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.

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

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

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)

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

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

\[ \frac{\partial D}{\partial t} + \frac{\partial (DV^i)}{\partial x^i} = 0 \]

\[ \frac{\partial E}{\partial t} + \frac{\partial (EV^i)}{\partial x^i} + P \frac{\partial W}{\partial t} + P \frac{\partial (WV^i)}{\partial x^i} = 0 \]

\[ \frac{\partial S_i}{\partial t} + \frac{\partial (SV^i)}{\partial x^i} - \frac{S^\mu S^\nu}{2} \frac{\partial g_{\mu\nu}}{\partial x^i} - \sqrt{-g} \frac{\partial (P)}{\partial x^i} = 0 \]

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

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

$$ds^2 = a^2(\mu,t)c^2 dt^2 - b^2(\mu,t)d\mu^2 - R^2(\mu,t)d\Omega^2 , \quad T^1_1 = T^2_2 = T^3_3 = P , \quad T^0_0 = -\rho(c^2 + \epsilon) ,$$

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.

J.A. Font, "Numerical Hydrodynamics and Magnetohydrodynamics in General Relativity", Living Rev. Rel-
avity 11 (2008), 7 has a comprehensive review.

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

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

where $u$ 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

$$\int dS \cdot \rho u = \int dV \nabla \cdot \rho u ,$$

we obtain the continuity equation

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

Next, consider conservation of momentum

$$\rho \frac{du}{dt} = F ,$$

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 advective term represents the change in fluid velocity due to motion of fluid from neighboring points
in space.

The force density $F$ has three contributions:
external or \textit{body forces} acting on the fluid, for example the force of gravity
\[ F_{\text{gravity}} = \rho g , \]
where \( g \) is the acceleration due to gravity

\textit{pressure forces} due to neighboring fluid elements
\[ - \int p \, dS = - \int \nabla p \, dV , \]
\[ F_{\text{pressure}} = - \nabla p , \]
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
\[ F_{\text{viscous}} = \mu \nabla^2 u + (\mu + \xi) \nabla (\nabla \cdot u) , \]
where \( \mu \) is the \textit{dynamic viscosity coefficient} and \( \xi \) is the \textit{bulk viscosity coefficient} of the fluid.

A special case that is interesting for many applications is that of \textit{incompressible flow}
\[ \rho = \text{constant} , \quad \nabla \cdot u = 0 . \]
Taking these forces into account results in the \textit{Navier-Stokes equations} for incompressible viscous flow:
\[ \frac{\partial u}{\partial t} + uu_x = \frac{1}{\rho} \nabla p + \nu \nabla^2 u , \]
where \( \nu = \mu/\rho \) is the \textit{kinematic viscosity}.

\textbf{One Dimensional Burgers’ Equation}

\[ \frac{\partial u}{\partial t} + uu_x = \nu \frac{\partial^2 u}{\partial x^2} , \]
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 \textit{inviscid} Burgers’ equation
\[ \frac{\partial u}{\partial t} + uu_x = 0 . \]

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 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 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 Codes**

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

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
enum {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:
                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 redraws

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;
    } else {
        clockStart = clockNow;
        done = false;
    }
    redraw();
}

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 \( u^n_j \).

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

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:

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

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-Friedrichs-Lewy (CFL) stability criterion

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

is satisfied.
The resulting algorithm can be expressed as a two-step formula:
\[
    u^*_{j+\frac{1}{2}} = \frac{1}{2} \left( u^n_j + u^n_{j+1} \right) - \frac{\tau}{2h} \left( F^n_{j+1} - F^n_j \right) + \frac{\nu \tau}{2h^2} \left[ \frac{1}{2} \left( u^n_{j+1} + u^n_{j-1} - 2u^n_j \right) + \frac{1}{2} \left( u^n_{j+2} + u^n_{j} - 2u^n_{j+1} \right) \right],
\]
\[
    u^{n+1}_j = u^n_j - \frac{\tau}{h} \left( F^n_{j+\frac{1}{2}} - F^n_{j-\frac{1}{2}} \right) + \frac{\nu \tau}{h^2} \left( u^n_{j+1} + u^n_{j-1} - 2u^n_j \right).
\]

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.,

\[
    u(x, t + \tau) = u(x, t) + \tau \frac{\partial u}{\partial t} + \frac{\tau^2}{2} \frac{\partial^2 u}{\partial t^2} + O(\tau^3).
\]
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

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

where \( F_{j+\frac{1}{2}} \) 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

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

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

\[
    F_{j+\frac{1}{2}} = \max \left\{ \frac{1}{2}(u_{j+1})^2, \frac{1}{2}(u_{j+1})^2 \right\}.
\]

```
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;
```

burgers.cpp
```c
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**

The solutions are visualized using OpenGL graphics.

**Python Code**

The following Python code animates the solutions using Tkinter.

```python
# Burgers equation in one dimension
import math
import time
from tkinter import *
L = 1.0 # size of periodic region
N = 500 # number of grid points
dx = L / float(N) # lattice spacing
t = 0.0 # time
u_max = 1.0 # maximum wave amplitude
CFL_ratio = 1.0 # Courant-Friedrichs-Lewy ratio
dt = CFL_ratio * dx # time step
initial_waveform = "Sine"

nu = 1.0e-6 # kinematic viscosity
u = [] ; u_new = [] # the solution and its update
next = [] ; prev = [] # for periodic boundary conditions

