Simulation Project

04.26.2019

Solar System Formation
Simulation in C++

By: Osman El-Ghotmi

Abstract

Computational and numerical methods in fluid dynamics and heat transfer is a diverse and intricate field. It can be used to describe the physical properties and motion of fluids in various applications and environments. It introduces a combination of numerical methods, fluid mechanics, and software computations. In recent years, the implementation of these studies have been used to model detailed simulations of large scale phenomena in our universe. These models range from the simulation of large galaxy collisions to the simulation of supernova explosions. This is a very inspiring and desirable application because it provides highly detailed and physically accurate models of cosmic events that occur in distant galaxies.

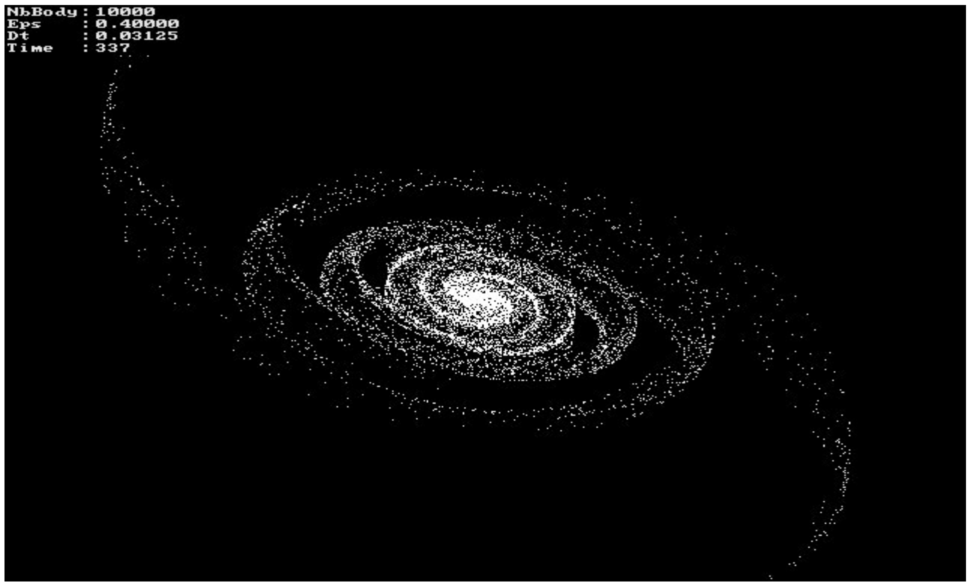

Figure 1: Galaxy Simulation

Contents

List of Figures

1 Introduction

The purpose of this project is to develop a simple two dimensional model that simulates the collapse of gas particles that eventually turn into a solar system. This is an important and interesting idea because it demonstrates the capability of numerical methods and computations in describing our physical universe. The early stages of a solar systems formation begins with a large collection of particles that are spread out with varying densities. These particles are then driven together to form stars, planets, moons, and other celestial bodies that remain in orbit with the respect to one another as a result of the force of gravity.

Figure 2 depicts a three dimensional representation for the formation of a solar system. The two dimensional model that will be described throughout this project will demonstrate similar attributes and properties.

