Numerical Method in Soft Matter

Homework Assignments

Anna Garbo

September 29, 2024

Contents

1 A1. Sampling

Here I report the code used to generate data and graphs to show how data are distributed in space.

To sample random points within D-dimensional domains by hit and miss, data has been sampled using C programming language and then visualised through python programs.

1.1 Sampling random points within D-dimensional domains by hit and miss

1.1.1 Rectangle

In this code, points have been generated in a specific interval using uniform distribution.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int numThrows = 10000000;
    int numHits = 0;
    double a = 1.0;
    double b = 3.0;
    double c = 2.0;
    double d = 5.0;
    srand(12345);

    FILE *file = fopen("hit_miss.csv", "w");
    if (file == NULL) {
        perror("Errore nell'apertura del file");
        return 1;
    }
    for (int i = 0; i < numThrows; i++) {
        double x = (double)rand() / RAND_MAX * (b - a) + a;
        double y = (double)rand() / RAND_MAX * (d - c) + c;
        if (x >= a && x <= b && y >= c && y <= d && r <= a) {
            numHits++;
        }
        fprintf(file, "%.6lf %.6lf /n", x, y);
    }
    fclose(file);
    printf("%d\n", numHits);
    double estimatedArea_rec = (double)numHits / numThrows * (b - a) * (d - c);
    printf("Estimated Area Rectangle: %lf\n", estimatedArea_rec);
    printf("Analytic Area Rectangle: %lf\n", (b - a) * (d - c));

    return 0;
```

```
}
```

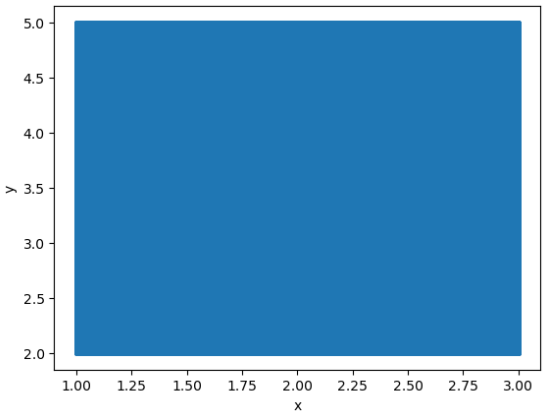


Figure 1: Rectangle of Area $A = L_{ab}L_{cd}$

1.1.2 Circle

In this code points have been generated between -1 and 1 to simulate the unit circle. To avoid the inverse sampling method considering a distribution different from the uniform one, the condition to build the circle is $x^2 + y^2 < 1$.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

int main() {
    srand(12345);
    int num_punti = 1000;
    FILE *file = fopen("circle.csv", "w");
    if (file == NULL) {
        perror("Errore nell'apertura del file");
        return 1;
    }
    fprintf(file, "x,y\n");
    for (int i = 0; i < num_punti; i++) {
        double x = (double)rand() / RAND_MAX * 2 - 1;
        double y = (double)rand() / RAND_MAX * 2 - 1;
        if (x * x + y * y <= 1) {
            fprintf(file, "%lf,%lf\n", x, y);
        }
    }
```

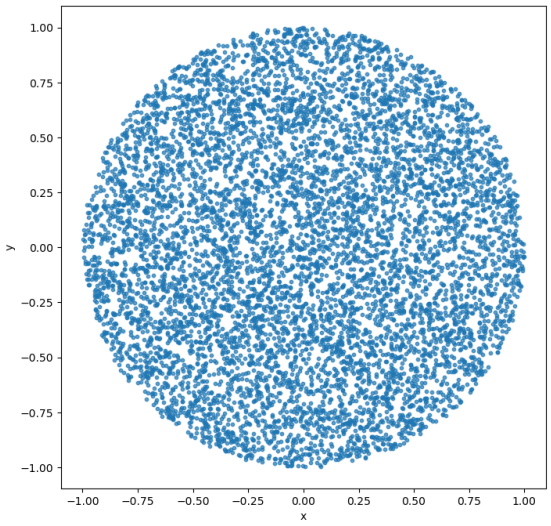


Figure 2: Random points generated in a unitary radius circle.

```
    fclose(file);
     return 0;
}
```