method = "Godunov" # integration method
```
step = 0  # integration step number

def initialize():
    
    # create arrays for lattice vectors
    global u, u_new
    u = [ 0.0 for j in range(N) ]
    u_new = [ 0.0 for j in range(N) ]

    # initialize arrays for periodic boundary conditions
    global next, prev
    next = [ j+1 for j in range(N) ]
    prev = [ j-1 for j in range(N) ]
    next[N-1] = 0
    prev[0] = N-1

    # reset lattice spacing and initialize waveform
    global dx, u_max
    dx = L / float(N)
    u_max = 0.0
    for j in range(N):
        x = j * dx

        if initial_waveform == "Sine":
            u[j] = math.sin(2 * math.pi * x) + 0.5 * math.sin(math.pi * x)
        elif initial_waveform == "Step":
            if x > L/4.0 and x < 3*L/4.0:
                u[j] = 1.0
            else:
                u[j] = 0.0
        else:
            u[j] = 1.0
            u_max = max(abs(u[j]), u_max)

    # set time, step and step number
    global t, dt, step
    t = 0.0
    dt = CFL_ratio * dx / u_max
    step = 0

    def FTCS():
        global u_new
        for j in range(N):
            u_new[j] = u[j] * (1.0 - dt / (2.0 * dx) * (u[next[j]] - u[prev[j]]))
            u_new[j] += nu * dt / dx**2 * (u[next[j]] + u[prev[j]] - 2 * u[j])
def Lax():
    global u_new
    for j in range(N):
        u_new[j] = (u[next[j]] + u[prev[j]]) / 2.0
        u_new[j] -= u[j] * dt / (2.0 * dx) * (u[next[j]] - u[prev[j]])
        u_new[j] += nu * dt / dx**2 * (u[next[j]] + u[prev[j]] - 2 * u[j])

def Lax_Wendroff():
    global u_new
    F = [ u[j]**2 / 2.0 for j in range(N) ] # flux vector
    for j in range(N):
        u_new[j] = (u[j] + u[next[j]]) / 2.0
        u_new[j] -= dt / (2.0 * dx) * (F[next[j]] - F[j])
        u_new[j] += nu * dt / (2.0 * dx**2) * (u[next[j]] + u[prev[j]] - 2 * u[j]) / 2.0 +
                   (u[next[next[j]]] + u[j] - 2 * u[next[j]]) / 2.0
    for j in range(N):
        F[j] = u_new[j]**2 / 2.0
    for j in range(N):
        u_new[j] += nu * dt / dx**2 * (u[next[j]] + u[prev[j]] - 2 * u[j])

def Godunov():
    global u_new
    u_plus = [ u[j] for j in range(N) ]
    u_minus = [ u[j] for j in range(N) ]
    for j in range(N):
        if u_plus[j] < 0.0:
            u_plus[j] = 0.0
        if u_minus[j] > 0.0:
            u_minus[j] = 0.0
    F = [ 0.0 ] * N
    for j in range(N):
        f1 = u_plus[prev[j]]**2 / 2.0
        f2 = u_minus[j]**2 / 2.0
        if f1 > f2:
            F[prev[j]] = f1
        else:
            F[prev[j]] = f2
        f1 = u_plus[j]**2 / 2.0
        f2 = u_minus[next[j]]**2 / 2.0
        if f1 > f2:
            F[j] = f1
    for j in range(N):
        if u_plus[j] < 0.0:
            u_plus[j] = 0.0
        if u_minus[j] > 0.0:
            u_minus[j] = 0.0
    F = [ 0.0 ] * N
    for j in range(N):
        f1 = u_plus[j]**2 / 2.0
        f2 = u_minus[j]**2 / 2.0
        if f1 > f2:
            F[j] = f2

        f1 = u_plus[j]**2 / 2.0
        f2 = u_minus[j]**2 / 2.0
        if f1 > f2:
            F[j] = f1
else:
    F[j] = f2
u_new[j] = u[j]
u_new[j] += nu * dt / dx**2 * (u[next[j]] + u[prev[j]] - 2 * u[j])
u_new[j] -= dt / dx * (F[j] - F[prev[j]])

T = 1.0  # time to travel length L
frames_per_sec = 25  # animation rate for screen redraws

def time_step():
    global t, u, u_new
    eval(method + "()")
    swap = u
    u = u_new
    u_new = swap
    t += dt;

class Application(Frame):
    def create_widgets(self):
        self.controls = Frame(self, relief=RAISED, borderwidth=2)
        self.controls.pack(side=BOTTOM)
        self.waveform = StringVar()
        self.waveform.set(initial_waveform)
        self.INIT = Menubutton(self.controls, text=initial_waveform)
        self.INIT.menu = Menu(self.INIT)
        for wave in ['Sine', 'Step']:
            self.INIT.menu.add_radiobutton(variable=self.waveform, value=wave, label=wave, command=self.reset)
        self.INIT['menu'] = self.INIT.menu
        self.INIT.pack(side=LEFT)
        self.algorithm = StringVar()
        self.algorithm.set(method)
        self.METHOD = Menubutton(self.controls, text=method.center(20))
        self.METHOD.menu = Menu(self.METHOD)
        for algo in ['FTCS', 'Lax', 'Lax_Wendroff', 'Godunov']:
            self.METHOD.menu.add_radiobutton(variable=self.algorithm, value=algo, label=algo)
        self.METHOD['menu'] = self.METHOD.menu
        self.METHOD.pack(side=LEFT)
        self.RESET = Button(self.controls, text="RESET", command=self.reset)
        self.RESET.pack(side=LEFT)
        self.QUIT = Button(self.controls, text="QUIT", command=self.quit)
自定义函数 update_frame:
    def update_frame(self):
        self.draw.delete(ALL)
        for j in range(1, N):
            x0 = int((j-1) * dx / L * 600); x1 = int(j * dx / L * 600)
            y0 = 200 - int(u[j-1] * 180); y1 = 200 - int(u[j] * 180)
            self.draw.create_line(x0, y0, x1, y1, fill="red")

全局方法
    global method
    method = self.algorithm.get()
    self.METHOD.config(text = method.center(20))

开始计时
    start_time = time.clock()

while True:
    time_step()
    if time.clock() - start_time > 1.0 / frames_per_sec:
        break
    self.after(frames_per_sec, self.update_frame)

初始化
    def __init__(self, master=None):
        Frame.__init__(self, master)

设置布局
    Pack.config(self)
    self.create_widgets()
    self.after(10, self.update_frame)

重置波形
    def reset(self):
        global initial_waveform
        initial_waveform = self.waveform.get()
        self.INIT.config(text = initial_waveform)
        initialize()

初始化
    initialize()

启动程序
    root = Tk()
    app = Application(master=root)
    app.mainloop()
    root.destroy()
The Riemann Problem and 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 various algorithms for 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:

\[
\frac{\partial \rho}{\partial t} + \frac{\partial (\rho u)}{\partial x} = 0
\]
\[
\frac{\partial (\rho u)}{\partial t} + \frac{\partial (\rho u^2 + p)}{\partial x} = 0
\]
\[
\frac{\partial e}{\partial t} + \frac{\partial (u(e + p))}{\partial x} = 0
\]

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

\[
\frac{\partial}{\partial t} \mathbf{U} + \frac{\partial}{\partial x} \mathbf{F}(\mathbf{U}) = 0
\]

where

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

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

\[
\begin{align*}
\rho(x, 0) &= \begin{cases} 
1.0 & \text{for } x \leq \frac{1}{2} \\
0.125 & \text{for } x > \frac{1}{2} 
\end{cases}, \\
p(x, 0) &= \begin{cases} 
1.0 & \text{for } x \leq \frac{1}{2} \\
0.1 & \text{for } x > \frac{1}{2} 
\end{cases}, \\
u(x, 0) &= 0.
\end{align*}
\]

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 \( x = ih, \ t = n\tau \) 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
\[ \partial_t f + \partial_x u = \nabla \cdot (f u) \]
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:
\[ \int \nabla \cdot (f u) \, dx \, dt = \oint f \left( \frac{\partial f}{\partial u} \right) \cdot \hat{n} \, d\ell = 0 \, , \]
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
\[ u_{2i+1}^{n+1} = \frac{1}{h} \int_{x_{i+\frac{1}{2}}}^{x_{i+\frac{3}{2}}} u(x,t_{n+1}) \, dx \]
\[ u_{2i}^{n} = \frac{1}{h} \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} u(x,t_{n}) \, dx \, , \]
and the time integral averages of the flux along the left and right sides of the rectangle
\[ f_{j-\frac{1}{2}} = \frac{1}{\tau} \int_{t_{n}}^{t_{n+1}} f(u(x_{j-\frac{1}{2}},t)) \, dt \, , \]
\[ f_{j+\frac{1}{2}} = \frac{1}{\tau} \int_{t_{n}}^{t_{n+1}} f(u(x_{j+\frac{1}{2}},t)) \, dt \, . \]
The line integral can be written
\[ (u_{2i}^{n+1} - u_{2i}^{n}) \, dx + \left( f_{j+\frac{1}{2}} - f_{j-\frac{1}{2}} \right) \, dt = 0 \, . \]
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 \( f_{i+\frac{1}{2}} \).

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 \( x_{i-\frac{1}{2}} < x < x_{i+\frac{1}{2}} \) 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

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

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 \( u_i^n \) 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

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

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 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.

The code was translated from Fortran examples in E.F. Toro, *Riemann Solvers and Numerical Methods for Fluid Dynamics: A Practical Introduction*. Codes are available on Prof. Toro’s web site.
Equilibrium of White Dwarfs and Neutron Stars

These are stars that have exhausted their nuclear fuel. They can be modeled as balls of ideal Fermi gases at temperature $T = 0$ with fermion degeneracy (Pauli exclusion) pressure balancing gravity. In white dwarfs the pressure is generated by degenerate electrons, an in neutron stars by degenerate neutrons.

