One Dimensional Burgers’ Equation


\[ \frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} = \nu \frac{\partial^2 u}{\partial x^2}, \]

as a simple model of shock propagation. This is basically a Navier-Stokes equation in one dimension without a pressure term. The convective term on the left is nonlinear. The diffusive term on the right represents the effects of viscosity.

The development of a shock can be seen by letting the kinematic viscosity \( \nu = 0 \). This gives the *inviscid* Burgers’ equation

\[ \frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} = 0. \]

Compare this with the linear equation

\[ \frac{\partial u}{\partial t} + c \frac{\partial u}{\partial x} = 0, \]

where \( c \) is a constant. The linear equation has the solution

\[ u(x, t) = f(x - ct), \]

where \( f \) is any differentiable function. This solution represents a wave form with shape \( f(x) \) moving to the right with constant speed \( c \).

Now, in the inviscid Burgers’ equation, the “speed” \( c = u \), i.e., the instantaneous speed of the wave form is proportional to its amplitude \( u \). This implies that a peak in the wave travels faster than a trough, which implies that the wave will tend to *break*. This is not allowed mathematically because breaking implies that the solution \( u(x, t) \) becomes multiple valued. What actually happens is that a *shock front* develops: this is a moving point at which the solution is discontinuous.

The viscous term in Burgers’ equation has two effects. First, it causes the wave amplitude to damp to zero in a diffusive fashion. Secondly, it prevents the development of a mathematical singularity at the shock front: the amplitude is continuous albeit varying very rapidly through the front.
Finite Difference Algorithms and their Stability

Consider the simpler *advection* equation

\[
\frac{\partial u}{\partial t} + c \frac{\partial u}{\partial x} = 0 .
\]

We discretize the variable \( x = x_0 + jh, \ j = 0, 1, 2 \ldots \) and the time \( t = t_0 + n\tau, \ n = 0, 1, 2, \ldots \). The solution \( u(x, t) \) is represented by \( u^n_j \).

**Forward Time Centered Space (FTCS) algorithm**

\[
u^{n+1}_j = u^n_j - \frac{c\tau}{2h} (u^n_{j+1} - u^n_{j-1}).
\]

This algorithm happens to be unstable. This can be seen from a *von Neumann stability analysis*, which employs an approximate solution of the form

\[
u(x, t) = z^t e^{ikx},
\]

where \( k \) is the wave number of a spatial Fourier component of the solution, and \( z \) is an *amplification factor*. Substituting this form into the discretized equation gives

\[
z^\tau = 1 - \frac{c\tau}{2h} (e^{ikh} - e^{-ikh}) = 1 - i \frac{c\tau}{h} \sin(kh).
\]

The magnitude of the amplification per time step is

\[
|z^\tau| = \sqrt{1 + \left( \frac{c\tau}{h} \right)^2 \sin^2(kh)},
\]

which is greater than unity. This shows that the algorithm is unconditionally unstable: the solution grows exponentially as a function of time if \( \sin(kh) \neq 0 \).

**The Lax differencing scheme**
The mathematician Peter Lax discovered a simple solution to the instability problem with the FTCS scheme:

\[ u_{j}^{n+1} = \frac{1}{2} (u_{j+1}^{n} + u_{j-1}^{n}) - \frac{c\tau}{2h} (u_{j+1}^{n} - u_{j-1}^{n}) . \]

It is easy to see that

\[ z^\tau = \frac{1}{2} (e^{ikh} + e^{-ikh}) - \frac{c\tau}{2h} (e^{ikh} - e^{-ikh}) = \cos(kh) - i \frac{c\tau}{h} \sin(kh) . \]

The amplification per time step is now

\[ |z^\tau| = \sqrt{\cos^2(kh) + \left( \frac{c\tau}{h} \right)^2 \sin^2(kh)} , \]

which is less than unity only if the Courant-Friedrichs-Lewy (CFL) stability criterion

\[ \left| \frac{c\tau}{h} \right| \leq 1 , \]

is satisfied.

