## 1 The Problem

In this report I programmed a gravity simulation that approximates the evolution of position and velocity of N stars in 2 dimensions. The code was written in C. The problem consists of reading a binary file containing information about position, velocity, mass and brightness of each star (the latter attribute was not used in this solution). The code was to be optimized for runtime while keeping the accuracy within a few magnitudes of order from the rounding errors of the datatypes used. This assignment was restricted to double types for the attributes of the stars. I used the modified force calculation called plummer-spheres, that reduces the risk of instability as stars get close and the denominator gets small by adding a constant epsilon:

$$\vec{F_i} = -Gm_i \sum_{j=0, j\neq i}^{N-1} \frac{m_j}{(r_{ij}+\epsilon_0)^3}\vec{r_{ij}} \tag{1}$$

I used the euler symplectic update of the stars velocity and position, because it preserves the total energy of the system.

## 2 The Solution

All following performance tests were run using the following parameters:

- N = 3000
- $\Delta t = 10^{-5}$
- $\epsilon_0 = 10^{-3}$
- $G = \frac{100}{N}$
- steps = 100

Where N is the number of stars, $\Delta t$ is the time stepsize, $\epsilon_0$ is the dampening factor in the force calculation, G is the gravitational constant and steps is the number of updates to iterate through. N and steps was initialized as const ints and $\Delta t$, $\epsilon_0$ and G as const doubles for higher precision.

For all the performance tests I verified that the code solved the problem with at least an error less than $10^{-13}$ that was validated using reference files with the correct final state of the system. Graphics was used to get a better understanding of the bugs in the code, but once the code functioned as intended the graphics was turned off to allow for a faster runtime. All timings were calculated as a mean over 3 runs, using -O3 compiler optimization. I will briefly highlight the relevant parts of the codes that did not do best in testing, and end the report with the full contents of the code that performed best. The timings of all codes as well was a deeper discussion of optimization techniques used, considered and discarded can be found under section "Performance and Discussion". All codes presented used epsilon as a global constant, and all codes finalized the run by writing the results to a new .gal file.

## 2.1 First version

My first iteration of a solution consisted of a struct type as follows:

```
/* Define struct for a celestial body */
typedef struct {
        double xp; //X Position
        double yp; //Y Position
        double mass;
        double xv; //X Velocity
        double yv; //Y Velocity
        double brightness; //Keep, but not used in this simulation
} cbody_t;
```

Initialized as:

```
cbody_t stars[N];
```

And the function that updates the stars were initialized with following input arguments:

```
void update_stars(cbody_t stars[], double G, int N, double delta_t){
            for (int i = 0; i < N; i++) {
                /* Initialize variables used in this scope */

                /* More code... */

                for (int j = 0;j < N; j++) {
                        if (j == i){
                                continue;
                        }
                        /* More code... */

                        /* Acceleration contribution to star i from star j */
                        double ax = fx/stars[i].mass;
                        double ay = fy/stars[i].mass;

                        /* Speed contribution to star i from star j*/
                        sum_xv += delta_t*ax;
                        sum_yv += delta_t*ay;
                }
                /* New speed of star i*/
                stars[i].xv += sum_xv;
                stars[i].yv += sum_yv;

                /* New position of star i */
                stars[i].xp += delta_t*stars[i].xv;
                stars[i].yp += delta_t*stars[i].yv;
        }
}
```

## 2.2 Second version

My second iteration of a solution kept the same structure for the struct cbody_t, but initialized using dynamic allocation with pointers instead.

```
cbody_t *stars = (cbody_t *)malloc(N*sizeof(cbody_t));
```

