Fluid Dynamics in Astrophysics and Cosmology

Large systems of particles are approximated in Fluid dynamics by continuous media which obey partial differential equations. Computational fluid dynamics is important in astrophysics, cosmology, biological physics, geophysics, and numerous areas of engineering and technology.

Fluid Dynamics in Cosmology

Figure 1: A historical time-line showing the major evolutionary stages of our Universe according to the standard model, from the earliest moments of the Planck era to the present. From P. Anninos, Living Reviews in Relativity 4 (2001).
Partial Differential Equations of Inflationary Cosmology


The Einstein field equations of general relativity can be written in the form

\[ R_{\mu\nu} - \frac{1}{2} g_{\mu\nu} R + g_{\mu\nu} \Lambda = \frac{8\pi G}{c^4} T_{\mu\nu}. \]

The homogeneous and isotropic FLRW metric

\[ ds^2 = -dt^2 + a^2(t) \left[ \frac{dr^2}{1 - kr^2} + r^2 (d\theta^2 + \sin^2 \theta d\phi^2) \right] \]

is used in the standard model of inflationary cosmology.

In the simplest approximation the density \( \rho(t, r) \) and pressure \( p(t, r) \) of matter and radiation that contribute to the stress-energy tensor \( T_{\mu\nu} \) do not depend on \( r \), the evolution of the universe is determined by ordinary differential equations

\[ p = \begin{cases} \frac{\rho}{3} & \text{Radiation} \\ 0 & \text{Non-relativistic matter} \end{cases} \quad \text{Equation of State} \]

\[ H = \frac{8\pi G}{3} \rho - \frac{k}{a^2} \quad \text{Friedmann equation} \]

\[ \dot{\rho} + 3H(\rho + p) = 0 \quad \text{Fluid equation} \]

This system can be solved using ODE algorithms such as Runge-Kutta.
The full system of PDEs to be solved in the perfect fluid approximation with stress-energy tensor (see Anninos LRR Section 6.2.4)

\[ T_{\mu\nu} = \rho \left( 1 + \epsilon + \frac{P}{\rho} \right) u_\mu u_\nu + Pg_{\mu\nu}, \]

where \( \epsilon(t, r) \) is the internal energy and \( u_\mu \) is the 4-velocity of the fluid, can be written

\[
\begin{align*}
\frac{\partial D}{\partial t} + \frac{\partial(DV^i)}{\partial x^i} &= 0 \\
\frac{\partial E}{\partial t} + \frac{\partial(EV^i)}{\partial x^i} + P \frac{\partial W}{\partial t} + P \frac{\partial(WV^i)}{\partial x^i} &= 0 \\
\frac{\partial S_i}{\partial t} + \frac{\partial(S_i V^j)}{\partial x^i} - S^\mu S^\nu \frac{\partial g_{\mu\nu}}{\partial x^i} - \sqrt{-g} \frac{\partial(P)}{\partial x^i} &= 0
\end{align*}
\]

\[ P - (\Gamma - 1) \frac{E}{W} = 0 \quad \text{(Equation of State)} \]

where

\[ W = \sqrt{-g} u^0, \quad D = W \rho, \quad E = W \rho \epsilon, \quad S_i = W \rho \left( 1 + \epsilon + \frac{P}{\rho} \right) u_i. \]

Even for this simple case of a perfect fluid, this is a complex set of coupled partial differential equations.

To drive inflation the stress-energy tensor of the scalar inflaton field \( \phi(t, r) \) must be added to the system

\[ T_{\mu\nu} = \partial_\mu \phi \partial_\nu \phi - \frac{1}{2} g_{\mu\nu} g^{\rho\sigma} \partial_\rho \phi \partial_\sigma \phi - g_{\mu\nu} V(\phi). \]
Fluid Dynamics in Gravitational Collapse

The formation of various compact astrophysical objects can be modeled by gravitational collapse.

