double [meshr+2];
    double *uvdif   = new double [meshr+2];
    double *ssc     = new double [meshr+2];
    double *vsc     = new double [meshr+2];
    double **a      = new double* [meshr+2];
    double **ac1    = new double* [meshr+2];
    double **ac2    = new double* [meshr+2];
    double **w      = new double* [meshr+2];
    double **eiglam = new double* [meshr+2];
    double **sgn    = new double* [meshr+2];
    double **fluxc  = new double* [meshr+2];
    double **fluxl  = new double* [meshr+2];
    double **fluxr  = new double* [meshr+2];
    double *ptest   = new double [meshr+2];
    int **isb       = new int* [meshr+2];
    for (int i = 0; i < meshr + 2; i++) {
        fludif[i] = new double [3];
        a[i]      = new double [3];
        ac1[i]    = new double [3];
        ac2[i]    = new double [3];
        w[i]      = new double [4];
        eiglam[i] = new double [3];
        sgn[i]    = new double [3];
        fluxc[i]  = new double [3];
        fluxl[i]  = new double [3];
        fluxr[i]  = new double [3];
        isb[i]    = new int [3];
    }

    // initialize control variable to 0
```

```
icntl = 0;

// find parameter vector w
for (int i = 0; i <= meshr + 1; i++) {
    w[i][0] = sqrt(vol[i] * state[i][0]);
    w[i][1] = w[i][0] * state[i][1] / state[i][0];
    w[i][3] = (gamma - 1) * (state[i][2] - 0.5 * state[i][1]
              * state[i][1] / state[i][0]);
    w[i][2] = w[i][0] * (state[i][2] + w[i][3]) / state[i][0];
}

// calculate the fluxes at the cell center
for (int i = 0; i <= meshr + 1; i++) {
    fluxc[i][0] = w[i][0] * w[i][1];
    fluxc[i][1] = w[i][1] * w[i][1] + vol[i] * w[i][3];
    fluxc[i][2] = w[i][1] * w[i][2];
}

// calculate the fluxes at the cell walls
// assuming constant primitive variables
for (int n = 0; n < 3; n++) {
    for (int i = 1; i <= meshr + 1; i++) {
        fluxl[i][n] = fluxc[i - 1][n];
        fluxr[i][n] = fluxc[i][n];
    }
}

// calculate the flux differences at the cell walls
for (int n = 0; n < 3; n++)
```

```
        for (int i = 1; i <= meshr + 1; i++)
            fludif[i][n] = fluxr[i][n] - fluxl[i][n];

    // calculate the tilded state variables = mean values at the interfaces
    for (int i = 1; i <= meshr + 1; i++) {
        rsumr[i] = 1 / (w[i - 1][0] + w[i][0]);

        utilde[i] = (w[i - 1][1] + w[i][1]) * rsumr[i];
        htilde[i] = (w[i - 1][2] + w[i][2]) * rsumr[i];

        absvt[i] = 0.5 * utilde[i] * utilde[i];
        uvdif[i] = utilde[i] * fludif[i][1];

        ssc[i] = (gamma - 1) * (htilde[i] - absvt[i]);
        if (ssc[i] > 0.0)
            vsc[i] = sqrt(ssc[i]);
        else {
            vsc[i] = sqrt(abs(ssc[i]));
            ++icntl;
        }
    }

    // calculate the eigenvalues and projection coefficients for each
    // eigenvector
    for (int i = 1; i <= meshr + 1; i++) {
        eiglam[i][0] = utilde[i] - vsc[i];
        eiglam[i][1] = utilde[i];
        eiglam[i][2] = utilde[i] + vsc[i];
        for (int n = 0; n < 3; n++)
```

```
            sgn[i][n] = eiglam[i][n] < 0.0 ?  -1 :  1;
        a[i][0] = 0.5 * ((gamma - 1) * (absvt[i] * fludif[i][0] + fludif[i][2]
                  - uvdif[i]) - vsc[i] * (fludif[i][1] - utilde[i]
                  * fludif[i][0])) / ssc[i];
        a[i][1] = (gamma - 1) * ((htilde[i] - 2 * absvt[i]) * fludif[i][0]
                  + uvdif[i] - fludif[i][2]) / ssc[i];
        a[i][2] = 0.5 * ((gamma - 1) * (absvt[i] * fludif[i][0] + fludif[i][2]
                  - uvdif[i]) + vsc[i] * (fludif[i][1] - utilde[i]
                  * fludif[i][0])) / ssc[i];
    }

    // divide the projection coefficients by the wave speeds
    // to evade expansion correction
    for (int n = 0; n < 3; n++)
        for (int i = 1; i <= meshr + 1; i++)
            a[i][n] /= eiglam[i][n] + tiny;

    // calculate the first order projection coefficients ac1
    for (int n = 0; n < 3; n++)
        for (int i = 1; i <= meshr + 1; i++)
            ac1[i][n] = - sgn[i][n] * a[i][n] * eiglam[i][n];

    // apply the 'superbee' flux correction to made 2nd order projection
    // coefficients ac2
    for (int n = 0; n < 3; n++) {
        ac2[1][n] = ac1[1][n];
        ac2[meshr + 1][n] = ac1[meshr + 1][n];
    }
```