Meaning that now the variable stars is a pointer to an array of cbody_t structs, the size of N. This pointer was passed to the update_stars function. I changed the function update_stars accordingly as well as added an optimization to the inner loop, further discussed in the next section.

```
void update_stars(cbody_t *stars, double G, int N, double dt){
    /* Initialize variables used in this scope */
```

```
        cbody_t *star_ptr_i = stars;
        cbody_t *star_ptr_j = stars + 1;

        /* More code... */
        for (int i = 0; i < N; i++) {
                /* Initialize variables used in this scope */

                /* More code... */

                /* Add contribution from i to all j as well as all j to i */
                for (int j = i+1;j < N; j++) {
                        /* More code... */

                        /*add this contribution to j */
                        (star_ptr_j)->xv += vj*rix;
                        (star_ptr_j)->yv += vj*riy;
                        star_ptr_j++;
                }
                /* New speed of star i as consequence of all other stars*/
                (star_ptr_i)->xv += -Gdt*fx_sum/im;
                (star_ptr_i)->yv += -Gdt*fy_sum/im;
                (star_ptr_i)->xp += ((star_ptr_i)->xv)*dt;
                (star_ptr_i)->yp += ((star_ptr_i)->yv)*dt;
                star_ptr_i++;
                star_ptr_j = star_ptr_i + 1;
        }
}
```

## 2.3 Third and final version

In the final code I changed the structure of the struct cbody_t:

```
/* Define struct for a celestial body */
typedef struct {
        double* xp; //X Position
        double* yp; //Y Position
        double* mass;
        double* xv; //X Velocity
        double* yv; //Y Velocity
        double* brightness; //Keep, but not used in this simulation
} cbody_t;
```

And initialized as:

```
    cbody_t stars;
        stars.xp = malloc(N * sizeof(double));
        stars.yp = malloc(N * sizeof(double));
        stars.mass = malloc(N * sizeof(double));
        stars.xv = malloc(N * sizeof(double));
        stars.yv = malloc(N * sizeof(double));
        stars.brightness = malloc(N * sizeof(double));
```

meaning that I know pass a structure of arrays instead of an array of structures. I initialized pointers in the function update_stars that each pointed at one of the arrays holding a specific attribute of all the stars in the simulation.