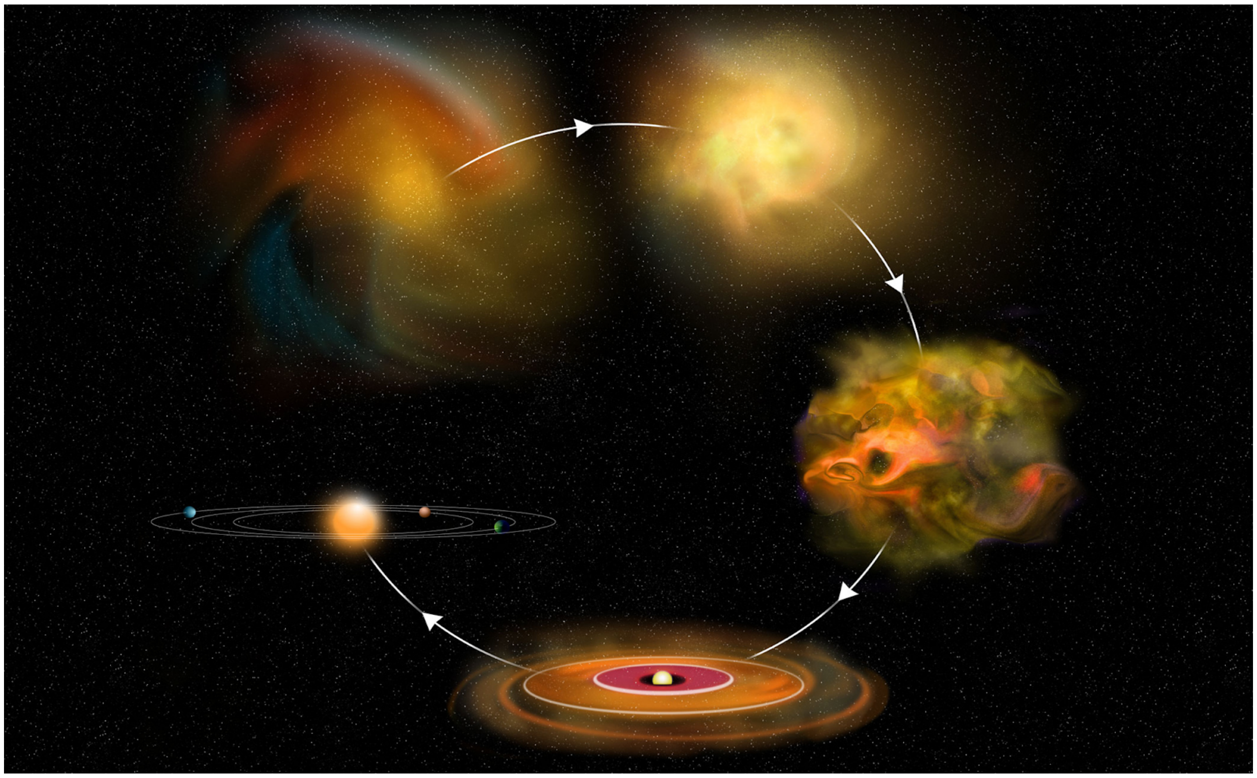

Figure 2: Solar System Formation

2 Problem Definition

The problem definition for this project is to develop and write a model that simulates the two dimensional collapse of gas particles to eventually form a solar system. The model is assumed to be in an absolute vacuum environment and so pressure will be neglected. The system can be modelled as a flow of gaseous particulates where the force of gravity is the driving force for their movement. Some of the equations used to describe the system are as follows:

$$\text{Force of Gravity: } \vec{F}_G = \frac{Gm_1m_2}{r^2} \cdot \hat{r}. \tag{2.1}$$

Where $m1$ and $m2$ are the individual masses being compared, r is the radial distance between the two masses, G is the gravitational constant:

$$G = 6.67408 \cdot 10^{-11}\mathrm{m}^3/\mathrm{kg} \cdot \mathrm{s}^2, \tag{2.2}$$

and $\hat{r}$ is the unit vector that describes the direction of the gravitational force, such that:

$$\hat{r} = \frac{x_2 - x_1}{|x_2 - x_1|}. \tag{2.3}$$

Another consideration is the initial conditions to be placed on each of the individual particles. To promote the formation of multiple bodies in the system, an initial orbital velocity can be initialized in each cell. The equation to describe orbital velocity is given by:

$$v = \sqrt{\frac{Gm}{r}}. \tag{2.4}$$

Initial conditions for particle densities will also be set. Cells in the center will have higher densities to promote the formation of a star and cells on the outside of the structured mesh will have lower densities to promote the formation of other celestial bodies.

3 Continuous Model

The model used in this simulation for particulate flow is represented by the following P.D.E.s.

$$\frac{\partial U}{\partial t} + \frac{\partial F_x}{\partial x} + \frac{\partial F_y}{\partial y} = Source. \tag{3.1}$$

Here, the flux vector U is represented by

$$U = \begin{bmatrix} \rho \\ \rho u_x \\ \rho u_y \end{bmatrix}, \tag{3.2}$$

where ρ is the density of the particles in each cell, u_x is the velocity of the cells particles in the x-direction, and u_y is the velocity of the cells particles in the y-direction.

The flux vector in the x-direction is given by:

$$F_x = \begin{bmatrix} \rho u_x \\ \rho u_x^2 \\ \rho u_x u_y \end{bmatrix}, \tag{3.3}$$

and the flux vector in the y-direction is given by:

$$F_y = \begin{bmatrix} \rho u_y \\ \rho u_x u_y \\ \rho u_y^2 \end{bmatrix}. \tag{3.4}$$

For the particulate flow model, Equation (3.1) is expanded to

$$\frac{\partial(\rho)}{\partial t} + \frac{\partial(\rho u_x)}{\partial x} + \frac{\partial(\rho u_y)}{\partial y} = Source, \tag{3.5}$$

in the x and y directions with the accelerating source term, we obtain:

$$\frac{\partial(\rho u_x)}{\partial t} + \frac{\partial(\rho u_x^2)}{\partial x} + \frac{\partial(\rho u_x u_y)}{\partial y} = \rho a_x, \tag{3.6}$$

$$\frac{\partial(\rho u_y)}{\partial t}+\frac{\partial(\rho u_x u_y)}{\partial x}+\frac{\partial(\rho u_y^2)}{\partial y}=\rho a_y. \tag{3.7}$$

The particulate flow model has maximum wave speeds that can be found by determining the eigenvalues of the flux Jacobian. The flux vector U, the flux in the x-direction F_x, and the flux in the y-direction F_y can be represented as:

$$U=\begin{bmatrix}U_0\\U_1\\U_2\end{bmatrix},\quad F_x=\begin{bmatrix}U_1\\\frac{U_1^2}{U_0}\\\frac{U_1U_2}{U_0}\end{bmatrix},\quad F_y=\begin{bmatrix}U_1\\\frac{U_1U_2}{U_0}\\\frac{U_1^2}{U_0}\end{bmatrix} \tag{3.8}$$

Next, the flux Jacobian is represented in the following matrix form:

$$J=\begin{bmatrix}\frac{\partial F_0}{\partial U_0}&\frac{\partial F_0}{\partial U_1}&\frac{\partial F_0}{\partial U_2}\\\frac{\partial F_1}{\partial U_0}&\frac{\partial F_1}{\partial U_1}&\frac{\partial F_1}{\partial U_2}\\\frac{\partial F_2}{\partial U_0}&\frac{\partial F_2}{\partial U_1}&\frac{\partial F_2}{\partial U_2}\end{bmatrix} \tag{3.9}$$

The flux Jacobian matrix in the x-direction then becomes:

$$\frac{\partial F_x}{\partial U}=\begin{bmatrix}0&1&0\\-\left(\frac{U_1}{U_0}\right)^2&2\frac{U_1}{U_0}&0\\-\frac{U_1U_2}{U_0^2}&\frac{U_2}{U_0}&\frac{U_1}{U_0}\end{bmatrix}. \tag{3.10}$$

Obtaining the characteristic equation of the flux Jacobian and solving for its roots provides us with the eigenvalues, which, in turn, provides us with the maximum wave speeds of the particulate flow model. The following is used to determine the characteristic equation.

$$\det\left(\frac{\partial F_x}{\partial U}-\lambda I\right)=0 \tag{3.11}$$

The result is a third degree polynomial who's roots are determined to be as follows:

$$\lambda_1=\lambda_2=\lambda_3=\frac{U_1}{U_0}=\frac{\rho u_x}{\rho}=u_x \tag{3.12}$$

Similarly, in the y-direction, solving for the maximum wave speed will yield u_y.