A first order of magnitude estimate was suggested by Landau. Consider $N$ fermions in a star of radius $R$. The number density $n \sim N/R^3$. A fermion confined to a volume $\sim 1/n$ has momentum $\sim \hbar n^{1/3}$ by Heisenberg’s uncertainty principle. The energy of a fermion is comparable to the Fermi energy $E_F \sim \hbar c n^{1/3} R$.

The gravitational potential energy of a fermion is $E_G \sim -\frac{G M m_B}{R}$, $M = N m_B$ where $m_B$ is the baryon mass (per electron for white dwarfs or the neutron mass for neutron stars).

In static equilibrium at $T = 0$, the total energy is a minimum $E = E_F + E_G = \frac{h c N^{1/3}}{R} - \frac{G N m_B^2}{R}$. Note that both terms scale like $1/R$ so the equilibrium is determined by the sign of $E$.

- If $N$ is small enough $E > 0$ and is lowered by increasing $R$. This causes $E_F$ to decrease. The fermions will become non-relativistic
  
  $E_F \sim \frac{p_F^2}{2m} \sim \frac{\hbar^2 N^{2/3}}{2m R^2}$

  and there will be a stable minimum at
  
  $\frac{dE}{dR} \sim -\frac{\hbar^2 N^{2/3}}{R^3} + \frac{G N m_B^2}{2 R^2} = 0$

- If $N$ is large enough the total energy can become negative $E < 0$ and decreases without bound as gravity causes the star to collapse.

These two limiting cases imply that there is a maximum baryon number for equilibrium given by

$E = 0$, $N_{\text{max}} \sim \left(\frac{h c}{G m_B^2}\right)^{3/2} \sim 2 \times 10^{57}$, $M_{\text{max}} \sim N_{\text{max}} m_B \sim 1.5 M_\odot$.

- If $N = N_{\text{max}}$ and the fermions are relativistic $E_F \geq m c^2$

  $R \leq \frac{\hbar}{m c} \left(\frac{h c}{G m_B^2}\right)^{1/2} \sim \begin{cases} 5 \times 10^3 \text{ km}, & m = m_e \text{ (white dwarfs)} \\ 3 \text{ km}, & m = m_n \text{ (neutron stars)} \end{cases}$

This is essentially the physics of the Chandrasekhar and Tolman-Oppenheimer-Volkoff limits.
White Dwarfs

The theory of white dwarfs was first developed by S. Chandrasekhar Mon. Not. Roy. Astron. Soc. 95, 207-225 (1935) based on modeling the electrons as a $T = 0$ ideal Fermi gas

$$E_F = \sqrt{p_F^2 c^2 + m_e^2 c^4}, \quad n_e = \frac{2}{\hbar^3} \int_0^{p_F} dp 4\pi p^2 = \frac{8\pi}{3\hbar^3} p_F^3 = \frac{1}{3\pi^2 \lambda_e^3} x^3, \quad x \equiv \frac{p_F}{m_e c}$$

where $\lambda_e = \hbar/(m_e c)$ is the Compton wavelength of the electron. The pressure and energy density of the gas are given by

$$P_e = \frac{8\pi m_e^4 c^5}{3\hbar^3} \int_0^x dx \frac{x^4}{\sqrt{1 + x^2}}, \quad \varepsilon_e = \frac{8\pi m_e^4 c^5}{3\hbar^3} \int_0^x dx x^2 \sqrt{1 + x^2}.$$

More realistic results take into account Coulomb corrections, and various nuclear effects.

Neutron Stars

The physics of white dwarfs is essentially determined by the properties of a single elementary particle, the electron. The physics of neutron stars is determined by neutrons, protons and many of the stable atomic nuclei, and the strong and weak interactions play an essential role in addition to degeneracy pressure and Coulomb forces.

Neutron star models suggest a structure similar to a planet, with a solid crust, an intermediate mantle, and a liquid core:

- There is an outer envelope which is probably a lattice of Fe ions with an electron gas.
- The outer crust is a Coulomb lattice of electrons and neutron-rich nuclei.
- The inner crust consists of neutrons, nuclear clusters and electrons.
- A mantle separates the solid crust from the liquid core.
- The central core is likely a liquid of neutrons, protons and electrons.
- In a sufficiently massive star a quark-gluon liquid may develop at the center of the core.
The Tolman-Oppenheimer-Volkoff Equilibrium Equations

Consider a spherical star described by a perfect fluid in equilibrium with its gravitational field. The Tolman-Oppenheimer-Volkoff equations using the notation of Baym, Pethick and Sutherland Astrophys. J. 170, 299-317 (1971) is

$$\frac{dP(r)}{dr} = -\frac{G}{r^2} \left[ \rho(r) + \frac{P(r)}{c^2} \right] \left[ m(r) + \frac{4\pi r^3 P(r)}{c^2} \right] \left[ 1 - \frac{2Gm(r)}{rc^2} \right]^{-1}$$

where the $P(r)$ and $\rho(r)$ are the pressure and density at radial coordinate $r$, and

$$m(r) = \int_0^r d^3 r' \rho(r')$$

is mass contained within a sphere of radius $r$. The radius of the star is the radial coordinate value at which the pressure $P(R) = 0$, and total mass of the star is

$$M = m(R) = \int d^3 r \rho(r) .$$

The number of baryons within radius $r$ is

$$b(r) = \int_0^r d^3 r' \frac{n_b(r')}{\sqrt{1 - 2Gm(r')/rc^2}}$$

where $n_b(r)$ is the baryon number density, and the total number of baryons is

$$B = b(R) .$$
To determine the equilibrium configuration of the star, the TOV equation must be integrated from \( r = 0 \) to \( r = R \) and matched onto the external Schwarzschild solution.

Note that there are two unknown functions in the equation, the pressure \( P(r) \) and the density \( \rho(r) \). To close the system, an equation of state (EOS) for nuclear matter must be specified.

Baym, Pethick and Sutherland Astrophys. J. 170, 299-317 (1971) developed an EOS applicable to nuclear matter and obtained the results shown in the Figure:

![Graph showing mass versus central density for zero-temperature nonrotating stars in nuclear equilibrium.](image)

**Fig. 1.**—Mass versus central density for zero-temperature nonrotating stars in nuclear equilibrium. Stars to the left of the maximum (the Chandrasekhar limit) at \( \rho_c = 1.4 \times 10^9 \) g cm\(^{-3} \) are stable white dwarfs, while stars to the right of the minimum at \( 1.55 \times 10^{14} \) g cm\(^{-3} \) are neutron stars. The lower dashed extension of the curve was constructed from Pandharipande's hyperonic equation of state C, and the upper dashed extension from Pandharipande's equation of state for pure neutrons. Neutron stars beyond the maximum are unstable.
Equation of State for Nuclear Matter

The general relativistic adiabatic index (ratio of specific heats) is

\[
\Gamma = \frac{n_b \frac{\partial P}{\partial n_b}}{P \frac{\partial P}{\partial \rho}} = \frac{\rho + P/c^2 \frac{\partial P}{\partial \rho}}{P}.
\]

Figure 2.3. The adiabatic index \( \Gamma = (\partial \ln P)/(\partial \ln \rho) \) as a function of \( \rho \) for the equations of state shown in Figure 2.2.
Equations of State and Solving the TOV Equation

A white dwarf or neutron star in equilibrium has exhausted all its nuclear fuel. Gravitational collapse is prevented by fermion degeneracy pressure. The thermal energy per fermion $k_B T$ is much smaller than the Fermi energy $E_F$ and can be set to zero. The star is then in its quantum ground state. An equation of state relates pressure, density, and temperature. At $T = 0$ the EOS relates pressure to density

$$P = P(\rho) .$$

This relation can be computed from the electromagnetic and nuclear forces of the electrons, ions, protons, neutrons, and other subatomic particles in the star.

Baym, Pethick and Sutherland Astrophys. J. 170, 299-317 (1971) provide equations of state appropriate for various density ranges in the crust, mantle and core of a degenerate star.

Feynman-Metropolis-Teller EOS

<table>
<thead>
<tr>
<th>TABLE 5</th>
<th>EQUATION OF STATE OF MATTER IN ITS GROUND STATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\rho$ (g cm$^{-3}$)</td>
<td>$P$ (dyn cm$^{-2}$)</td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
</tr>
<tr>
<td>Feynman-Metropolis-Teller</td>
<td></td>
</tr>
<tr>
<td>7.80</td>
<td>$\leq 1.01129$</td>
</tr>
<tr>
<td>7.90</td>
<td>1.01210</td>
</tr>
<tr>
<td>8.15</td>
<td>1.01211</td>
</tr>
<tr>
<td>8.60</td>
<td>1.21E12</td>
</tr>
<tr>
<td>16.4</td>
<td>1.40913</td>
</tr>
<tr>
<td>45.1</td>
<td>1.70E24</td>
</tr>
<tr>
<td>212</td>
<td>5.82E25</td>
</tr>
<tr>
<td>1130</td>
<td>1.90E17</td>
</tr>
<tr>
<td>Present Calculation</td>
<td></td>
</tr>
<tr>
<td>1.044E5</td>
<td>9.74E18</td>
</tr>
<tr>
<td>2.022E5</td>
<td>4.96E19</td>
</tr>
</tbody>
</table>

For densities $\rho \leq 1.0 \times 10^4$ gm/cm$^3$ they use the EOS of Feynman, Metropolis and Teller Phys. Rev. 75, 1561-1573 (1949) based on the Thomas-Fermi model of an electron gas is a crystalline lattice of ions. This model was developed for the electronic structure of metals. The outermost crust of a degenerate star is well approximated as a crystal of the most stable nuclide $^{56}$Fe$^{26}$ with atomic number $Z = 26$ and mass number $A = 56$. 
Baym-Pethick-Sutherland EOS

As the density increases deeper in the star two important competing effects change the composition of the $^{56}\text{Fe}$ lattice:

- The electrons become relativistic and are forced into the nuclei where they combine with protons by inverse $\beta$-decay
  $$e^- + p \rightarrow n + \nu_e$$
  and the Fe nuclei are transformed into isotopes with more neutrons. Neutrons are prevented from decaying back via
  $$n \rightarrow p + e^- + \bar{\nu}_e$$
  because there are no empty free electron states in the filled Fermi sea of electrons.