The images Kepler’s supernova of Type 1a formed from the collapse of a white dwarf to a neutron star, a Hypernova resulting from collapse to a black hole, and Sagittarius A* the supermassive black hole at the center of our galaxy.

An early classic numerical computation of gravitational collapse was done by M.M. May and R.H. White Phys. Rev. 141, 1232-1241 (1966). They use a Schwarzschild metric and stress-energy tensor

\[ ds^2 = a^2(\mu, t) c^2 dt^2 - b^2(\mu, t) d\mu^2 - R^2(\mu, t) d\Omega^2, \quad T_1^{1} = T_2^{2} = T_3^{3} = P, \quad T_0^{0} = -\rho(c^2 + \epsilon), \]

where \( 2\pi R(\mu, t) \) is the circumference of a circle through point \((\mu, t)\) and the condition \(4\pi \rho R^2 b = 1\) defines \(\mu\) as the rest mass between \((\mu, t)\) and the origin.

Newtonian Fluid Dynamics

The equations of fluid dynamics follow from conservation of mass and momentum.

Consider a volume $V$ inside the fluid. The mass of fluid in this volume is given by

$$\int dV \; \rho ,$$

where $\rho$ is the fluid density. The rate at which this mass decreases is determined by the rate at which fluid leaves the volume

$$\frac{d}{dt} \int dV \; \rho = - \int dS \cdot \rho \mathbf{u} ,$$

where $\mathbf{u}$ is the fluid velocity and the integral on the right is taken over the surface of the volume with $dS$ being a surface element with direction along the outward normal. Using the divergence theorem

$$\int dS \cdot \rho \mathbf{u} = \int dV \; \nabla \cdot \rho \mathbf{u} ,$$

we obtain the continuity equation

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \rho \mathbf{u} = 0 .$$

Next, consider conservation of momentum

$$\rho \frac{d\mathbf{u}}{dt} = \mathbf{F} ,$$
which is just Newton's equation of motion for an element of fluid with unit mass. The total derivative on the left has two contributions

\[
\frac{du}{dt} = \frac{\partial u}{\partial t} + (u \cdot \nabla) u ,
\]

the first term on the right represents the change in fluid velocity with \( t \) at a fixed point in space, and the second \textit{advective} term represents the change in fluid velocity due to motion of fluid from neighboring points in space.

The force density \( \mathbf{F} \) has three contributions:

- \textit{external or body forces} acting on the fluid, for example the force of gravity
  \[
  \mathbf{F}_{\text{gravity}} = \rho \mathbf{g} ,
  \]
  where \( \mathbf{g} \) is the acceleration due to gravity

- \textit{pressure forces} due to neighboring fluid elements
  \[
  - \int p \, d\mathbf{S} = - \int \nabla p \, dV ,
  \]
  \[
  \mathbf{F}_{\text{pressure}} = -\nabla p ,
  \]
  where \( p \) is the fluid pressure and the integrals are taken over the surface and volume of the element, respectively

- \textit{viscous forces} due to internal friction or shearing stresses in the moving fluid
  \[
  \mathbf{F}_{\text{viscous}} = \mu \nabla^2 \mathbf{u} + (\mu + \xi) \nabla (\nabla \cdot \mathbf{u}) ,
  \]
where $\mu$ is the *dynamic viscosity coefficient* and $\xi$ is the *bulk viscosity coefficient* of the fluid. A special case that is interesting for many applications is that of *incompressible flow*

$$\rho = \text{constant}, \quad \nabla \cdot \mathbf{u} = 0.$$  

Taking these forces into account results in the *Navier-Stokes equations* for incompressible viscous flow:

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = \mathbf{g} - \frac{1}{\rho} \nabla p + \nu \nabla^2 \mathbf{u},$$

where $\nu = \mu/\rho$ is the *kinematic viscosity*.

**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 \( c \).

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 \textit{break}. Breaking of two-dimensional surface waves is of course very familiar, see for example, Hokusai’s Great Wave Off Kanagawa.