4 Discrete Model

The discrete model for this project will use a finite difference method. The reason for this is that finite difference methods can be easily applied to simple geometries mapped by Cartesian meshes. Finite-volume schemes can be of benefit where cell geometry becomes more complex. However, this project will be using a structured mesh where all cells are of equal sizes and so the finite difference method will suffice and be used in place of a finite volume method. Another thing to consider is the stability of the simulated system. Methods that respect the direction of information propagation are said to be up-winded and stable. Furthermore, a system can be made stable through a method of artificial dissipation. Artificial dissipation damps out oscillations caused by an unstable treatment of the hyperbolic parts of the system. This can be achieved through the use of the local lax friedrich method. The local lax friedrich for this model is described as:

$$\begin{aligned} U_{i,j}^{n+1} = U_{i,j}^{n} - \frac{\Delta t}{2\Delta x}(F_{i+1,j}^{n} - F_{i-1,j}^{n}) + \frac{\lambda_{max}\Delta t}{2\Delta x}(U_{j+1}^{n} - 2U_{j}^{n} + U_{j-1}^{n}) \\ - \frac{\Delta t}{2\Delta y}(F_{j+1}^{n} - F_{j-1}^{n}) + \frac{\lambda_{max}\Delta t}{2\Delta y}(U_{j+1}^{n} - 2U_{j}^{n} + U_{j-1}^{n}) + Source * \Delta t \end{aligned} \tag{4.1}$$

where the following define the positions in each cell:

- $F_{i+1,j} = F_{Right}$
- $F_{i-1,j} = F_{Left}$
- $U_{i+1,j} = U_{Right}$
- $U_{i,j} = U_{Middle}$
- $U_{i-1,j} = U_{Left}$
- $F_{i,j+1} = F_{Top}$
- $F_{i,j-1} = F_{Bottom}$
- $U_{i,j+1} = U_{Top}$
- $U_{i,j} = U_{Middle}$
- $U_{i,j-1} = U_{Bottom}$

The local lax friedrich method is stable if:

$$\Delta t < \frac{length_{min}}{\lambda_{max}} \tag{4.2}$$

Additionally a safety factor can be applied to the stability condition:

$$\Delta t = CFL \frac{length_{min}}{\lambda_{max}} \tag{4.3}$$

where the CFL number ranges from: $0 < CFL < 1$.

The next thing to consider is the implementation of the model. Since this model will be represented in a two dimensional environment, density will be represented as unit mass per unit area, where density is represented by ρ and area is represented by $\mathrm{d}A$ such that:

$$\mathrm{d}A = \mathrm{d}x\mathrm{d}y \tag{4.4}$$

The force of gravity term can be rewritten as follows:

$$\mathrm{d}\vec{F} = \frac{Gm_1m_2}{r^2} \cdot \frac{\vec{r}}{|\vec{r}|} = \frac{G\rho_1 \mathrm{d}x_1 \mathrm{d}y_1 \rho_2 \mathrm{d}x_2 \mathrm{d}y_2}{(\vec{x}_2 - \vec{x}_1)^2} \cdot \frac{\vec{x}_2 - \vec{x}_1}{|\vec{x}_2 - \vec{x}_1|} \tag{4.5}$$

$$\vec{F} = \iint_{A_2} \frac{G\rho_1 \mathrm{d}x_1 \mathrm{d}y_1 \rho_2 \mathrm{d}x_2 \mathrm{d}y_2}{(\vec{x}_2 - \vec{x}_1)^2} \cdot \frac{\vec{x}_2 - \vec{x}_1}{|\vec{x}_2 - \vec{x}_1|} \tag{4.6}$$

$$\vec{F} = \rho_1 \mathrm{d}x_1 \mathrm{d}y_1 \iint_{A_2} \frac{G\rho_2 \mathrm{d}x_2 \mathrm{d}y_2}{(\vec{x}_2 - \vec{x}_1)^2} \cdot \frac{\vec{x}_2 - \vec{x}_1}{|\vec{x}_2 - \vec{x}_1|} = m\vec{a_i}^n, \tag{4.7}$$

where $\vec{a_i}^n$ represents the resulting acceleration due to the effects of gravity evaluated at each cell with respect to every other cell. It can be written in the following form:

$$\vec{a_i}^n = \sum_{j=0}^{\#\ \text{Cells}} \frac{G\rho_j^n A_j}{(\vec{x}_{cj} - \vec{x}_{ci})^2} \cdot \frac{\vec{x}_{cj} - \vec{x}_{ci}}{|\vec{x}_{cj} - \vec{x}_{ci}|}. \tag{4.8}$$

5 Resolution Study

Once the described model has been implemented, it is important to perform a resolution study and analyze the programs results for varying geometries, initial conditions, and cell sizes for a given area. The first study that I conducted was for a geometry where density in the center is high with an outer ring who's density is moderate. I used this configuration for my resolution study. It is also worth noting that the effects of gravity are always acting upon each of the particulates in the model and so steady-state is never reached in the simulation. It is also important to note that the method used to calculate the gravitational effects of each cell against every other cell becomes an n^4 problem. As a result, increasing the mesh size drastically increases the cost and time to run these simulations and conduct the computations. Mesh sizes between 25x25 and 50x50 are reasonable to conduct; however, mesh sizes between 100x100 and 1000x1000 were proven to be extremely time consuming and unreasonable to conduct within the scope of this project.