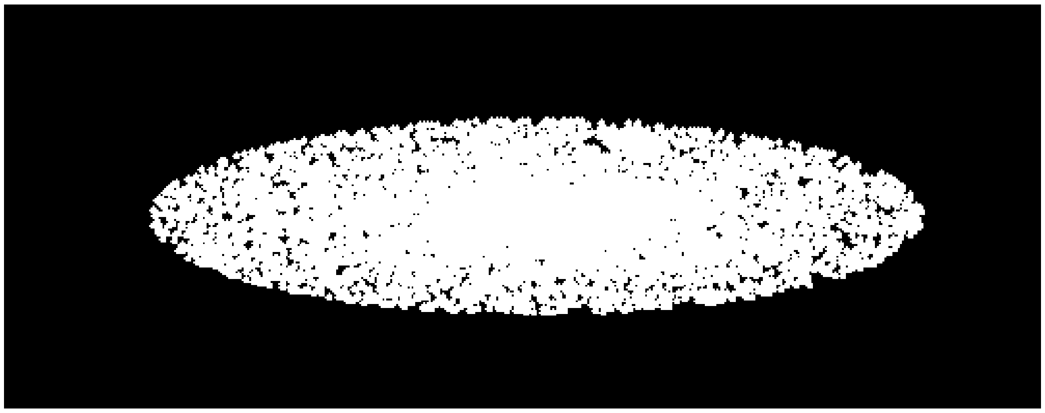

Figure 1: Initial galaxy state, N = 10000

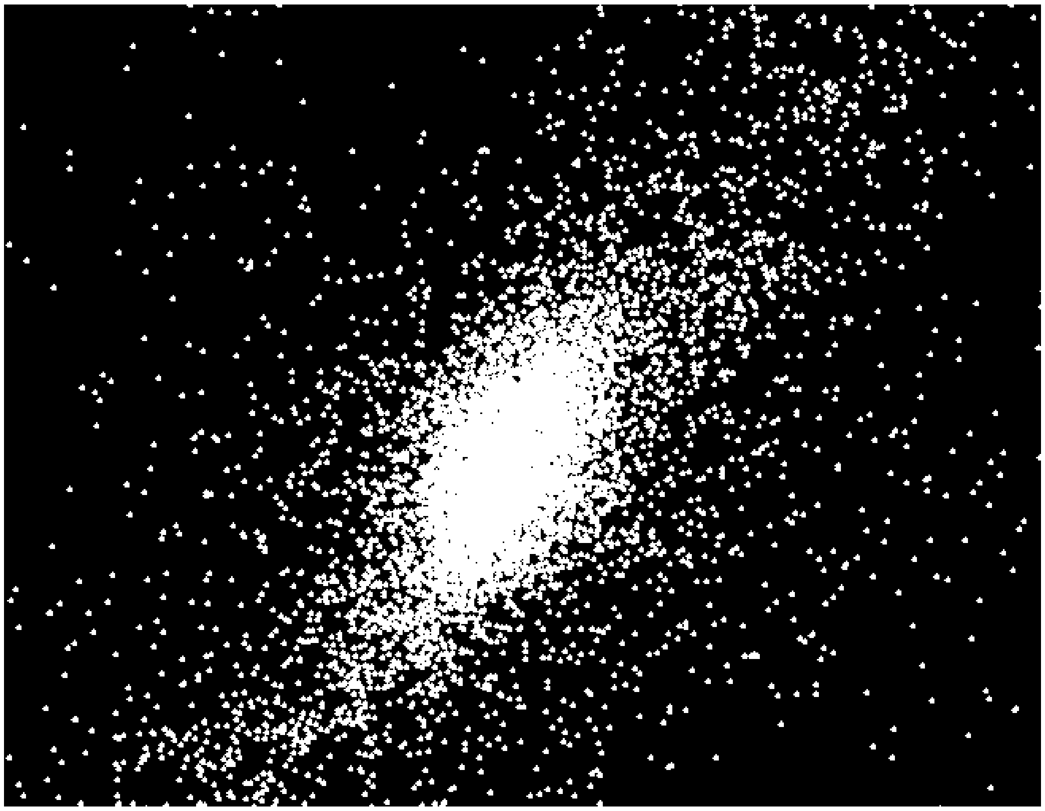

Figure 2: Galaxy state after a few hundred steps

# 3 Performance and Discussion

## 3.1 CPU and compiler used in this assignment

All codes were tested on a virtual linux machine (fredholm) provided by Uppsala University, here are the CPU specifications:

- Vendor ID: GenuineIntel
- Model name: Intel(R) Xeon(R) CPU E5520 @ 2.27GHz
- CPU family: 6
- Model: 26

- Thread(s) per core: 2
- Core(s) per socket: 4
- Socket(s): 2
- Stepping: 5
- CPU max MHz: 2267,0000
- CPU min MHz: 1600,0000
- BogoMIPS: 4533.50

Compiler information: 4:11.2.0-1ubuntu1 amd64 GNU C compiler

## 3.2 Results

I got the following runtimes for my code versions, using the "time" command in ubuntu:

### 3.2.1 First version

| Real | User | Sys |
|---|---|---|
| 49.227 | 49.145s | 0.072s |

### 3.2.2 Second version

| Real | User | Sys |
|---|---|---|
| 22.330s | 22.262s | 0.060s |

### 3.2.3 Third and final version

| Real | User | Sys |
|---|---|---|
| 18.186s | 18.177s | 0.000s |

### 3.2.4 Complexity analysis

It was hypothesized that this algorithm has a complexity $\mathcal{O}(N^2)$. This means that runtime as a function of input N should scale as follows:

$$t(N) = c * N^2 \tag{2}$$

Where c is a constant that depends on factors outside of the algorithm, such as hardware. If the algorithm adheres to this complexity, then we would expect to see the following increase in time each time N is doubled:

$$\frac{t(2N)}{t(N)} = \frac{c * (2N)^2}{c * N^2} = 4 \tag{3}$$

meaning that time increases with a factor of 4 when N is doubled. When running the time command in ubuntu for N = 1000, 2000, 4000 and 8000 i got the following results:

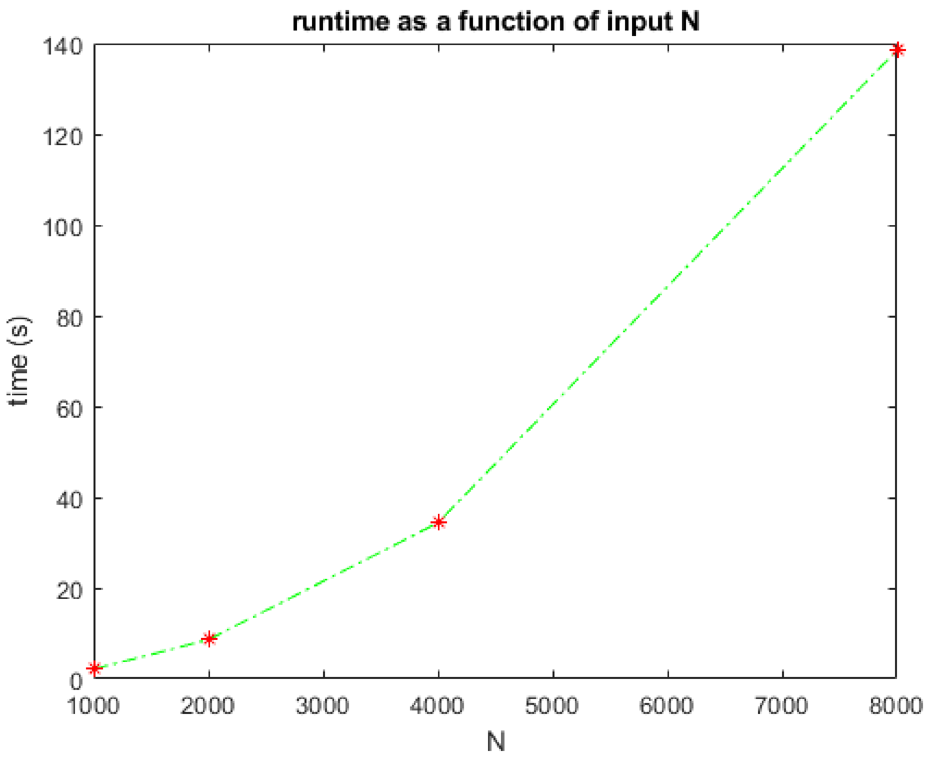

Figure 3: Results from complexity analysis

thus we can confirm that the algorithm does in fact have a complexity of $\mathcal{O}(N^2)$.

## 3.3 Discussion

## 3.4 First version

The first code that performed the worst did so as a result of bad implementation of data structures and storing. Since function update_stars is passed the entire array of structs including the values that may not be necessary, such as brightness, this caused a lot of unnecessary memory usage which slowed the program down significantly.

## 3.5 Second version

In my second version I instead opted to send the structs as pointers which almost halved the runtime. There was still some problems with data locality, since each pointer pointed to an entire struct, and it also became intuitively complicated to handle the pointers, since each pointer had to be iterated through the array of pointers continuously in order to iterate to the next star. In this version I also changed the function update_stars, specifically the inner loop. Since the gravitational force between two objects is symmetrical, that means that we can use the force calculation from j to i to also update the contribution to change in velocity and position from i to j. This means that we never have to calculate the gravitational force between to objects more than once, and the number of iterations needed is greatly reduced.

## 3.6 Third and final version

The problem with the struct in the previous version was solved in the final version by converting the structure from an array of structs (AoS) to a struct of arrays (SoA). This meant that in the second version when operations of multiple attributes of the same struct was performed, the memory access pattern needs to "jump around" which can lead to poor cache locality and decrease performance. With a single struct, where each attribute is an array holding that specific attribute for all stars, all attributes of the same type lies in sequential order and there is no need for pointers to jump around in memory. Ultimately it did not give a significant improvement, but reduced the runtime by about 2-3 seconds nonetheless. Furthermore, I initialized new variables from an expression wherever that expression was used more than once, so as to reduce the number of operations needed. I also initialized and assigned values to all variables that does not depend on i or j outside the loops, as well as initializing variables that only depends of i inside this loop, but outside the j loop to keep the inner loop as small as possible since this is usually good practice.