Sampling random numbers from a given distribution

1.2 Inversion Method I

1.2.1 Sampling from $\rho(x) = cx^n$ $n = 3, 4$ $x \in [0, 1]$

$\rho(x) = cx^3$

- Find the constant c using the PDF costraint

$$\int_0^1 \rho(x)dx = 1 \ \Rightarrow c = 4 \tag{1}$$

- Compute the CDF

$$F(x) = \int_0^x \rho(t)dt = x^4 \tag{2}$$

- Generate random points using Cumulative Density Function

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
```

```
int main() {
    srand(12345);
    int num_samples = 1000000;
    FILE *file = fopen("power_law_points_3.csv", "w");
    if (file == NULL) {
        perror("Errore nell'apertura del file");
        return 1;
    }
    for (int i = 0; i < num_samples; i++) {
        double u =(double)rand() / RAND_MAX;
        double x = pow(u, 1.0 / 4.0);
        fprintf(file, "%lf\n", x); }
    fclose(file);
    return 0;
}
```

- Verify the distribution with python graphic tools.

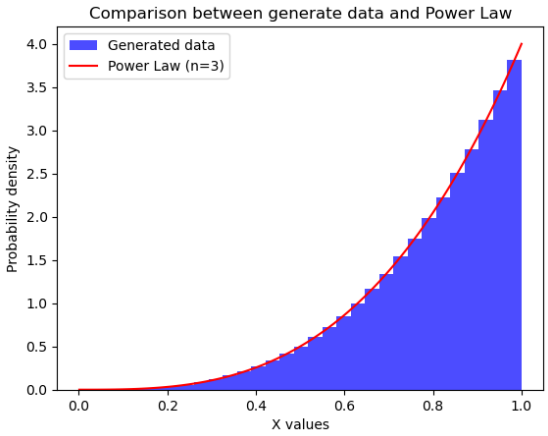


Figure 3: Inverse Sampling Method for PDF $\rho(x) = 4x^3$

$\rho(x) = cx^4$

- Find the constant c using the PDF costraint

$$\int_0^1 \rho(x)dx = 1 \;\Rightarrow c = 5 \tag{3}$$

- Compute the CDF

$$F(x) = \int_0^x \rho(t)dt = x^5 \tag{4}$$

- Generate random points using Cumulative Density Function

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

int main() {
    srand(12345);
    int num_samples = 1000000;
    FILE *file = fopen("power_law_points_4.csv", "w");
    if (file == NULL) {
        perror("Errore nell'apertura del file");
        return 1;
    }
    for (int i = 0; i < num_samples; i++) {
        double u =(double)rand() / RAND_MAX;
        double x = pow(u, 1.0 / 5.0);
        fprintf(file, "%lf\n", x); }
    fclose(file);
    return 0;
}
```

- Verify the distribution with python graphic tools.