- The neutron-rich isotopes will undergo nuclear fission if the fission fragments have a lower energy. The fission fragments will then recombine and fission until the system comes to equilibrium again at with some fixed number density $n_N$ of nuclei with $Z$ protons, $A-Z$ neutrons, and $Z$ electrons per nucleon in the electron gas.

The BPS EOS is based on a Semi-empirical mass formula

$$E_{\text{tot}}(A, Z, n_N) = n_N(W_N + W_L) + E_e(Zn_n)$$

where $W_N$ is the rest energy of an isolated nucleus determined by experimental nuclear physics data, $W_L$ is the lattice energy of a Body-centered-cubic lattice of nucleons with lattice constant determined by $n_Na^3 = 2$, and $E_e$ is the energy of the electron gas.

They find that the composition of the outer crust changes from $^{56}\text{Fe}_{26}$ at $\rho = 1.0 \times 10^4$ gm/cm$^3$, through a sequence of heavier stable nuclei to $^{118}\text{Kr}_{36}$ at $\rho = 4.0 \times 10^{11}$ gm/cm$^3$. Note that the heaviest stable isotope of Krypton on Earth is $^{86}\text{Kr}_{36}$, so the Krypton layer in degenerate star crusts has an excess of 32 additional neutrons!

Baym-Bethe-Pethick EOS above Neutron Drip

As the density increases beyond $\rho = 1.0 \times 10^4$ gm/cm$^3$, neutrons begin to leak out of the neutron-rich nuclei. This is called Neutron drip.

The nuclear matter changes above neutron from a pure metal to a two-phase system consisting of the metal lattice and free neutrons. The equilibrium composition of this system consisting of a metal lattice and free electron and neutron gases was done by Baym, Bethe and Pethick Nucl. Phys. A 175, 225-271 (1971).

The BBP EOS is also based on a semi-empirical mass formula for the metal lattice, and takes into account the following effects:

- The interaction energy of the free neutrons in the neutron gas with the neutrons and protons at the surface of the lattice nucleons.
- The reduction in the surface energy terms of the semi-empirical mass formula due to contact of the nuclei with the surrounding neutron fluid.
- The contribution of strong nuclear forces to the nuclear lattice Coulomb energy.

The resulting BBP EOS predicts a series of increases in $Z$ and $A$ until the nuclei are all converted to free neutrons for densities $\rho \geq 1.0 \times 10^{14}$ gm/cm$^3$.

Hyperonic Matter EOS and Pure Neutron Matter EOS

At higher densities it can become energetically favorable to create excited states of nucleons such as the Delta baryon and Hyperons with non-zero strangeness quantum numbers.

Baym et al. provide two alternative EOSs due to V.R. Pandharipande, Nucl. Phys. A 178, 123-144 (1971), applicable to this density regime:

<table>
<thead>
<tr>
<th>State C</th>
<th>$\rho$</th>
<th>$P$</th>
<th>$m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hyperonic Matter</td>
<td>6.908E14</td>
<td>2.720E34</td>
<td>4.000E38</td>
</tr>
<tr>
<td>Pure Neutron Matter</td>
<td>7.004E14</td>
<td>3.235E34</td>
<td>4.000E38</td>
</tr>
</tbody>
</table>

Solving the Tolman-Oppenheimer-Volkoff Equation

The equilibrium of a spherically symmetric star consisting of a perfect fluid in equilibrium with its gravitational field is determined by the Tolman-Oppenheimer-Volkoff equations

$$\frac{dP(r)}{dr} = -\frac{G}{r^2} \left[ \rho(r) + \frac{P(r)}{c^2} \right] \left[ m(r) + \frac{4\pi r^3 P(r)}{c^2} \right] \left[ 1 - \frac{2Gm(r)}{rc^2} \right]^{-1}$$
where the $P(r)$ and $\rho(r)$ are the pressure and density at radial coordinate $r$, and

$$m(r) = \int_0^r d^3r' \rho(r')$$

is the mass contained within a sphere of radius $r$.

**Geometrized Units**

It is convenient in astrophysical and cosmological calculations to use a Geometrized unit system to avoid arithmetic overflow or underflow in combining very large and small numerical values.

For example, we can set

$$h = c = G = k_B = \frac{1}{4\pi\epsilon_0} = 1$$

by choosing appropriate units of mass, length, time, temperature, and electric charge. In relativity and cosmology all physical quantities are expressed in terms of a single unit of length (geometry). For example, time is measured in meters and $1 \text{s} = 2.99 \times 10^8 \text{m}$ and the mass of the Sun is $M_\odot = 1.4766 \text{km}$. SI units can always be recovered from the dimensions of the quantity and appropriate conversion factors.

In high energy physics, quantities are expressed in terms of energy and the system is termed “natural units”.

**General Relativistic Equation for the Metric**

It can be shown that the most spherically symmetric metric can be written in terms of two metric functions $\Phi(r, t)$ and $\lambda(r, t)$ independent of the angular coordinates $\Omega = (\theta, \phi)$:

$$ds^2 = -e^{2\Phi} dt^2 + e^{2\lambda} dr^2 + r^2 d\Omega^2 ,$$

For an isolated star, the density $\rho(r)$ is zero outside a finite radius $r = R$ and the metric must reduce to the Schwarzschild metric

$$ds^2 = -\left(1 - \frac{2M}{r}\right) dt^2 + \left(1 - \frac{2M}{r}\right)^{-1} dr^2 + r^2 d\Omega^2 ,$$

where $M$ is the total mass inside of the star.

According to Birkhoff’s theorem this is actually also true for any spherically symmetric time-dependent density inside of the star!

Einstein’s equation

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

for spherically symmetric stars can now be written in terms of the metric functions $\Phi, \lambda$. For stellar applications the cosmological constant $\Lambda$ just contributes a constant background energy and can be dropped. In addition, for a static solution the metric functions do not depend on the time $t$.

General covariance still leaves some freedom in choosing the form of the metric functions. A convenient
choice is to relate them to the mass within a radius $r$ with the definition
\[ e^{2\lambda(r)} = \frac{1}{\left(1 - \frac{2m(r)}{r}\right)}. \]

This is possible because the coordinate $r$ is not the invariant distance measured by a meter stick from the center of the star. The invariant distance is determined by integrating $ds$ once the metric is known.

**Equilibrium Equations to be Solved**

Assuming zero temperature $T = 0$, the EOS gives the pressure $P(\rho)$ and a function of the density $\rho(r)$ at radial coordinate $r$. Actually, $P$ and $\rho$ are not the pressure and density measured by an external observer, but the proper pressure and density in an inertial (freely falling) coordinate system at the point $r$.

The TOV equilibrium equation is
\[
\frac{dP}{dr} = -\frac{\rho m}{r^2} \left(1 + \frac{P}{\rho}\right) \left(1 + \frac{4\pi Pr^3}{m}\right) \left(1 - \frac{2m}{r}\right)^{-1}.
\]

The mass inside radius $r$ is determined by the equation
\[
\frac{dm}{dr} = 4\pi r^2 \rho.
\]

This is a system of 2 ordinary differential equations for 3 unknown functions with the EOS relating $P$ and $\rho$.

They can be solved by integrating from $r = 0$ outward with starting values
\[
\rho(0), \quad P(0) = P(\rho(0)), \quad m(0) = 0,
\]
and integrating outward using the Runge-Kutta algorithm until the pressure drops to zero
\[
P(R) = 0, \quad m(R) = M,
\]
to obtain the mass of the star $M(\rho(0))$ as a function of the central density. This integration yields the pressure profile $P(r)$, and the corresponding density profile $\rho(r)$ for given central density.

The metric inside the star must match smoothly onto the Schwarzschild metric at the surface. The equation for the metric function is
\[
\frac{d\Phi}{dr} = -\frac{1}{\rho} \frac{dP}{dr} \left(1 + \frac{P}{\rho}\right)^{-1}.
\]

This can be integrated with starting condition
\[
\Phi(r = R) = \frac{1}{2} \ln \left(1 - \frac{2M}{R}\right),
\]
at the surface of the star, down to the center of the star.
Baym et al. obtained the results shown in Figures 1 and 2 above for non-rotating stars. The solid lines are obtained using the FMT, BPS and BBP EOSs, and the dashed line extension using the Pandharipande EOSs for neutron matter (upper curve) or hyperon matter (lower curve).

These results can be reproduced using adaptive Runge-Kutta routines to integrate the TOV equation. The EOS $P(\rho)$ can be obtained from the numerical tables in Baym et al. using an interpolating routine.
General-Relativistic Gravitational Collapse

One of first detailed numerical calculations of the collapse of a spherically symmetric massive star was done by M.M. May and R.H. White, Phys. Rev. 141, 1232-1241 (1966).

Collapse Equations for an Ideal Fluid

The equations solved by May and White were derived in a very clear article by C.W. Misner and D.H. Sharp, Phys. Rev. 136, B571-B576 (1964)

The general-relativistic equations to be solved are

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

where the metric tensor \( g^{\mu\nu} \) is defined by the spherically symmetric form

\[ ds^2 = -e^{2\phi} dt^2 + e^\lambda dr^2 + R^2 d\Omega^2, \quad d\omega^2 = d\theta^2 + \sin^2 \theta d\phi^2, \]

where the metric functions \( \phi(r, t) \), \( \lambda(r, t) \) and \( R(r, t) \) are independent of \( \Omega = (\theta, \varphi) \).