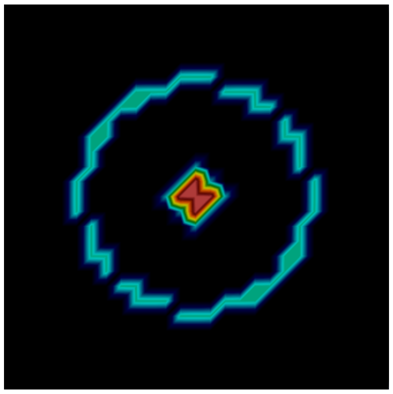

Figure 3: Resolution 25 x 25 (1)

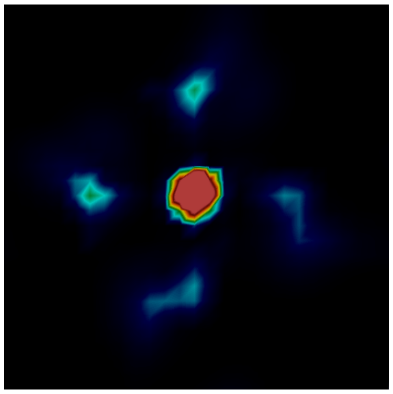

Figure 4: Resolution 25 x 25 (2)

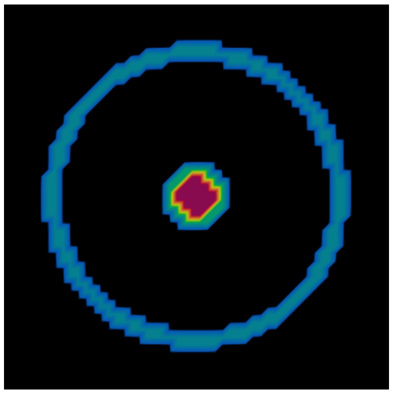

Figure 5: Resolution 50 x 50 (1)

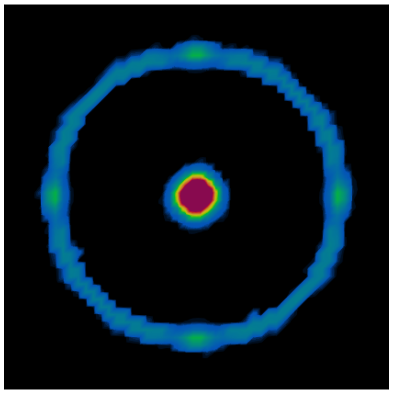

Figure 6: Resolution 50 x 50 (2)

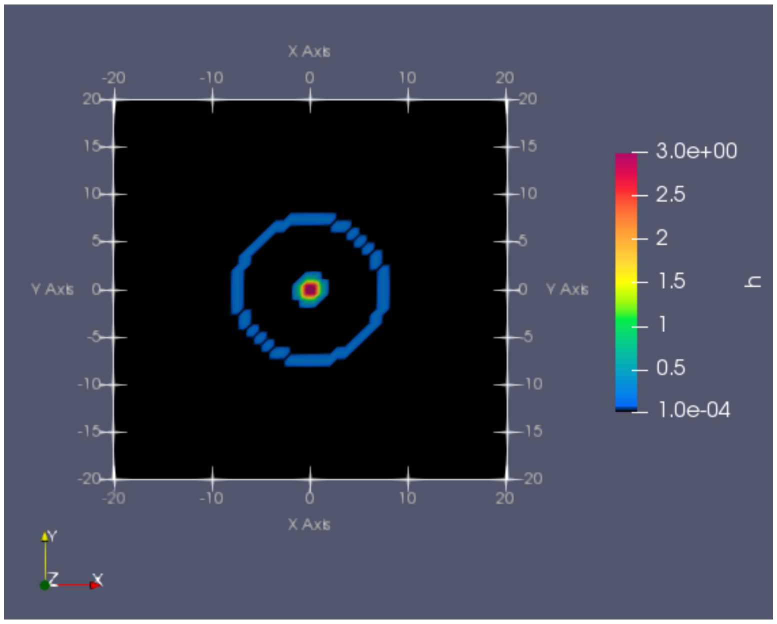

Figure 7: Resolution 50 x 50 - 40x40 Grid (1)

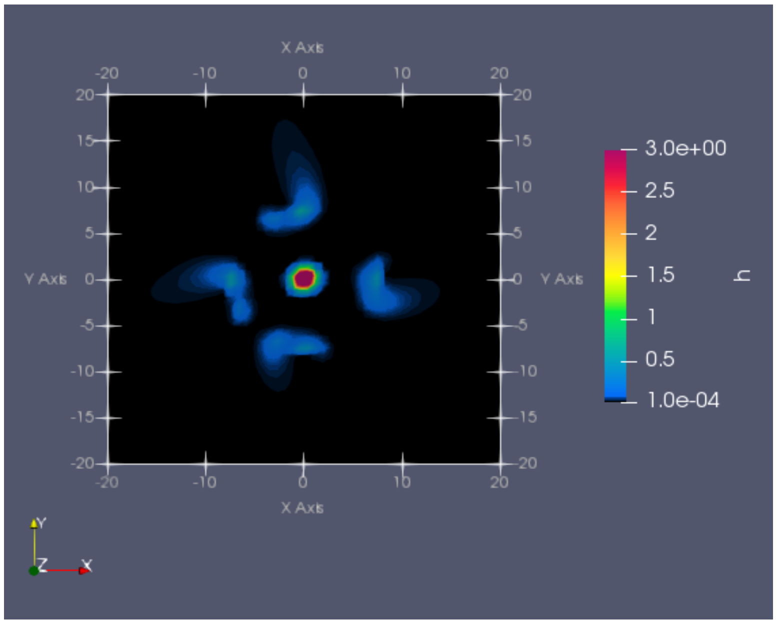

Figure 8: Resolution 50 x 50 - 40x40 Grid (2)