In one dimension, breaking is not allowed mathematically because breaking implies that the solution \( u(x, t) \) becomes multiple valued. What actually happens is that a \textit{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.
// Burgers equation in one dimension

#include <cmath>
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <string>
#include <sstream>
#include <vector>
using namespace std;

#ifdef __APPLE__
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif

const double pi = 4 * atan(1.0);

double L = 1;  // size of periodic region
int N = 200; // number of grid points
double h; // lattice spacing
double t; // time
double uMax; // maximum wave amplitude
double tau; // time step
double CFLRatio = 1; // Courant-Friedrichs-Lewy ratio tau/h
eum {SINE, STEP};
int initialWaveform = SINE; // sine function, step, etc.

double nu = 1e-6; // kinematic viscosity
vector<double> u, uNew; // the solution and its update
vector<double> F; // the flow
vector<double> uPlus, uMinus; // for Godunov scheme
int step; // integration step number

void initialize() {
    u = vector<double>(N);
uNew = vector<double>(N);
    F = vector<double>(N);
uPlus = vector<double>(N);
uMinus = vector<double>(N);

    h = L / N;
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;
t = 0;
step = 0;
void (*integrationAlgorithm)();
void redraw();

double T = 5;  // time to cross screen
double framesPerSec = 50;  // animation rate for screen redrews

void takeStep() {
    static clock_t clockStart;
    static bool done;
    if (!done) {
        double t0 = t;
        do {
            integrationAlgorithm();
            u = uNew;
            t += tau;
            ++step;
        } while (abs(uMax * (t - t0)) < L / T / framesPerSec);
        done = true;
    }
    clock_t clockNow = clock();
    double seconds = (clockNow - clockStart) / double(CLOCKS_PER_SEC);
    if ( seconds < 1 / framesPerSec ) {
        return;
    }
Finite Difference Algorithms: Forward Time Centered Space (FTCS)

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 \( u_j^n \).

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

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 $\sin(kh) \neq 0$.

```cpp
def FTCS():
    for (int j = 0; j < N; j++) {
        int jNext = j < N - 1 ? j + 1 : 0;
        int jPrev = j > 0 ? j - 1 : N - 1;
    }
```
The Lax differencing scheme

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

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

It is easy to see that

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

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-FRIEDRICHLS-LEWY (CFL) stability criterion

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

is satisfied.

```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:

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

```c++
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 / (h * h)) * (u[jPlus1] + u[jMinus1] - 2 * u[j]);
}

**Godunov’s Algorithm**

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

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

and for the cell on the right

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

```cpp
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]);

OpenGL Graphics Code

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();
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluOrtho2D(0, w, h, 0);
}
gluOrtho2D(0, w, 0, h);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
}

void redraw() {
    glutSetWindow(solutionWindow);
    glutPostRedisplay();
}

void display() {
    glClear(GL_COLOR_BUFFER_BIT);
    glutSwapBuffers();
}

void drawText(const string& str, double x, double y) {
    glRasterPos2d(x, y);
    int len = str.find(’\0’);
    for (int i = 0; i < len; i++)
        glutBitmapCharacter(GLUT_BITMAP_HELVETICA_12, str[i]);
}

void displaySolution() {

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++) {
        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();
    ostringstream os;
    os << "CFL Ratio = " << CFLRatio << " nu = " << nu << " t = " << step * tau << " ";
    drawText(os.str(), 0.02, -0.95);
    glutSwapBuffers();
}
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);
    ostringstream os;
    os << methodName[i] << ends;
    drawText(os.str(), (i + 0.2) * w / 4, 0.3 * h);
}
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) {
                glutIdleFunc(NULL);
                running = false;
            } else {
                glutIdleFunc(takeStep);
                running = true;
            }
        }
    }
}
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);
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);
}

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);
ingegrationAlgorithm = Lax;
glutCreateMenu(solutionMenu);
glutAddMenuEntry("Initial Sine Waveform", 1);
glutAddMenuEntry("Initial Step Waveform", 2);
glutAttachMenu(GLUT_RIGHT_BUTTON);

{
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();
}