Ideal Fluid Stress-Energy Tensor

The initial state of the star is assumed to be a spherically symmetric ideal fluid distribution specified by four-velocity \( u^\mu \), internal energy density \( \epsilon \) and pressure \( p \) with stress energy tensor

\[ T^{\mu\nu} = (p + \epsilon) u^\mu u^\nu + pg^{\mu\nu}. \]

- In non-relativistic thermodynamics \( p(r, t) \) and \( \epsilon(r, t) \) are scalars. In general relativity stresses (pressure, shear) are tensor components, and energy is the time component of a four-vector. In this application
the quantities $p$ and $\epsilon$ are defined at each spacetime point $x^\mu$ to be the values of pressure and energy density in the rest frame of the fluid at that point; these local rest-frame quantities are scalar functions.

- The energy density $\epsilon$ is defined as the internal energy of the fluid per unit proper rest volume, i.e., the energy per unit volume of the fluid in the local rest frame.

- In non-relativistic mechanics mass is conserved. In general relativistic collapse baryon number is conserved. The scalar baryon number density $n(r, t)$ is defined as the number of baryons per unit proper rest volume, and the specific volume $v(r, t) = 1/n(r, t)$.

### Comoving Coordinate System and Metric Functions

The GR equations are form-invariant under differential local coordinate transformations. This can be used to define a coordinate system comoving with the fluid. In this reference system the fluid is instantaneously at rest with four-velocity

$$u^i = \frac{dt}{d\tau} = e^{-\phi}, \quad u^i = 0, \quad i = r, \theta, \varphi.$$  

The equations of motion can be simplified by defining

$$U = D_t R \equiv u^\mu \frac{\partial R(r, t)}{\partial x^\mu} = e^{-\phi} \dot{R},$$

where $D_t$ is the comoving proper time derivative, so that $U d\theta$ gives the relative speed of two adjacent fluid elements at the same radial coordinate $r$ and angular separation $d\theta$ as they move radially in or out. The function $\phi(r, t)$ in the metric is replaced by

$$e^{2\phi} = \left( \frac{\dot{R}}{U} \right)^2.$$  

In the static case the mass $m(r)$ within a radius $r$ was used to replace the radial metric function

$$g_{rr} = e^{\lambda(r)} = \left( 1 - \frac{2m(r)}{r} \right)^{-1}.$$  

The appropriate generalization to the time-dependent case is

$$e^{\lambda(r, t)} = \left[ 1 + U^2 - \frac{2m(r, t)}{R} \right]^{-1} \left( \frac{\partial R}{\partial r} \right)^2,$$

which reduces to the static case where $R(r, t) = r$ and $U = 0$.

### Equation of State

As in the static case, it is necessary to specify the relation between the pressure $p(r, t)$ and energy density $\epsilon(r, t)$ on the right hand side of the general relativistic equations of motion.

The simplest possible approximation is an ideal gas of baryons in thermal equilibrium at temperature $T = 0$ and the flow is adiabatic, i.e., the energy of a fluid element does not change as it moves radially in or out.
The internal energy density is determined by the baryon number density, given the adiabatic equation of state
\[ \epsilon = \epsilon(n), \]
and the pressure is then determined by the thermodynamic relation
\[ p = n \left( \frac{\partial \epsilon}{\partial n} \right)_s - \epsilon, \]
where \( s \) is the specific entropy.

The specific internal energy \( \epsilon/n \) and pressure \( p \) of unit amount of fluid with specific volume \( v = 1/n \) containing a mole of baryons determine the specific **Enthalpy**
\[ h = \frac{\epsilon}{n} + pv = \frac{\epsilon + p}{n} = \left( \frac{\partial \epsilon}{\partial n} \right)_s. \]

**Conservation Equations**

The equations of fluid dynamics are based on various conservation laws. The stress-energy tensor is conserved
\[ T^{\mu\nu}_\mu = 0, \]
which relates the metric function \( \phi(r, t) \) to the specific enthalpy
\[ e^\phi = (-g_{00})^{1/2} = \frac{1}{h}. \]

**Einstein’s Equations**

Einstein’s equations can be simplified using the functions defined above. The derivation is done in detail by Misner and Sharp. There are three dynamical equations
\[ D_t R = U, \]
\[ D_t m = -4\pi R^2 p U, \]
\[ D_t U = - \left[ \frac{1 + U^2 - 2m/R}{\epsilon + p} \right] \left( \frac{\partial p}{\partial R} \right)_t - \frac{m + 4\pi R^3 p}{R^2}, \]
and one kinematical equation
\[ \left( \frac{\partial m}{\partial R} \right)_t = 4\pi R^2 \epsilon, \]
in addition to the equation of state and the two conservation equations.
Boundary Conditions and Initial Conditions

To solve these equations numerically, the initial values of the functions $R(r, 0)$, $m(r, 0)$ and $U(r, 0)$ are specified arbitrarily.

A unique solution is obtained for this hyperbolic system boundary conditions at $r = 0$ and the surface of the star at $r_s$ defined by

$$p = 0 \quad \text{at} \quad r = r_s = \text{constant},$$

which determines the mass of the star

$$M = m(r_s, t)$$

and the exterior Schwarzschild metric

$$ds^2 = - \left(1 - \frac{2M}{r}\right) dt^2 + \left(1 - \frac{2M}{r}\right)^{-1} dr^2 + r^2 d\Omega^2.$$

The boundary conditions at the center of the star are

$$R(0, t) = m(0, t) = U(0, t) = 0.$$

Finite Difference Equations to be Solved

May and White (Phys. Rev. 141, 1232-1241 (1966)) solved these equations numerically for $M = 2.1 M_\odot$, $21 M_\odot$ and $210 M_\odot$ using an adiabatic equation of state $P = 2\rho \epsilon / 3$ corresponding to an adiabatic index $\gamma = 5/3$. The initial conditions were taken to uniform inside the star.

Their equations were improved by K.A. van Riper, "General relativistic hydrodynamics and the adiabatic collapse of stellar cores", Astrophys. J. 232, 558-571 (1979) who gives explicit formulas convenient for coding in an appendix.
Cosmological Evolution and Structure Formation

New Simulation Published This Week!


“Our simulation code, AREPO, uses an unstructured Voronoi tessellation of the simulation volume, where the mesh-generating points of this tessellation are moved with the gas flow. The adaptive mesh is used to solve the equations of ideal hydrodynamics with a finite volume approach using a second-order unsplit Godunov scheme with an exact Riemann solver . . . The gravity calculation employs a Tree-PM scheme, where long-range forces are determined with a particle-mesh method (PM) while short-range forces are computed via a hierarchical tree algorithm.”

The simulation is based on the $\Lambda$CDM cosmological model, and produces galaxy morphologies and abundances that closely match observations:
See Moving Mesh Cosmology for more information.

The Millenium Simulation

The Millenium Simulation was the previous largest N-Body Simulation with over 10 billion particles. The simulation was carried out by the Virgo Consortium using a cluster of 512 processors located at the Max Planck Institute for Astrophysics in Garching, Germany. The simulations took a total of 28 days (∼ 600 hours) of wall clock time, and thus consumed around 343,000 hours worth of cpu-time. Results published in V. Springel et al., Nature 435, 629-636 (2005).

Initial Conditions and Cosmic Microwave Background

“...The Millennium Simulation was carried out with a customized version of the GADGET2 code, using the ‘TreePM’ method for evaluating gravitational forces. Initial conditions were laid down by perturbing a homogeneous, ‘glass-like’, particle distribution with a realization of a gaussian random field with the ΛCDM model linear power spectrum as given by the Boltzmann code CMBFAST. The simulation started at redshift $z = 127$ and was evolved to the present using a leapfrog integration scheme with individual and adaptive timesteps, with up to 11,000 timesteps for individual particles.”
The anisotropies of the Cosmic microwave background (CMB) as observed by Planck. The CMB is a snapshot of the oldest light in our Universe, imprinted on the sky when the Universe was just 380,000 years old. It shows tiny temperature fluctuations that correspond to regions of slightly different densities, representing the seeds of all future structure: the stars and galaxies of today. From ESA Planck Collaboration.

Model CMB anisotropies can be generated by the Fortran codes CMBFAST or its successor CAMB. There is also a C++ code CMBEASY, which is easier to understand and use.

The $N$-Body problem

Consider a set of $N$ particles (bodies) which interact with one another by long range forces which decrease inversely as the square of the distance. The gravitational interaction between masses and the electromagnetic interaction between charges are examples of long range forces.

This $N$-body problem is one of the oldest problems in physics. After Newton solved the 2-body problem exactly, numerous scientists and mathematicians attempted to find exact solutions to the 3-body problem. So far, no non-trivial exact solutions to the 3-body problem have been found. Mathematicians have been able to prove that the problem is non-integrable, and that many 3-body trajectories are chaotic and cannot be computed numerically.