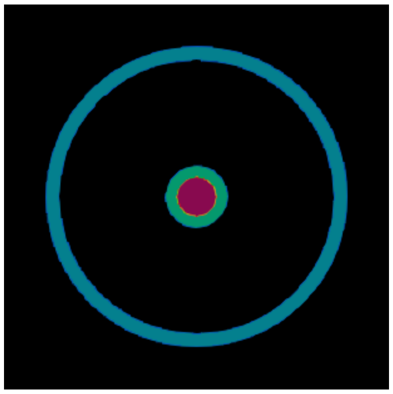

Figure 9: Resolution 500 x 500 (1)

Increasing the mesh size, increases the computing time and power by a substantial amount. Lower mesh sizes provide quicker results but provide poor representations of the ideal geometry based on the initial conditions. At a mesh size of 50x50, the results demonstrate that the code was implemented correctly and that the previously described physics is being executed accordingly. Figure 9 is a demonstration of initial geometry at a mesh size of 500x500; however, the computing time and power required to run the simulation is too high for a standard desktop computer.

6 Results

In this section of the report, I provide the various results that I obtained for the simulations that I conducted using distinct and varying initial conditions. The results presented in this section are done with a 50x50 size mesh. The gravitational constant was also increased in order to see more detailed responses in a shorter time frame.

The first study features a highly dense core with two moderately dense rings. No initial orbital velocity was applied to this configuration.

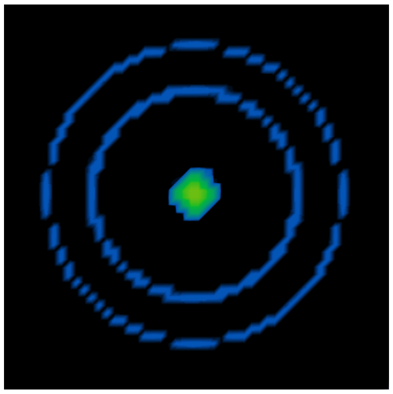

Figure 10: First Study (1)

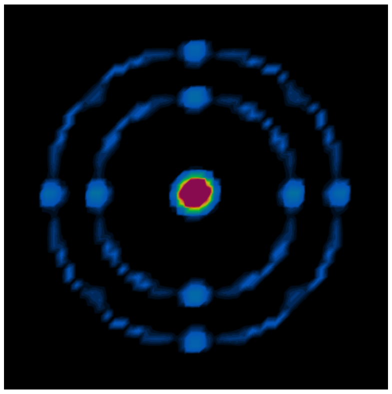

Figure 11: First Study (2)

The second study features a cloud of particles with concentrated masses. No initial orbital velocity was applied to this configuration.

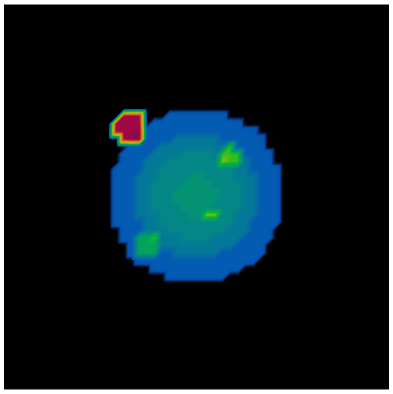
Figure 12: Second Study (1)

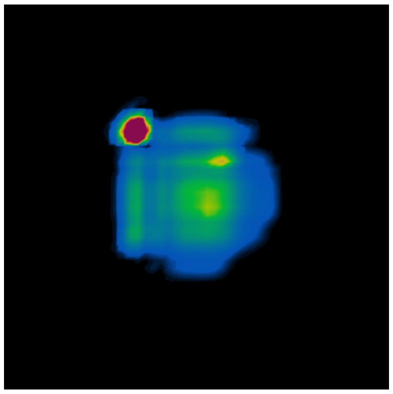
Figure 13: Second Study (2)

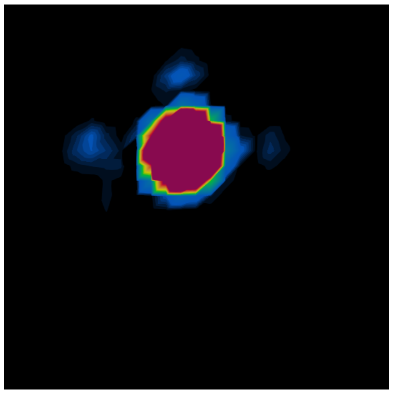
Figure 14: Second Study (3)

The third study features a highly dense core with two moderately dense rings. high orbital velocity was applied to this configuration.

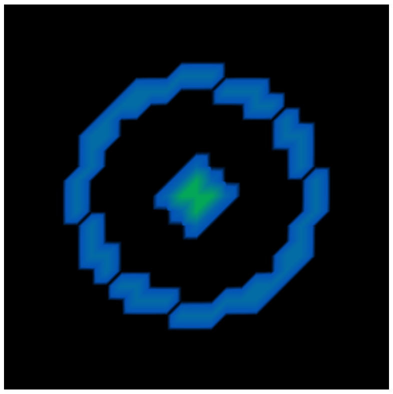
Figure 15: Third Study (1)

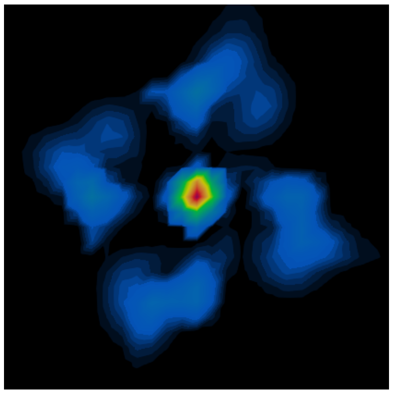
Figure 16: Third Study (2)

7 Conclusion

In conclusion, the implementation for the particulate flow model seems to demonstrate an accurate depiction for the effects that gravity would have on varying geometries. The low resolution simulations produce inaccurate representations of the desired geometries and, in turn, prevent accurate results. Ideally, computations of these sorts would run with more processing time or more processing power in order to generate more detailed models.