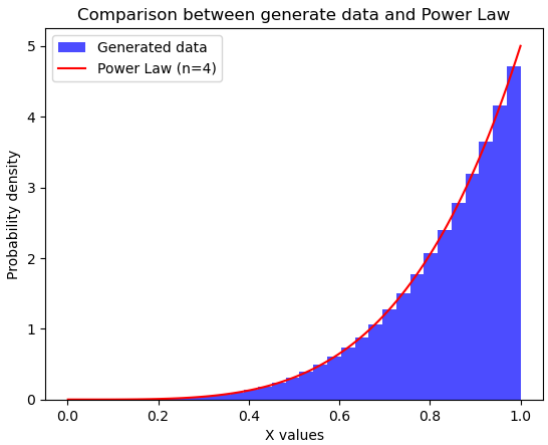


Figure 4: Inverse Sampling Method for PDF $\rho(x) = 5x^4$

1.3 Inverion Method II

Sampling from $\rho(x) = cx^2$ $x \in [0, 2]$

- Find the constant c using the PDF costraint

$$\int_0^2 \rho(x)dx = 1 \ \Rightarrow c = \frac{3}{8} \tag{5}$$

- Compute the CDF

$$F(x) = \int_0^x \rho(t)dt = \frac{x^3}{8} \tag{6}$$

- Generate random points using Cumulative Density Function

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <time.h>
    #include <math.h>
int main() {
    srand(12345);
    int num_samples = 1000000;
    FILE *file = fopen("power_law_points_2.csv", "w");
    if (file == NULL) {
        perror("Errore nell'apertura del file");
        return 1;
    }
    for (int i = 0; i < num_samples; i++) {
        double u =(double)rand() / RAND_MAX;
        double x = 2*pow(u, 1.0 / 3.0);
        fprintf(file, "%lf\n", x); }
    fclose(file);
    return 0;
}
```

- Verify the distribution with python graphic tools.

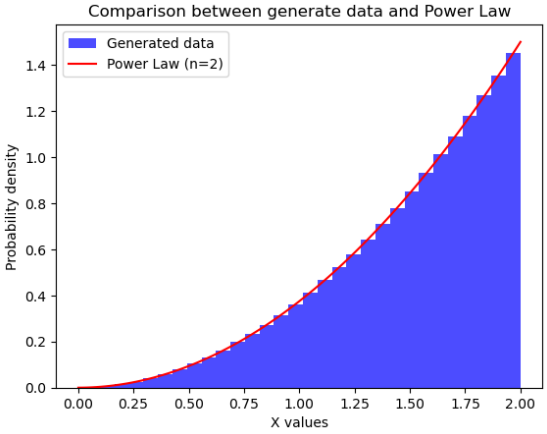


Figure 5: Inverse Sampling Method for PDF $\rho(x) = \frac{3}{8}x^2$

2 A.2 Sampling via transformation of coordinates

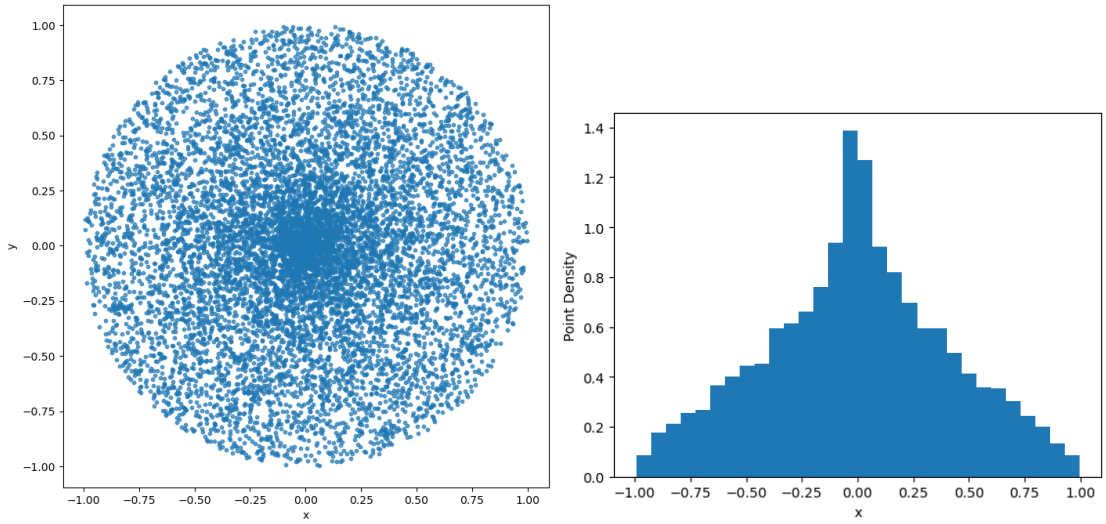
2.1 Sampling uniformly points within a unit radius disk

The procedure was the same as previous mentioned. Data has been generated using C compiler and then visualised with python compiler. As suggested in the exercise delivery in a first moment the random point in the unitary disk has been generated using $r = \xi_1$ $\theta = 2\pi\xi_2$ with ξ_1, ξ_2 two uniform generated number in [0,1]. This is the result.

- Code

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

double random_uniform() {
    return (double)rand() / RAND_MAX;
    }
void sample_points_in_unit_disk(double* x, double* y) {
    double r, theta;
    double u1 = random_uniform();
    r = u1;
    double u2 = 2.0 * M_PI * random_uniform();
    theta = u2;
    *x = r * cos(theta);
    *y = r * sin(theta);
}

int main() {
    FILE *file = fopen("coord.csv", "w");
if (file == NULL) {
     perror("Errore nell'apertura del file");
 return 1;
}
int num_samples = 10000;
double x, y;
srand(1234);
for (int i = 0; i < num_samples; i++) {
    sample_points_in_unit_disk(&x, &y);
    fprintf(file, "%lf,%lf\n", x, y);
}
return 0;
}
```