Why are $1/r^2$ forces called long range? Suppose a very large number of particles are distributed roughly uniformly over a large region of space. The area of a sphere of radius $r$ surrounding any particle increases like $4\pi r^2$. Thus the number of particles at distance $r$ times the strength of the force exerted by each of these particles is roughly independent of $r$. Thus all of the particles in the system influence the motion of any one particle.

Because the forces are long range, it is not possible to solve for the dynamics of any particle by considering a local neighborhood, as was done in speeding up the Lennard-Jones MD simulation by introducing a cut-off radius beyond which the force could be neglected.

\[
|F| \times 4\pi r^2 \rho(r) \, dr \sim \frac{1}{r^2} \times r^2 \times \frac{N}{V} \, dr \sim \frac{N}{r^5 V}
\]

Molecular Dynamics

\[
|F| \times 4\pi r^2 \rho(r) \, dr \sim \frac{1}{r^2} \times r^2 \times \frac{N}{V} \, dr \sim \frac{N}{V}
\]

Gravitational $N$-body Problem
Numerical solution of the $N$-body problem

Several approaches to speeding up $N$-body codes have been developed. The following are some examples of widely used methods:

**Particle-mesh methods:** These are based on introducing a uniform cubic lattice of points in space. The effect of each particle in a cube of the lattice is approximated by variables located at the neighboring grid points. This grid of variables is used to compute the potential which determines the force on the particles. These methods have been very successful in solving problems in which the particles are spread out roughly uniformly over all of space. These methods typically scale like $O(M \log M)$ where $M$ is the number of grid points.

**Tree-code methods:** These methods were developed to simulate astrophysical systems such as the motion of stars in a galaxy. The distribution of stars is generally highly non-uniform: particle-mesh methods do not work well for non-uniform distributions. Tree-code methods partition space hierarchically into a tree like structure of cubic regions: where there are few particles, the tree contains only a few large cubes; but in regions with many particles, the cubes are repeatedly sub-divided into smaller cubes. Tree-code methods typically scale like $O(N \log N)$.

**Fast multipole methods:** These methods use multipole expansions of the long range potential. A finite number of terms in the expansions are retained, and the algorithm then allows an accurate approximate determination of the forces which scales like $O(N)$. The Fast Multipole Algorithm was named one of the top ten algorithms of the twentieth century by the magazine *Computers in Science and Engineering*.

Applications of $N$-body methods

There are many areas in which $N$-body methods are used, for example:

**Astrophysics:** The dynamical evolution of stars within galaxies, and collisions between galaxies involve thousands of bodies all interacting through long range forces. Many of the $N$-body algorithms were developed to solve astrophysical problems.

**Plasma physics:** In a plasma, atoms are ionized into electons and positively charged ions which interact through long range Coulomb forces. Particle mesh methods were developed to solve the dynamics of uniform plasmas.

**Molecular dynamics:** Interactions between rare gas atoms such as Argon fall off like $1/r^6$. However, in materials made up of polar molecules (which have a permanent dipole moment), the Coulomb field falls off like $1/r^3$: while this is not strictly a long range force, the number of interacting particles which need to be taken into account can be very large, and $N$-body methods then become very useful. Such methods can also be useful is studying an electron gas confined in solid state devices.

**Fluid dynamics:** It can be shown that elliptic partial differential equations with Dirichlet boundary conditions can be solved using $N$-body techniques. In this approach, the solution is determined by a finite number of discrete sources on the boundary of the region.
Hut-Makino starter code for $N$-body simulations

Astrophysics is an interesting area in which to learn about $N$-body methods. Piet Hut and Jun Makino have a website with a nice tutorial program for doing $N$-body simulations. It is good to learn how to locate public domain programs, to understand how they work, and to use them to solve computational problems. One can learn many useful programming techniques by studying good code written by others.

The key functions of the code are explained briefly below: the full code and documentation are available at An Introduction to the N-Body Problem with computer simulation codes in C++.

The main function

The starter code solves Newton’s equations of motion for $N$ bodies, each of which can have a different mass, given initial positions and velocities. The bodies interact through Newton’s inverse square law of gravity.

typedef double real; // "real" as a general name for the standard floating-point data type
const int NDIM = 3; // number of spatial dimensions

int main(int argc, char *argv[])
{
    real dt_param = 0.03; // control parameter to determine time step size
    real dt_dia = 1; // time interval between diagnostics output
    real dt_out = 1; // time interval between output of snapshots
    real dt_tot = 10; // duration of the integration
    bool init_out = false; // if true: snapshot output with start at t = 0 with an echo of the input snapshot
    bool x_flag = false; // if true: extra debugging diagnostics output

    if (! read_options(argc, argv, dt_param, dt_dia, dt_out, dt_tot, init_out, x_flag))
        return 1; // halt criterion detected by read_options()

    int n; // N, number of particles in the N-body system
    cin >> n;

    real t; // time
    cin >> t;

    real * mass = new real[n]; // masses for all particles
    real (* pos)[NDIM] = new real[n][NDIM]; // positions for all particles
    real (* vel)[NDIM] = new real[n][NDIM]; // velocities for all particles
The `get_snapshot` function reads the values of the masses of the bodies and the initial positions and velocities. The `evolve` function integrates Newton's equations of motion for the desired number of time steps.

**Fourth-order Hermite integration algorithm**

The program uses an integration algorithm that the authors have found works well for various astrophysical problems.

First some definitions: The vector distance between particles $i$ and $j$ is defined to be

$$ r_{ji} = r_j - r_i , $$

and the relative velocity of the two particles is

$$ v_{ji} = v_j - v_i , $$

According to Newton's law of gravity, the acceleration of particle $i$ due to particle $j$ is

$$ a_{ji} = \frac{M_j}{r_{ji}^3} r_{ji} . $$

Here we use units such that Newton's constant $G = 1$. Note that the acceleration is directed towards particle $j$: gravity is an attractive force. The algorithm also makes use of the rate of change of the acceleration, which is called the Jerk

$$ j_{ji} = \frac{M_j}{r_{ji}^3} \left[ v_{ji} - 3 \frac{v_{ji} \cdot r_{ji}}{r_{ji}} r_{ji} \right] . $$

The acceleration and jerk of particle $i$ are then given by

$$ a_i = \sum_{j \neq i} a_{ji} , \quad j_i = \sum_{j \neq i} j_{ji} . $$

The Hermite algorithm used by Hut and Makino is a type of "predictor-corrector" algorithm. During a time step of size $\delta t$, the next position and velocity of the particle are “predicted” using the known acceleration
and jerk:

\[
\begin{align*}
\mathbf{r}_p &= \mathbf{r} + \mathbf{v}\delta t + \frac{1}{2}\mathbf{a}\delta t^2 + \frac{1}{6}\mathbf{j}\delta t^3 \\
\mathbf{v}_p &= \mathbf{v} + \mathbf{a}\delta t + \frac{1}{2}\mathbf{j}\delta t^2
\end{align*}
\]

These predicted positions and velocities are used to compute the predicted accelerations \(\mathbf{a}_p\) and jerks \(\mathbf{j}_p\). By making Taylor series expansions of the various formulas, it can be shown that the next two derivatives of the acceleration are given by

\[
\begin{align*}
\mathbf{k} &\equiv \frac{1}{2}\mathbf{a}^{\prime\prime}\delta t^2 = 2(\mathbf{a} - \mathbf{a}_p) + \delta t(\mathbf{j} - \mathbf{j}_p) \\
\mathbf{l} &\equiv \frac{1}{2}\mathbf{a}^{\prime\prime\prime}\delta t^3 = -3(\mathbf{a} - \mathbf{a}_p) - \delta t(2\mathbf{j} + \mathbf{j}_p)
\end{align*}
\]

This information is then used to get the “corrected” positions and velocities at the next time step:

\[
\begin{align*}
\mathbf{r}_c &= \mathbf{r}_p + \left(\frac{1}{12}\mathbf{k} + \frac{1}{20}\mathbf{l}\right)\delta t^2 \\
\mathbf{v}_c &= \mathbf{v}_p + \left(\frac{1}{3}\mathbf{k} + \frac{1}{4}\mathbf{l}\right)\delta t
\end{align*}
\]

This algorithm is implemented in the program as follows:

**Taking a single time step \(\delta t\)**