Appendices

Appendix A: Code

Main C++ Code

```cpp

//*********************************************************************//
//                       Osman El-Ghotmi                               //
//                    Solar System Formation                           //
//                         CFD Project                                 //
//*********************************************************************//

#include "eigen/Dense"
#include "eigen/Eigen"
#include <fstream>
#include <functional>
#include <iomanip>
#include <iostream>
#include <limits>
#include <string>
#include <sstream>
#include <utility>
#include <vector>

////////////////////////////////////////////////////////////////////////////

//Class representing particulate model solver
class Particulate_Model_Solver {
public:

    //Default constructors and assignment
    Particulate_Model_Solver()                                           = default;
    Particulate_Model_Solver(const Particulate_Model_Solver&)            = default;
    Particulate_Model_Solver(Particulate_Model_Solver&&)                 = default;
    Particulate_Model_Solver& operator=(const Particulate_Model_Solver&)    = default;
    Particulate_Model_Solver& operator=(Particulate_Model_Solver&&)         = default;

    //Constructors with real information
    Particulate_Model_Solver(double x_min_in, double x_max_in, int nx_in,
                             double y_min_in, double y_max_in, int ny_in,
                             const std::function<Eigen::Vector3d(double,double)>&
                                 initial_condition) :
            SolutionVector(3*nx_in*ny_in), //Solution vector composed of nx and ny
            ResidualVector(3*nx_in*ny_in), //Residual vector composed of nx and ny
            x_min(x_min_in), //Minimum X value from input
            x_max(x_max_in), //Maximum X value from input
            y_min(y_min_in), //Minimum Y value from input
            y_max(y_max_in), //Maximum Y value from input
            nx(nx_in), //nx from input
            ny(ny_in), //ny from input
            delta_x((x_max-x_min)/static_cast<double>(nx+1)), //Grid value of delta x
            delta_y((y_max-y_min)/static_cast<double>(ny+1)), //Grid value of delta y
            current_time(0.0) //Current time
    {
        //Checking for valid inputs
        if(x_min > x_max || nx <= 0 || y_min > y_max || ny <= 0) {
            throw std::runtime_error("Invalid inputs to solver.");
        }
        //Setting the initial value conditions
        set_initial_conditions(initial_condition);
    }

    //Returning values for nodes and grid dimensions
    auto num_x() const {return nx;} //Number of nodes in the x direction
    auto num_y() const {return ny;} //Number of nodes in the y direction
    auto dx()    const {return delta_x;} //delta x
    auto dy()    const {return delta_y;} //delta y

    //Positions of i-j nodes
    Eigen::Vector2d node(int i, int j) const {
        while   (i<0) i     += nx;
        auto    index_i    = i%nx;
        while   (j<0) j     += ny;
        auto    index_j    = j%ny;
        return {x_min + static_cast<double>(index_i)*delta_x + (i/nx)*(x_max-x_min),
                y_min + static_cast<double>(index_j)*delta_y + (j/ny)*(y_max-y_min)};
    }

    auto& U() {return SolutionVector;} //Full solution vector
    const auto& U() const {return SolutionVector;} //Full solution vector

    //Solution at i-j node. Returns eigen segment. Desired subvector reference
    auto U(int i, int j) {
        auto index = global_solution_index(i,j);
        return SolutionVector.segment<3>(index);
    }

    //Solution at i-j node. Returns eigen segment. Desired subvector reference
    auto U(int i, int j) const {
        auto index = global_solution_index(i,j);
        return SolutionVector.segment<3>(index);
    }

    auto& dUdt() {return ResidualVector;} //Full residual vector
    const auto& dUdt() const {return ResidualVector;} //Full residual vector

    //Solution at i-j node. Returns eigen segment. Desired subvector reference
    auto dUdt(int i, int j) {
        auto index = global_solution_index(i,j);
        return ResidualVector.segment<3>(index);
    }

    //Solution at i-j node. Returns eigen segment. Desired subvector reference
    auto dUdt(int i, int j) const {
        auto index = global_solution_index(i,j);
        return ResidualVector.segment<3>(index);
    }

    auto time() const {return current_time;} //Current time

    //Force x
    Eigen::Vector3d Fx(const Eigen::Vector3d& U) {
        return {U[1], U[1]*U[1]/U[0], U[1]*U[2]/U[0]};
    }

    //Force y
    Eigen::Vector3d Fy(const Eigen::Vector3d& U) {
        return {U[2], U[2]*U[1]/U[0], U[2]*U[2]/U[0]};
    }

    //Max wave speeds in x and y directions
    Eigen::Vector2d max_wavespeeds(const Eigen::Vector3d& U) {
        double a  = 0;
        double ux = U[1]/U[0];
        double uy = U[2]/U[0];
        return {fabs(ux)+a, fabs(uy)+a};
    }

    void time_march(double final_time, double CFL); //Time march to time

private:

    //Index in solution vector. Storing location for node i-j entries
    int global_solution_index(int i, int j) const {
        while(i<0) i += nx;
        auto index_i = i%nx;
        while(j<0) j += ny;
        auto index_j = j%ny;
        return 3 * (index_i + nx*index_j);
    }

    //Set initial conditions
    void set_initial_conditions(const std::function<Eigen::Vector3d(double, double)>&
        initial_condition) {
        for(int i = 0; i < nx; ++i) {
            for(int j = 0; j < ny; ++j) {
                auto n = node(i,j);
