Homework assignment report

## N-body simulation using numerical methods

## Introduction

The goal of this project was to build an n-body simulation using the knowledge learnt during Computer Science 2 course. The program uses four different methods of integration to compute paths of bodies and is capable of detecting and simulating simple collisions between them.

## Initialization

At startup the program will ask the user to specify the initialization method, the user can choose between reading the data from a text file and generating a procedural planetary system with either one or two stars in the middle and a few planets orbiting them. The stars generated by the program are roughly equivalent to real-life main sequence stars while planets fall within around half of the mass of Mercury and several times the mass of Jupiter. Obraz zawierający laptop, czarny, siedzi, patrzenie

Opis wygenerowany automatycznie

Figure 1 A randomly generated binary system with 5 planets

## Representing individual bodies

The program uses two structures to store information about bodies:

* Vector:

struct Vector

{

double x;

double y;

double z;

};

* Body:

struct Body

{

int index;

double mass;

Vector pos;

Vector vel;

double rad;

bool has\_collided;

int collision\_count;

};

## Numerical methods used

As stated in the introduction, the program uses four different methods to perform integration. Each one having their own function that takes an array of bodies, their number and length of a timestep as arguments. Verlet integration is enabled by default, but can be changed it by un-commenting one of the following lines.

//symplecticEulerForce(body, n, dt);

//eulerForce(body, n, dt);

//rk4Force(body, n, dt);

verletForce(body, n, dt);

* 4th order Runge-Kutta method – Applying 4th order Runge-Kutta method to solve mechanical problems can be tricky due to presence of a second order derivative, it requires integrating both the position and velocity in parallel and switching between the two during the calculations.

void rk4Force(Body\* body, int n, double dt);

* Euler method.

void eulerForce(Body\* body, int n, double dt);

* Semi-implicit Euler method – It’s actually easier to implement than standard Euler method since it doesn’t require storing the values of acceleration, since change of velocity is applied right away.

void symplecticEulerForce(Body\* body, int n, double dt);

* Velocity Verlet – It’s a simple 2nd order method, almost as simple as semi-implicit Euler, but much more accurate.

void symplecticEulerForce(Body\* body, int n, double dt);

## Comparison of different integration methods

To compare different methods of integration we measure how much energy a given system has at the beginning and how it changes over the course of the simulation. For that purpose I set up a two-body system with Earth and Moon orbiting it with a step size of 50,000s over a course of 10,000 iterations. I calculated the relative error by taking an average value of energy from the last 10 iterations, dividing it by the initial energy and subtracting one. Below are the resulting graphs:

We can see that that two symplectic methods (semi-implicit Euler and velocity Verlet) show a rapid wobble, however their error never crosses a certain treshold. On the other hand 4th order  
Runge-Kutta while really steady, shows a linear upward trend. Standard Euler method causes the system to fall apart almost immidiately. It be concluded that high-order non-symplectic methods are appropriate for short simulations, where accuracy within a single or a few orbits is key, but for long term simulations that are supposed to capture a gradual evolution of a system over eons, symplectic methods might offer a higher degree of accuracy, as their errors don’t seem to add up as much over time.