```c
void evolve_step(const real mass[], real pos[][NDIM], real vel[][NDIM],
real acc[][NDIM], real jerk[][NDIM], int n, real & t,
real dt, real & epot, real & coll_time)
{
    real (* old_pos)[NDIM] = new real[n][NDIM];
    real (* old_vel)[NDIM] = new real[n][NDIM];
    real (* old_acc)[NDIM] = new real[n][NDIM];
    real (* old_jerk)[NDIM] = new real[n][NDIM];

    for (int i = 0; i < n; i++)
        for (int k = 0; k < NDIM; k++){
            old_pos[i][k] = pos[i][k];
            old_vel[i][k] = vel[i][k];
            old_acc[i][k] = acc[i][k];
            old_jerk[i][k] = jerk[i][k];
        }

    predict_step(pos, vel, acc, jerk, n, dt);
    get_acc_jerk_pot_coll(mass, pos, vel, acc, jerk, n, epot, coll_time);
    correct_step(pos, vel, acc, jerk, old_pos, old_vel, old_acc, old_jerk,
                 n, dt);
    t += dt;
}```
Taking the predictor step

```c
void predict_step(real pos[][NDIM], real vel[][NDIM],
                  const real acc[][NDIM], const real jerk[][NDIM],
                  int n, real dt)
{
    for (int i = 0; i < n ; i++)
        for (int k = 0; k < NDIM ; k++)
            pos[i][k] += vel[i][k]*dt + acc[i][k]*dt*dt/2
                         + jerk[i][k]*dt*dt*dt/6;
    vel[i][k] += acc[i][k]*dt + jerk[i][k]*dt*dt/2;
}
```