                U(i,j) = initial_condition(n.x(), n.y());
            }
        }
    }

    Eigen::VectorXd SolutionVector; //Solution vector (U)
    Eigen::VectorXd ResidualVector; // Residual vector (dUdt)
    const double x_min; //Minimum value of x
    const double x_max; //Maximum value of x
    const int nx; //Number of nodes in the x direction
    const double delta_x; //Spacing of nodes in the x direction
    const double y_min; //Minimum value of y
    const double y_max; //Maximum value of y
    const int ny; //Number of nodes in the y direction
    const double delta_y; //Spacing of nodes in the y direction
    double current_time; //Solution time
    constexpr static double g_const = 6.67408e-11; //Gravitational constant
};

//////////////////////////////////////////////////////////////////////////

//Local lax friedrichs time march
void Particulate_Model_Solver::time_march(double final_time, double CFL) {
    std::cout << "Time marching from t = " << current_time
              << " to t = " << final_time
              << ".  Total difference = " << final_time-current_time << ".\n";
    const double time_tolerance = 1.0e-12; //Tolerance
    const double min_length = std::min(dx(), dy()); //Minimum length
    double A = delta_x*delta_y; //Area
    double mass_one = 0; //Mass of evaluated cell
    double mass_two = 0; //Mass of compared cell
    double distance_X = 0; //X direction distance
    double distance_Y = 0; //Y direction distance
    double calculated_Force_X = 0; //Calculated force in the x direction
    double calculated_Force_Y = 0; //Calculated force in the y direction
    double acceleration_X = 0; //Acceleration in the x direction
    double acceleration_Y = 0; //Acceleration in the y direction
    double x_position = 0.0;
    double y_position = 0.0;
    double radius = 0.0;
    double velocity = 0.0;
    double theta = 0.0;
    double rho = 0.0;
    int counter = 0;

    while(current_time < final_time - time_tolerance) {
        auto dt = std::numeric_limits<double>::max();
        dUdt().fill(0.0);
        for(int i = 0; i < num_x(); ++i) {
            for(int j = 0; j < num_y(); ++j) {
                Eigen::Vector3d Um = U(i  , j  );
                Eigen::Vector3d Ul = U(i-1, j  );
                Eigen::Vector3d Ur = U(i+1, j  );
                Eigen::Vector3d Ub = U(i  , j-1);
                Eigen::Vector3d Ut = U(i  , j+1);
                Eigen::Vector2d lambda = max_wavespeeds(Um);
                auto max_lambda = std::max(lambda[0], lambda[1]);
                dt = std::min(dt, CFL*min_length/max_lambda);
                dUdt(i,j) =  (Fx(Ul) - Fx(Ur) + lambda[0]*(Ul - 2.0*Um + Ur)) / (2.0*delta_x)
                             +(Fy(Ub) - Fy(Ut) + lambda[1]*(Ub - 2.0*Um + Ut)) / (2.0*delta_y);
            }
        }

        //Initial Condition for Orbital Velocity
        if(counter == 0) {
            counter = 1;
            for(int i = x_min; i < x_max; ++i) {
                for(int j = y_min; j < y_max; ++j) {
                    x_position = i*delta_x;
                    y_position = j*delta_y;
                    radius = sqrt(fabs(x_position*x_position) + fabs(y_position*y_position));
                    theta = atan2(y_position, x_position);
                    rho = U(i,j)[0];
                    if(radius > 0.0){
                        velocity = sqrt(fabs((g_const*rho*A)/radius));
                    }
                    else{
                        velocity = 0.0;
                    }
                    U(i,j)[1] += -rho*velocity*cos(theta+3.14159/2);
                    U(i,j)[2] += -rho*velocity*sin(theta+3.14159/2);
                }
            }
        }

        //Force of gravity calculation
        for(int i1 = 0; i1 < num_x(); ++i1){
            //New cell to be evaluated: x coordinate
            for(int j1 = 0; j1 < num_y(); ++j1){
                //New cell to be evaluated: y coordinate
                mass_one = U(i1,j1)[0]*A; //Calculating the mass of the cell being evaluated
                for(int i2 = 0; i2 < num_x(); ++i2){
                    //Comparing cell: x coordinate
                    for(int j2 = 0; j2 < num_y(); ++j2){
                        //Comparing cell: y coordinate
                        mass_two = U(i2,j2)[0]*A; //Calculating the mass of the comparing cell
                        distance_X = (i2 - i1)*delta_x; //Distance in the x direction
                        distance_Y = (j2 - j1)*delta_y; //Distance in the y direction

                        //Calculating the force between each cell
                        if(fabs(distance_X) > 0.0 && fabs(distance_Y) > 0.0) {
                            calculated_Force_X = mass_one*((g_const*mass_two)/((distance_X)*(fabs(distance_X))));
                            calculated_Force_Y = mass_one*((g_const*mass_two)/((distance_Y)*(fabs(distance_Y))));
```

```cpp
                        }
                        else if(fabs(distance_Y) > 0.0 && fabs(distance_X) == 0.0) {
                            calculated_Force_X = 0.0;
                            calculated_Force_Y = mass_one*((g_const*mass_two)/((distance_Y)
                                *(fabs(distance_Y))));
                        }

                        else if(fabs(distance_X) > 0.0 && fabs(distance_Y) == 0.0) {
                            calculated_Force_X = mass_one*((g_const*mass_two)/((distance_X)
                                *(fabs(distance_X))));
                            calculated_Force_Y = 0.0;
                        }

