**The Riemann Problem and Shock Tube Problem**

**Sod’s Shock Tube Problem**

A simple one dimensional model of a gas was introduced by G.A. Sod, *J. Computational Physics* 27, 1 (1978), to test the ability of various algorithms in solving fluid dynamics problems with shock wave behavior.

**The equations of gas dynamics**

The equations of fluid dynamics are mathematical statements of three fundamental physical principles:

- Mass is conserved
- \( F = ma \), i.e., Newton’s second law
- Energy is conserved

The one-dimensional equations for the fluid dynamics of a gas can be written in *conservation form* as follows:

\[
\begin{align*}
\frac{\partial \rho}{\partial t} + \frac{\partial (\rho u)}{\partial x} &= 0 \\
\frac{\partial (\rho u)}{\partial t} + \frac{\partial}{\partial x}(\rho u^2 + p) &= 0 \\
\frac{\partial e}{\partial t} + \frac{\partial}{\partial x}(u(e + p)) &= 0
\end{align*}
\]
where $\rho$ is the density of the fluid, $u$ is the fluid velocity, $e$ is the energy per unit volume (length), and $p$ is the pressure. We need one more equation to close the system. This is the equation of state

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

where $\gamma$ is the adiabatic gas index. For an ideal gas $\gamma = 1.4$.

These equations can be written in vector form

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

where

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

**The shock tube problem**

Sod considered a one-dimensional tube of unit length $0 \leq x \leq 1$ and the following initial conditions at $t = 0$:

$$\rho(x, 0) = \begin{cases} 1.0 & \text{for } x \leq \frac{1}{2} \\ 0.125 & \text{for } x > \frac{1}{2} \end{cases},$$

$$p(x, 0) = \begin{cases} 1.0 & \text{for } x \leq \frac{1}{2} \\ 0.1 & \text{for } x > \frac{1}{2} \end{cases},$$

$$u(x, 0) = 0.$$
This initial state can be produced by having a diaphragm in the middle of the tube. The gas to the left and right of the diaphragm is initially at rest. The pressure and density are discontinuous across the diaphragm. At \( t = 0 \), the diaphragm is broken. Two types of singularities then propagate through the gas:

- **Contact discontinuities**: The pressure \( p \) and velocity \( u \) are continuous, but the density \( \rho \) and energy per unit volume \( e \) are discontinuous.

- **Shock waves**: All quantities \( p, u, \rho \) and \( e \) are in general discontinuous across the shock front.

To simulate a closed tube, reflection boundary conditions can be applied at \( x = 0, 1 \). The shock tube then exhibits interesting behavior with shock waves and contact discontinuities bouncing back and forth in the tube and interacting with one another.

### 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 Riemann problem

The mathematician Bernhard Riemann studied the ideal gas equations in an article "Ueber die Fortpflanzung ebener Luftwellen von endlicher Schwingungsweite" published in 1860, see his Mathematical Papers.

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.

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 $\frac{\partial u}{\partial x}$ 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 \( x = i h, \ t = n \tau \) 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

\[
\partial_x f + \partial_t u = \nabla \cdot \begin{pmatrix} f \\ u \end{pmatrix}
\]

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:

\[
\int \nabla \cdot \begin{pmatrix} f \\ u \end{pmatrix} \, dx \, dt = \oint \begin{pmatrix} f \\ u \end{pmatrix} \cdot \mathbf{n} \, d\ell = 0,
\]

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
\[ u_{i}^{n+1} = \frac{1}{h} \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} u(x, t_{n+1}) \, dx \]
\[ u_{i}^{n} = \frac{1}{h} \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} u(x, t_{n}) \, dx , \]
and the time integral averages of the flux along the left and right sides of the rectangle
\[ f_{i-\frac{1}{2}} = \frac{1}{\tau} \int_{t_{n}}^{t_{n+1}} f(u(x_{i-\frac{1}{2}}, t)) \, dt , \]
\[ f_{i+\frac{1}{2}} = \frac{1}{\tau} \int_{t_{n}}^{t_{n+1}} f(u(x_{i+\frac{1}{2}}, t)) \, dt . \]
The line integral can be written
\[ (u_{i}^{n+1} - u_{i}^{n}) \, dx + \left( f_{i+\frac{1}{2}} - f_{i-\frac{1}{2}} \right) \, dt = 0 . \]
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 $u_{i + 1/2}(x, t)$, then the Godunov flux is taken to be

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

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

$$u_{i + 1/2}(x, t) = \begin{cases} 
  u^n_i & \text{if } c > 0 \\
  u^n_{i+1} & \text{if } c < 0
\end{cases}$$

and hence

$$f_{i + 1/2} = \begin{cases} 
  cu^n_i & \text{if } c > 0 \\
  cu^n_{i+1} & \text{if } c < 0
\end{cases}$$

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

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

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

where the Courant-Friedrichs-Lewy number

$$\lambda_{CFL} = \frac{c \tau}{h}. $$
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 \( u^n_i \) or \( u^n_{i+1} \) in the equations above.

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

\[
\mathbf{U} = \begin{pmatrix} \rho \\ \rho u \\ e \end{pmatrix}, \quad \mathbf{F} = \begin{pmatrix} \rho u \\ \rho u^2 + p \\ u(e + p) \end{pmatrix}.
\]

Instead of a single wave speed, the solution to the Riemann problem involves finding the eigenvalues of a 3 \( \times \) 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.
Code to Solve the 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.

```cpp
#include <cmath>
#include <cstdlib>
#include <iostream>
#include <cstdio>
#include <cstring>
using namespace std;
#include <GL/glut.h>
#include "roe-solver.hpp" // Roe’s Riemann solver for Euler equations
#include "riemann-solver.hpp" // 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
```

```
// Program to solve Sod’s shock tube problem
```
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];
\[ U[N - 1][1] = -U[N - 2][1]; \]
\[ U[N - 1][2] = U[N - 2][2]; \]

```java
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];
        avg += y;
    }
glEnd();
\[ y = U[j][2]; \]
if (plot == plotWindow[3])
    \[ y = (U[j][2] - U[j][1] \ast U[j][1] / U[j][0] / 2) \ast (gama - 1); \]
glVertex2d(j \ast 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 \ast (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:
void displayControl() {
    stepAlgorithm = RoeStep;
    initialize();
    break;
    case 1:
        stepAlgorithm = LaxWendroffStep;
        initialize();
        break;
    case 2:
        stepAlgorithm = upwindGodunovStep;
        initialize();
        break;
    case 3:
        stepAlgorithm = LaxFriedrichsStep;
        initialize();
        break;
    default:
        break;
}
}

void displayControl() {
    glutPostRedisplay();
    redraw();
}
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();
}