Computing the accelerations and jerks

This is similar to the `computeAccelerations` function in the MD programs. It is the most time consuming part of the computation because one must examine all $N(N - 1)/2$ pairs of particles. The following code also computes the total potential energy of the system

$$U = - \sum_{\text{pairs}} \frac{M_i M_j}{r_{ji}},$$

as well as two estimates of the “least collision time”, which are used to adjust the time step $\delta t$ as the computation proceeds.

```c
void get_acc_jerk_pot_coll(const real mass[], const real pos[][NDIM],
                           const real vel[][NDIM], real acc[][NDIM],
                           real jerk[][NDIM], int n, real & epot,
                           real & coll_time)
{
    for (int i = 0; i < n ; i++)
        for (int k = 0; k < NDIM ; k++)
            acc[i][k] = jerk[i][k] = 0;
```
epot = 0;
const real VERY_LARGE_NUMBER = 1e300;
real coll_time_q = VERY_LARGE_NUMBER; // collision time to 4th power
real coll_est_q; // collision time scale estimate // to 4th power (quartic)

for (int i = 0; i < n ; i++){
    for (int j = i+1; j < n ; j++){
        real rji[NDIM]; // rji[] is the vector from
        real vji[NDIM]; // particle i to particle j
        for (int k = 0; k < NDIM ; k++){
            rji[k] = pos[j][k] - pos[i][k];
            vji[k] = vel[j][k] - vel[i][k];
        }
        real r2 = 0; // | rji |^2
        real v2 = 0; // | vji |^2
        real rv_r2 = 0; // ( rji . vji ) / | rji |^2
        for (int k = 0; k < NDIM ; k++){
            r2 += rji[k] * rji[k];
            v2 += vji[k] * vji[k];
            rv_r2 += rji[k] * vji[k];
        }
        rv_r2 /= r2;
        real r = sqrt(r2); // | rji |
        real r3 = r * r2; // | rji |^3
        // add the {i,j} contribution to the total potential energy for the system:
        epot -= mass[i] * mass[j] / r;
        // add the {j (i)} contribution to the {i (j)} values of acceleration and jerk:
        real da[3]; // main terms in pairwise
        real dj[3]; // acceleration and jerk
        for (int k = 0; k < NDIM ; k++){
            da[k] = rji[k] / r3; // see equations
            dj[k] = (vji[k] - 3 * rv_r2 * rji[k]) / r3; // in the header
        }
        for (int k = 0; k < NDIM ; k++){
            acc[i][k] += mass[j] * da[k]; // using symmetry
            acc[j][k] -= mass[i] * da[k]; // find pairwise
            jerk[i][k] += mass[j] * dj[k]; // acceleration
            jerk[j][k] -= mass[i] * dj[k]; // and jerk
        }
    }
}
PHY 411-506 Computational Physics 2 18 Friday, May 9
// first collision time estimate, based on unaccelerated linear motion:

coll_est_q = (r2*r2) / (v2*v2);
if (coll_time_q > coll_est_q)
coll_time_q = coll_est_q;

// second collision time estimate, based on free fall:

real da2 = 0; // da2 becomes the square of the pair-wise acceleration between particles i and j
for (int k = 0; k < NDIM ; k++)
da2 += da[k] * da[k];
double mij = mass[i] + mass[j];
da2 *= mij * mij;
coll_est_q = r2/da2;
if (coll_time_q > coll_est_q)
coll_time_q = coll_est_q;
}

// from q for quartic back to linear collision time

coll_time = sqrt(sqrt(coll_time_q));
}

void correct_step(real pos[][NDIM], real vel[][NDIM],
const real acc[][NDIM], const real jerk[][NDIM],
const real old_pos[][NDIM], const real old_vel[][NDIM],
const real old_acc[][NDIM], const real old_jerk[][NDIM],
int n, real dt)
{
for (int i = 0; i < n ; i++)
for (int k = 0; k < NDIM ; k++){
vel[i][k] = old_vel[i][k] + (old_acc[i][k] + acc[i][k])*dt/2
+ (old_jerk[i][k] - jerk[i][k])*dt*dt/12;
pos[i][k] = old_pos[i][k] + (old_vel[i][k] + vel[i][k])*dt/2
+ (old_acc[i][k] - acc[i][k])*dt*dt/12;
}
}
Steering the calculation

The evolve function steers the calculation using the functions described above.

```c
void evolve(const real mass[], real pos[][NDIM], real vel[][NDIM],
            int n, real & t, real dt_param, real dt_dia, real dt_out,
            real dt_tot, bool init_out, bool x_flag)
{
    cerr << "Starting a Hermite integration for a " << n
         << "-body system,\n from time t = " << t
         << " with time step control parameter dt_param = " << dt_param
         << " until time " << t + dt_tot
         << " ,\n with diagnostics output interval dt_dia = "
         << dt_dia << " ,\n and snapshot output interval dt_out = "
         << dt_out << "." << endl;

    real (* acc)[NDIM] = new real[n][NDIM]; // accelerations and jerks
    real (* jerk)[NDIM] = new real[n][NDIM]; // for all particles
    real epot; // potential energy of the n-body system
    real coll_time; // collision (close encounter) time scale

    get_acc_jerk_pot_coll(mass, pos, vel, acc, jerk, n, epot, coll_time);

    int nsteps = 0; // number of integration time steps completed
    real einit; // initial total energy of the system

    write_diagnostics(mass, pos, vel, acc, jerk, n, t, epot, nsteps, einit,
                      true, x_flag);

    if (init_out) // flag for initial output
        put_snapshot(mass, pos, vel, n, t);

    real t_dia = t + dt_dia; // next time for diagnostics output
    real t_out = t + dt_out; // next time for snapshot output
    real t_end = t + dt_tot; // final time, to finish the integration

    while (true){
        while (t < t_dia && t < t_out && t < t_end){
            real dt = dt_param * coll_time;
            evolve_step(mass, pos, vel, acc, jerk, n, t, dt, epot, coll_time);
            nsteps++;
        }

        if (t >= t_dia){
            write_diagnostics(mass, pos, vel, acc, jerk, n, t, epot, nsteps,
                              true, x_flag);
        }

        t += dt;
    }
}
```
einit, false, x_flag);
    t_dia += dt_dia;
}
if (t >= t_out){
    put_snapshot(mass, pos, vel, n, t);
    t_out += dt_out;
}
if (t >= t_end)
    break;

delete[] acc;
delete[] jerk;
}

The remaining functions in the program are:

- **get_snapshot** reads an initial configuration of particles
- **put_snapshot** write the current configuration of particles
- **read_options** parses parameters specified by the user on the command line
- **write_diagnostics** writes more detailed output including the total and potential energies, etc.