**Program to solve the 1-D Burgers’ Equation**

```cpp
#include <cmath>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <iostream>

// Program to solve the 1-D Burgers’ Equation
```
using namespace std;

#include <GL/glut.h>

const double pi = 4 * atan(1.0); // value of pi

double L = 1; // size of periodic region
int N = 200; // number of grid points
double h; // lattice spacing
double tau; // time step
double CFLRatio = 1; // Courant-Friedrichs-Lewy ratio tau/h
double nu = 1e-6; // kinematic viscosity

double *u; // the solution
double *uNew; // for updating
double *F; // the flow
double *uPlus, *uMinus; // for Godunov scheme

void allocate() {
    static int oldN = 0;
    if (oldN != N) {
        if (u != 0)
            delete[] u; delete[] uNew; delete[] F;
        delete[] uPlus; delete[] uMinus;
    }
}

int initialWaveform = SINE; // sine function, step, etc.

int step; // integration step number
oldN = N;
u = new double [N];
uNew = new double [N];
F = new double [N];
uPlus = new double [N];
uMinus = new double [N];
}

void initialize() {

allocate();
h = L / N;

double uMax = 0;
for (int i = 0; i < N; i++) {
    double x = i * h;
    switch (initialWaveform) {
        case SINE:
            u[i] = sin(2 * pi * x) + 0.5 * sin(pi * x);
            break;
        case STEP:
            u[i] = 0;
            if (x > L / 4 && x < 3 * L / 4)
                u[i] = 1;
            break;
        default:
            u[i] = 1;
            break;
    }
}
if (abs(u[i]) > uMax)
    uMax = abs(u[i]);

tau = CFLRatio * h / uMax;
step = 0;

Integration algorithms

void (*integrationAlgorithm)();
void redraw();

void takeStep() {
    integrationAlgorithm();
    double *swap = u;
    u = uNew;
    uNew = swap;
    redraw();
    ++step;
}

void FTCS() {
    for (int j = 0; j < N; j++) {
        int jNext = j < N - 1 ? j + 1 : 0;
        int jPrev = j > 0 ? j - 1 : N - 1;
    }
}
Lax algorithm

```cpp
void Lax() {
    for (int j = 0; j < N; j++) {
        int jNext = j < N - 1 ? j + 1 : 0;
        int jPrev = j > 0 ? j - 1 : N - 1;
        uNew[j] = (u[jNext] + u[jPrev]) / 2
            - u[j] * tau / (2 * h) * (u[jNext] - u[jPrev])
    }
}
```

Lax-Wendroff algorithm

The Lax-Wendroff algorithm is constructed in two steps. First, the time and convective derivatives are expressed in terms of a flow function $F$ as follows:

$$\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} = \frac{\partial u}{\partial t} + \frac{\partial F}{\partial x}, \quad F(x, t) = \frac{1}{2} u^2(x, t).$$

This is the form of a conservation equation with $F$ representing the current of the quantity $u$.

Second, a Taylor series expansion in the time step $\tau$ of all variables is made and terms up to and including $O(\tau^2)$ are retained, e.g.,

$$u(x, t + \tau) = u(x, t) + \tau \frac{\partial u}{\partial t} + \frac{\tau^2}{2} \frac{\partial^2 u}{\partial t^2} + O(\tau^3).$$
The resulting algorithm can be expressed as a two-step formula:

\[
\begin{align*}
    u_{j+\frac{1}{2}}^* &= \frac{1}{2} (u_j^n + u_{j+1}^n) - \frac{\tau}{2h} \left( F_{j+1}^n - F_j^n \right) + \\
    &\quad \frac{\nu \tau}{2h^2} \left[ \frac{1}{2} (u_{j+1}^n + u_{j-1}^n - 2u_j^n) + \frac{1}{2} (u_{j+2}^n + u_j^n - 2u_{j+1}^n) \right], \\
    u_j^{n+1} &= u_j^n - \frac{\tau}{h} \left( F_{j+\frac{1}{2}}^* - F_{j-\frac{1}{2}}^* \right) + \frac{\nu \tau}{h^2} (u_{j+1}^n + u_{j-1}^n - 2u_j^n).
\end{align*}
\]

```cpp
void LaxWendroff() {
    for (int j = 0; j < N; j++)
        F[j] = u[j] * u[j] / 2;
    for (int j = 0; j < N; j++) {
        int jMinus1 = j > 0 ? j - 1 : N - 1;
        int jPlus1 = j < N - 1 ? j + 1 : 0;
        int jPlus2 = jPlus1 < N - 1 ? jPlus1 + 1 : 0;
        uNew[j] = (u[j] + u[jPlus1]) / 2 -
                   (tau / 2 / h) * (F[jPlus1] - F[j]) +
                   (nu * tau / (2 * h * h)) * (u[jPlus1] + u[jMinus1] - 2 * u[j]) / 2 +
                   (u[jPlus2] + u[j] - 2 * u[jPlus1]) / 2;
    }
    for (int j = 0; j < N; j++)
    for (int j = 0; j < N; j++) {
        int jMinus1 = j > 0 ? j - 1 : N - 1;
        int jPlus1 = j < N - 1 ? j + 1 : 0;
        uNew[j] = u[j] - (tau / h) * (F[j] - F[jMinus1]) +
                   (nu * tau / (2 * h * h)) * (u[jPlus1] + u[jMinus1] - 2 * u[j]) / 2 +
                   (u[jPlus2] + u[j] - 2 * u[jPlus1]) / 2;
    }
}```
Godunov Scheme

This type of scheme was introduced by S.K. Godunov, *Mat. Sb.* 47, 271 (1959). This is an *upwind* differencing scheme which makes use of the solution to a local *Riemann problem*.

A *Riemann problem* is an initial value problem for a partial differential equation with a *piecewise constant* initial value function, i.e., the solution at $t = 0$ is a step function. A *Riemann solver* is an exact or approximate algorithm for solving a Riemann problem.

The basic formula for updating $u$ is

$$u_j^{n+1} = u_j^n - \frac{\tau}{h} \left[ F_{j+\frac{1}{2}} - F_{j-\frac{1}{2}} \right] + \frac{\nu \tau}{h^2} \left[ u_{j+1} + u_{j-1} - 2u_j \right],$$

where $F_{j \pm \frac{1}{2}}$ represents the average flux on the cells to the right and left of the lattice point $j$ respectively. These average flux values are computed from Riemann problems in the cells to the right and left of $j$ using *upwind* initial data

$$u_j^{(+)} = \begin{cases} u_j & \text{if } u_j > 0 \\ 0 & \text{otherwise} \end{cases} \quad u_j^{(-)} = \begin{cases} u_j & \text{if } u_j < 0 \\ 0 & \text{otherwise} \end{cases}$$

The solution to the Riemann problem on the left cell is

$$F_{j-\frac{1}{2}} = \max \left\{ \frac{1}{2} (u_{j-1}^{(+)})^2, \frac{1}{2} (u_{j}^{(-)})^2 \right\},$$

and for the cell on the right

$$F_{j+\frac{1}{2}} = \max \left\{ \frac{1}{2} (u_{j}^{(+)})^2, \frac{1}{2} (u_{j+1}^{(-)})^2 \right\}.$$
void Godunov() {
    for (int j = 0; j < N; j++) {
        uMinus[j] = u[j] < 0 ? u[j] : 0;
    }
    for (int j = 0; j < N; j++) {
        int jNext = j < N - 1 ? j + 1 : 0;
        int jPrev = j > 0 ? j - 1 : N - 1;
        double f2 = uMinus[j] * uMinus[j] / 2;
        F[jPrev] = f1 > f2 ? f1 : f2;
        f1 = uPlus[j] * uPlus[j] / 2;
        f2 = uMinus[jNext] * uMinus[jNext] / 2;
        F[j] = f1 > f2 ? f1 : f2;
        uNew[j] -= (tau / h) * (F[j] - F[jPrev]);
    }
}

void reshape(int w, int h) {

}

Graphics

int mainWindow, solutionWindow, controlWindow;
int margin = 10;
int controlHeight = 30;
void reshape(int w, int h) {
glViewport(0, 0, w, h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0, w, 0, h);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
}
void redraw() {
    glutSetWindow(solutionWindow);
    glutPostRedisplay();
}
void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    glutSwapBuffers();
}
void displaySolution() {
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3ub(255, 255, 255);
    glBegin(GL_LINE_STRIP);
    for (int i = 0; i < N; i++) {
        int iNext = i < N - 1 ? i + 1 : 0;
        glVertex2d(i * h, u[i]);
        glVertex2d((i + 1) * h, u[iNext]);
    }
    glEnd();
char str[100];
sprintf(str, "CFL Ratio = %.4f  nu = %.4g  t = %.4f",
        CFLRatio, nu, step * tau);
glRasterPos2d(0.02, -0.95);
for (int j = 0; j < strlen(str); j++)
    glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, str[j]);
glutSwapBuffers();
}

void (*method[])() = {FTCS, Lax, LaxWendroff, Godunov};
char methodName[][20] = {"FTCS", "Lax", "Lax Wendroff", "Godunov"};

void displayControl() {
    glClear(GL_COLOR_BUFFER_BIT);
    int w = glutGet(GLUT_WINDOW_WIDTH);
    int h = glutGet(GLUT_WINDOW_HEIGHT);
    for (int i = 0; i < 4; i++) {
        if (method[i] == integrationAlgorithm)
            glColor3ub(255, 0, 0);
        else
            glColor3ub(0, 0, 255);
        glRectd((i + 0.025) * w / 4, 0.1 * h, (i + 0.975) * w / 4, 0.9 * h);
        glColor3ub(255, 255, 255);
        glRasterPos2d((i + 0.2) * w / 4, 0.3 * h);
        for (int j = 0; j < strlen(methodName[i]); j++)
            glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, methodName[i][j]);
    }
    glutSwapBuffers();
}
void reshapeMain(int w, int h) {
    reshape(w, h);

    glutSetWindow(solutionWindow);
    glutPositionWindow(margin, margin);
    glutReshapeWindow(w - 2 * margin, h - 3 * margin - controlHeight);

    glutSetWindow(controlWindow);
    glutPositionWindow(margin, h - margin - controlHeight);
    glutReshapeWindow(w - 2 * margin, controlHeight);
}

void reshapeSolution(int w, int h) {
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0, 1, -1, +1.5);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void mouseSolution(int button, int state, int x, int y) {
    static bool running = false;

    switch (button) {
        case GLUT_LEFT_BUTTON:
            if (state == GLUT_DOWN) {
                if (running) {
                    if (running) {

PHY 411-506 Computational Physics II
glutIdleFunc(NULL);
running = false;
} else {
    glutIdleFunc(takeStep);
    running = true;
}
break;
default:
break;
}

void mouseControl(int button, int state, int x, int y) {
    if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN) {
        int w = glutGet(GLUT_WINDOW_WIDTH);
        int algorithm = int(x / double(w) * 4);
        if (algorithm >= 0 && algorithm < 4)
        integrationAlgorithm = method[algorithm];
        glutPostRedisplay();
    }
}

void makeMainWindow() {
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(600, 400);
    glutInitWindowPosition(100, 100);
    mainWindow = glutCreateWindow("One-dimensional Burgers’ Equation");
glClearColor(1.0, 1.0, 1.0, 0.0);
glShadeModel(GL_FLAT);
glutDisplayFunc(display);
glutReshapeFunc(reshapeMain);
}

void solutionMenu(int menuItem) {
    switch (menuItem) {
        case 1:
            initialWaveform = SINE;
            break;
        case 2:
            initialWaveform = STEP;
            break;
        default:
            break;
    }
    initialize();
    glutPostRedisplay();
}

void makeSolutionWindow() {
    glutSetWindow(mainWindow);
    int w = glutGet(GLUT_WINDOW_WIDTH);
    int h = glutGet(GLUT_WINDOW_HEIGHT);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    solutionWindow = glutCreateSubWindow(mainWindow, margin, margin, w - 2 * margin, h - 3 * margin - controlHeight);
    glClearColor(0.0, 0.0, 0.0, 0.0);
glShadeModel(GL_FLAT);
glutDisplayFunc(displaySolution);
glutReshapeFunc(reshapeSolution);
glutMouseFunc(mouseSolution);
integrationAlgorithm = Lax;
glutCreateMenu(solutionMenu);
glutAddMenuEntry("Initial Sine Waveform", 1);
glutAddMenuEntry("Initial Step Waveform", 2);
glutAttachMenu(GLUT_RIGHT_BUTTON);
}

void makeControlWindow() {
   glutSetWindow(mainWindow);
   int w = glutGet(GLUT_WINDOW_WIDTH);
   int h = glutGet(GLUT_WINDOW_HEIGHT);
   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
   controlWindow = glutCreateSubWindow(mainWindow,
                                     margin, h - margin - controlHeight,
                                     w - 2 * margin, controlHeight);
   glClearColor(0.0, 1.0, 0.0, 0.0);
   glShadeModel(GL_FLAT);
   glutDisplayFunc(displayControl);
   glutReshapeFunc(reshape);
   glutMouseFunc(mouseControl);
}

int main(int argc, char *argv[]) {
   glutInit(&argc, argv);
   if (argc > 1)
CFLRatio = atof(argv[1]);
if (argc > 2)
    nu = atof(argv[2]);
initialize();
makeMainWindow();
makeSolutionWindow();
makeControlWindow();
glutMainLoop();
}