```
double dtdx = dt / dr;
for (int n = 0; n < 3; n++) {
    for (int i = 2; i <= meshr; i++) {
        isb[i][n] = i - int(sgn[i][n]);
        ac2[i][n] = ac1[i][n] + eiglam[i][n] *
                    ((max(0.0, min(sbpar1 * a[isb[i][n]][n], max(a[i][n],
                    min(a[isb[i][n]][n], sbpar2 * a[i][n])))) +
                    min(0.0, max(sbpar1 * a[isb[i][n]][n], min(a[i][n],
                    max(a[isb[i][n]][n], sbpar2 * a[i][n])))) ) *
                    (sgn[i][n] - dtdx * eiglam[i][n]));
    }
}

// calculate the final fluxes
for (int i = 1; i <= meshr + 1; i++) {
    flux[i][0] = 0.5 * (fluxl[i][0] + fluxr[i][0] + ac2[i][0]
                 + ac2[i][1] + ac2[i][2]);
    flux[i][1] = 0.5 * (fluxl[i][1] + fluxr[i][1] +
                 eiglam[i][0] * ac2[i][0] + eiglam[i][1] * ac2[i][1] +
                 eiglam[i][2] * ac2[i][2]);
    flux[i][2] = 0.5 * (fluxl[i][2] + fluxr[i][2] +
                 (htilde[i] - utilde[i] * vsc[i]) * ac2[i][0] +
                 absvt[i] * ac2[i][1] +
                 (htilde[i] + utilde[i] * vsc[i]) * ac2[i][2]);
}

// calculate test variable for negative pressure check
for (int i = 1; i <= meshr; i++) {
    ptest[i] = dr * vol[i] * state[i][1] +
               dt * (flux[i][1] - flux[i + 1][1]);
    ptest[i] = - ptest[i] * ptest[i] + 2 * (dr * vol[i] * state[i][0] +
               dt * (flux[i][0] - flux[i + 1][0])) * (dr * vol[i] *
               state[i][2] + dt * (flux[i][2] - flux[i + 1][2]));
}

// check for negative pressure/internal energy and set fluxes
// left and  right to first order if detected
for (int i = 1; i <= meshr; i++) {
    if (ptest[i] <= 0.0 || (dr * vol[i] * state[i][0] + dt * (flux[i][0]
                            - flux[i + 1][0])) <= 0.0) {

        flux[i][0] = 0.5 * (fluxl[i][0] + fluxr[i][0] +
            ac1[i][0] + ac1[i][1] + ac1[i][2]);
        flux[i][1] = 0.5 * (fluxl[i][1] + fluxr[i][1] +
            eiglam[i][0] * ac1[i][0] + eiglam[i][1] * ac1[i][1] +
            eiglam[i][2] * ac1[i][2]);
        flux[i][2] = 0.5 * (fluxl[i][2] + fluxr[i][2] +
            (htilde[i]-utilde[i] * vsc[i]) * ac1[i][0] +
            absvt[i] * ac1[i][1] +
            (htilde[i] + utilde[i] * vsc[i]) * ac1[i][2]);
        flux[i + 1][0] = 0.5 * (fluxl[i + 1][0] + fluxr[i + 1][0] +
             ac1[i + 1][0] + ac1[i + 1][1] + ac1[i + 1][2]);
        flux[i + 1][1] = 0.5 * (fluxl[i + 1][1] + fluxr[i + 1][1] +
             eiglam[i + 1][0] * ac1[i + 1][0] + eiglam[i + 1][1] *
             ac1[i + 1][1] + eiglam[i + 1][2] * ac1[i + 1][2]);
        flux[i + 1][2] = 0.5 * (fluxl[i + 1][2] + fluxr[i + 1][2] +
             (htilde[i + 1] - utilde[i + 1] * vsc[i + 1]) * ac1[i + 1][0]
             + absvt[i + 1] * ac1[i + 1][1] +
             (htilde[i + 1] + utilde[i + 1] * vsc[i + 1]) * ac1[i + 1][2]);

        // Check if it helped, set control variable if not

        ptest[i] = (dr * vol[i] * state[i][1] +
                   dt * (flux[i][1] - flux[i + 1][1]));
        ptest[i] = 2.0 * (dr * vol[i] * state[i][0]
            + dt * (flux[i][0]-flux[i + 1][0])) * (dr * vol[i] *
            state[i][2] + dt * (flux[i][2] - flux[i + 1][2]))
            - ptest[i] * ptest[i];
        if (ptest[i] <= 0.0 || (dr * vol[i] * state[i][0] +
                dt * (flux[i][0] - flux[i + 1][0])) <= 0.0)
            icntl = icntl + 1;
    }
}

// free temporary arrays
for (int i = 0; i < meshr + 2; i++) {
    delete [] fludif[i];
    delete [] a[i];
    delete [] ac1[i];
    delete [] ac2[i];
    delete [] w[i];
    delete [] eiglam[i];
    delete [] sgn[i];
    delete [] fluxc[i];
    delete [] fluxl[i];
    delete [] fluxr[i];
    delete [] isb[i];
```

```
        }
        delete [] fludif;
        delete [] rsumr;
        delete [] utilde;
        delete [] htilde;
        delete [] absvt;
        delete [] uvdif;
        delete [] ssc;
        delete [] vsc;
        delete [] a;
        delete [] ac1;
        delete [] ac2;
        delete [] w;
        delete [] eiglam;
        delete [] sgn;
        delete [] fluxc;
        delete [] fluxl;
        delete [] fluxr;
        delete [] ptest;
        delete [] isb;
    }

    #endif /* ROESOLVER_H_INCLUDED */
```