I considered using a hard rule that would stop calculations of forces between two stars if they were too far apart to have a significant force between them, although that may beat the purpose of using the double data type. I tried to analyze to see if any forces equalled zero from some rounding error, but with the double data type no forces was detected to be small enough for it not being able to be stored in a double type. I did not consider inlining since the only function in this code is relatively large with loops that would outweigh the performance decrease for that of the overhead of calling the function. I did not figure out any way to break the function down in to smaller chunks to inline the smaller functions instead. this code could be improved through parallelization, but this assignment was limited to serial codes only.

It should also be added that the time measurements were done at different times of the day, and it seemed that I got better results when I ran the final code at around 04.00 (the time presented in this report) in the morning compared to around 13.00 (about +1 second in run time). This could perhaps be due to the system that was used is a shared distributed system where there are more active users during the day.

## 4 References

Initially I used the following code as starting point:
$https://github.com/LapuhRok/GalaxySimulation/blob/master/galsim.c$
although I ended up using very little from this code, apart from the function void updateForceBasic() which is very similar to my code, except that I applied certain optimizations and declared variables within their respective scope, as well as using pointers to update the stars velocity and position.

## 5 My final code

```
#include <stdio.h>
#include <stdlib.h>
#include "graphics.h"
#include <math.h>

/* Define struct for a celestial body */
typedef struct {
        double* xp; //X Position
        double* yp; //Y Position
        double* mass;
        double* xv; //X Velocity
        double* yv; //Y Velocity
        double* brightness; //Keep, but not used in this simulation
} cbody_t;

const float circleRadius=0.0025, circleColor=0;
const int windowWidth=800;
const double eps = 0.001;
//int check_counter = 0;

/* Calculates and updates for new velocity and position of single star */
void update_stars(cbody_t *stars, const double G, const int N, const double dt){
        double *xp = stars->xp;
        double *yp = stars->yp;
        double *mass = stars->mass;
        double *xv = stars->xv;
        double *yv = stars->yv;
        const double Gdt = G*dt;
        for (int i = 0; i < N; i++) {
                double im = mass[i];
                double ix = xp[i];
                double iy = yp[i];
                double axi = 0;
                double ayi = 0;
                for (int j = i+1;j < N; j++) {
                        double mj = mass[j];
                        double rx = ix - xp[j];
                        double ry = iy - yp[j];
                        double rsqrt = sqrt(rx*rx + ry*ry) + eps;
                        double inv_r = 1.0/(rsqrt*rsqrt*rsqrt);
                        double inv_r_mj = inv_r*mj;
                        axi += rx*inv_r_mj;
                        ayi += ry*inv_r_mj;
                        xv[j] += Gdt*im*inv_r*rx;
                        yv[j] += Gdt*im*inv_r*ry;
                }
                xv[i] += -Gdt*axi;
                yv[i] += -Gdt*ayi;
                xp[i] += xv[i]*dt;
                yp[i] += yv[i]*dt;
        }
}

int main(int argc, char const *argv[]) {
        /* Make sure input arguments are correct */
        if (argc != 6) {
                printf("Error: Correct inputs: ./galsim N filename nsteps delta_t
                return 1;
            }

        /* Save initial arguments and create the stars objects */
        const int N = atoi(argv[1]);
        const char *filename = argv[2];
        const int nsteps = atoi(argv[3]);
        const double dt = atof(argv[4]);
        const int graphics = atoi(argv[5]);
        int count = 0;
        const double G = 100/(double)N;

        //double G = 1.0;
        printf("N = %d, filename = %s, nsteps = %d, delta_t = %lf, graphics = %d\