                        //Calculating the acceleration resulting between each of the cells
                        acceleration_X = calculated_Force_X/mass_one;
                        acceleration_Y = calculated_Force_Y/mass_one;

                        //Summing the resulting accelerations of each cell
                        U(i2,j2)[1] += -acceleration_X*dt*U(i1,j1)[0];
                        U(i2,j2)[2] += -acceleration_Y*dt*U(i1,j1)[0];

                        //Output comments to analyse the resulting output from each of the
                            cells
                        //std::cout << "mass one: " << mass_one << ".\n";
                        //std::cout << "mass two: " << mass_two << ".\n";
                        //std::cout << "distance X: " << distance_X << ".\n";
                        //std::cout << "distance Y: " << distance_Y << ".\n";
                        //std::cout << "Force X: " << calculated_Force_X << ".\n";
                        //std::cout << "Force Y: " << calculated_Force_Y << ".\n";
                    }
                }
            }
        }

        //Checking the position of the current time
        if(current_time + dt > final_time) {
            dt = final_time - current_time;
        }

        U()  += dt*dUdt();
        current_time += dt;
        //std::cout << "Time = " << std::setw(10) << current_time << '\n';
    }

    std::cout << "Time marching done.\n";
}

///////////////////////////////////////////////////////////////////////////

//Writing out to VTK
void write_to_VTK(const Particulate_Model_Solver& solver,
                  const std::string& filename) {
```

```
    std::ofstream fout(filename);
    if(!fout) {
        throw std::runtime_error("Could not open file: " + filename);
    }

    std::cout << "Writing output to file: " << filename << '\n';
    const auto num_nodes = (solver.num_x()+1)*(solver.num_y()+1);
    fout << "# vtk DataFile Version 2.0\n"
         << "Particulate Flow Equations Solution\n"
         << "ASCII\n"
         << "DATASET STRUCTURED_GRID\n"
         << "DIMENSIONS " << solver.num_x()+1 << " " << solver.num_y()+1 << " 1\n"
         << "POINTS " << num_nodes << " double\n";
    for(int j = 0; j <= solver.num_y(); ++j) {
        for(int i = 0; i <= solver.num_x(); ++i) {
            auto n = solver.node(i,j);
            auto U = solver.U(i,j);
            fout << n.x() << " " << n.y() << " " << 0.0 << '\n';
        }
    }

    fout << "\nPOINT_DATA " << num_nodes
         << "\nSCALARS h double 1\nLOOkUP_TABLE default\n";
    for(int j = 0; j <= solver.num_y(); ++j) {
        for(int i = 0; i <= solver.num_x(); ++i) {
            fout << solver.U(i,j)[0] << '\n';
        }
    }

    fout << "\nVECTORS u double\n";
    for(int j = 0; j <= solver.num_y(); ++j) {
        for(int i = 0; i <= solver.num_x(); ++i) {
            auto ux = solver.U(i,j)[1]/solver.U(i,j)[0];
            auto uy = solver.U(i,j)[2]/solver.U(i,j)[0];
            fout << ux << " " << uy << " 0.0\n";
        }
    }
}

//////////////////////////////////////////////////////////////////////

//Making movie
void make_movie(Particulate_Model_Solver& solver,
                const double final_time,
                const double CFL,
                const int number_of_frames,
                const std::string& filename_base) {
    if(number_of_frames < 2) {
        throw std::runtime_error("make_movie requires at least two frames.");
    }

    const auto frame_time = (final_time-solver.time())/static_cast<double>(number_of_frames
        -1);
    const auto build_filename = [](const std::string& filename_base, int index) {
        std::stringstream filename_ss;
        filename_ss <<  filename_base << "_" << std::setfill('0') << std::setw(5) << index
            << ".vtk";
        return filename_ss.str();
    };

    write_to_VTK(solver, build_filename(filename_base, 0));
    for(int i = 1; i<number_of_frames; ++i) {
        double target_time = static_cast<double>(i)*frame_time;
        solver.time_march(target_time, CFL);
        write_to_VTK(solver, build_filename(filename_base, i));
    }
}

//////////////////////////////////////////////////////////////////////////

//Main
int main() {
    std::cout << "|-------------------------------------|\n"
              << "|          Osman El-Ghotmi            |\n"
              << "|       Solar System Formation        |\n"
              << "|-------------------------------------|\n";
    double x_min = -10.0; //Initial condition for minimum x value
    double x_max =  10.0; //Initial condition for maximum x value
    double y_min = -10.0; //Initial condition for minimum y value
    double y_max =  10.0; //Initial condition for maximum y value
    int       nx =  25; //Initial condition for number of x cells
    int       ny =  25; //Initial condition for number of y cells

    //Initial conditions for density values
    auto initial_condition = [] (double x, double y) {
        Eigen::Vector3d U;
        U[0] = 1.0e-4;
        U[1] = 1.0e-4;
        U[2] = 1.0e-4;
        if(fabs(x*x+y*y) < 1.0) {
            U[0] += 3.0;
        }
        if(fabs(x*x+y*y) < 2.5 && fabs(x*x+y*y) >= 1.0) {
            U[0] += 0.75;
        }
        if(fabs(x*x+y*y) < 40.0 && fabs(x*x+y*y) >= 30.0) {
            U[0] += 1.0;
        }
        return U;
    };

    //Calling on the particulate model solver with initial conditions
    auto solver = Particulate_Model_Solver(x_min, x_max, nx, y_min, y_max, ny,
        initial_condition);

    //Creating a movie for paraview
    make_movie(solver, 2.5, 0.5, 750, "movie");
    return 0;
}
```