        //CBody stars[N];
        cbody_t stars;
        stars.xp = malloc(N * sizeof(double));
        stars.yp = malloc(N * sizeof(double));
        stars.mass = malloc(N * sizeof(double));
        stars.xv = malloc(N * sizeof(double));
        stars.yv = malloc(N * sizeof(double));
        stars.brightness = malloc(N * sizeof(double));

        /* Read the file */
        FILE *file = fopen(filename, "r");
        if (file == NULL) {
                printf("Error: failed to open the file %s\n", filename);
                return 1;
        }

        /* Input initial conditions into stars */
        /*for (i = 0; i < N; i++) {
                fread(&stars[i], sizeof(CBody), 1, file);
        }*/

        for (int i = 0; i < N; i++) {
                //cbody_t *new_star = &stars[i];
                //fread(, sizeof(cbody_t), 1, file);
                fread(&stars.xp[i], sizeof(double),1,file);
                fread(&stars.yp[i], sizeof(double),1,file);
                fread(&stars.mass[i], sizeof(double),1,file);
                fread(&stars.xv[i], sizeof(double),1,file);
                fread(&stars.yv[i], sizeof(double),1,file);
                fread(&stars.brightness[i], sizeof(double),1,file);
        }

        fclose(file); //Don't need file anymore

        /* print values to check its working */
        for (int i = 0; i < N; i++) {
                printf("Star #%d\n", i+1);
                printf("xp = %.15lf\n", stars.xp[i]);
                printf("yp = %.15lf\n", stars.yp[i]);
                printf("xv = %.15lf\n", stars.xv[i]);
                printf("yv = %.15lf\n", stars.yv[i]);
```

```
                printf("brightness = %.15lf\n", stars.brightness[i]);
                printf("mass = %.15lf\n", stars.mass[i]);
        }

        if (graphics == 1){
                printf("Graphics on\n");
                float L=1, W=1;

                InitializeGraphics(argv[0],windowWidth,windowWidth);
                SetCAxes(0,1);

                printf("Hit q to quit.\n");
                while(count < nsteps) {
                        /* Update star position and velocity */
                        update_stars(&stars, G, N, dt);
                        /* Call graphics routines. */
                        ClearScreen();
                        for (int k = 0; k < N; k++) {
                                DrawCircle(stars.xp[k], stars.yp[k], L, W, circleR
                        }
                        Refresh();
                        /* Sleep a short while to avoid screen flickering. */
                        usleep(3000);
                        count++;
                }
                FlushDisplay();
                CloseDisplay();
        } else {
                printf("Graphics turned off\n");
                /*for (i = 0; i < N; i++) {
                         stars[i].xp += delta_t*stars[i].xv;
                         stars[i].yp += delta_t*stars[i].yv;
                 }*/
                while (count < nsteps) {
                        update_stars(&stars, G, N, dt);
                        count++;
                }/*
                 for (i = 0; i < N; i++) {
```

```
                                stars[i].xp += delta_t*stars[i].xv;
                                stars[i].yp += delta_t*stars[i].yv;
                        }*/
        }
        //printf("HERE COMES COUNT");
        //printf("Count = %d\n", count);
        /* Write result to new file */
        FILE *new_file = fopen("output.gal", "wb");
        //cbody_t *starPointer = stars;
        for (int i = 0; i < N; i++){
                fwrite(&stars.xp[i],sizeof(double),1,new_file);
                fwrite(&stars.yp[i],sizeof(double),1,new_file);
                fwrite(&stars.mass[i],sizeof(double),1,new_file);
                fwrite(&stars.xv[i],sizeof(double),1,new_file);
                fwrite(&stars.yv[i],sizeof(double),1,new_file);
                fwrite(&stars.brightness[i],sizeof(double),1,new_file);
        }
            fclose(new_file);
        free(stars.xp);
        free(stars.yp);
        free(stars.mass);
        free(stars.xv);
        free(stars.yv);
        free(stars.brightness);
            return 0;
}
```