- Data Visualization and Data Distribution



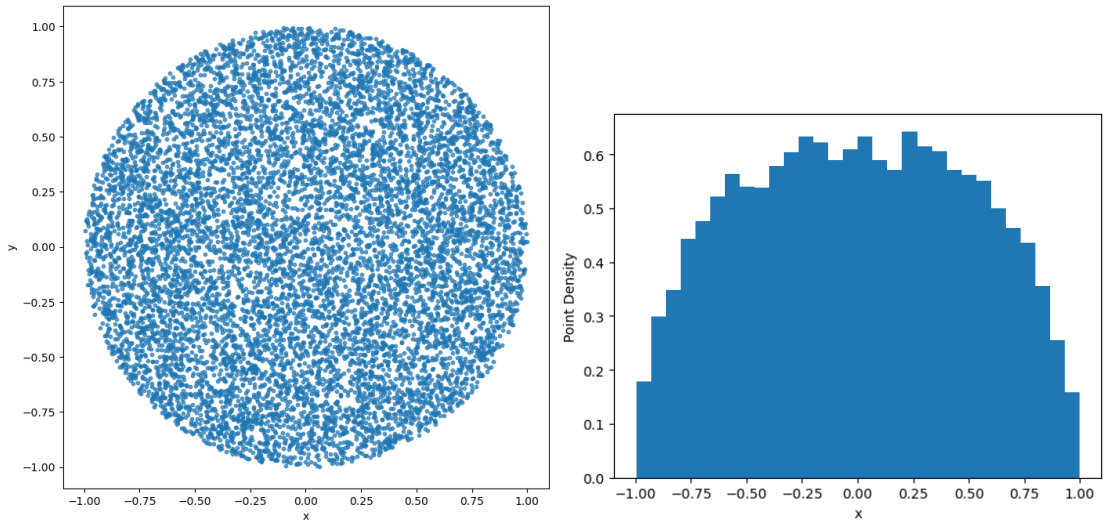
(a) Data Visualization

(b) Data Distribution

Figure 6: Data generated within a unit radius circle considering r and θ uniformly distributed.

The principle reason why this algorithm does not generated data uniformly in a unit radius disk is that r gives an higher probability for points near the center of the disk instead of points more external. This fact give the hints to find a probability density that represents this variability. Indeed I took $r = \sqrt{\xi_1}$ because this is thought as the root square of a ring with radius r. Essentially, the choice of $r = \sqrt{\xi_1}$ takes into consideration the probability distribution associated with the area in polar coordinates, ensuring uniform sampling inside the disk. This balances the probability so that points located farther from the center of the disk have a higher probability of being sampled than they would with a simple $r = \xi_1$. This is the result.

- Code: Same as mentioned before with the difference in taking $r = \sqrt{\xi_1}$.
- Data Visualization and Data Distribution.



(a) Data Visualization (b) Data Distribution

Figure 7: Data generated within a unit radius circle considering $r = \sqrt{\xi_1}$ and θ uniformly distributed.

2.2 Box-Muller Transformation

Box-Muller Transformation is a way to generate numbers from a 2D Gaussian Distribution $\mathcal{N}(0, 1)$. Starting from:

$$\rho(x, y) = \frac{1}{2\pi} e^{-\frac{(x^2+y^2)}{2}} = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \frac{1}{\sqrt{2\pi}} e^{-\frac{y^2}{2}} \tag{7}$$

Changing Coordinates

$$x = r\cos(\theta) \tag{8}$$
$$y = r\sin(\theta) \tag{9}$$

The equation becomes:

$$\rho(r) = \frac{1}{2\pi} e^{-\frac{r^2}{2}} \tag{10}$$

From which we can extract

$$r \sim \sqrt{-2\ln(u1)} \tag{11}$$
$$\theta = 2\pi u2 \tag{12}$$

where $u1$ and $u2$ are random variables uniformly distributed.

This takes to the way to sample data from a 2D Gaussian:

$$x = \sqrt{-2\ln(u1)} cos(2\pi u2) \tag{13}$$
$$y = \sqrt{-2\ln(u1)} sin(2\pi u2) \tag{14}$$

Under this analytical calculation, will be reported the code used to sample using Box-Muller Transformation and a graphic verification for one dimension.

```
    #include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>


void boxMullerTransform(double *z0, double *z1) {
    // Generate two random numbers uniformly distributed between 0 and 1
    double u1 = rand() / (RAND_MAX + 1.0);
    double u2 = rand() / (RAND_MAX + 1.0);

    // Apply the Box-Muller transformation to get standard normal variables
    *z0 = sqrt(-2.0 * log(u1)) * cos(2.0 * M_PI * u2);
    *z1 = sqrt(-2.0 * log(u1)) * sin(2.0 * M_PI * u2);
}

int main() {
    srand((unsigned)time(NULL));
    int numPairs = 1000;
    FILE *file = fopen("gauss_dist.csv", "w");
    if (file == NULL) {
        fprintf(stderr, "Error opening the file.\n");
        return 1;
    }
    fprintf(file, "Pair,Value1,Value2\n");
    for (int i = 0; i < numPairs; ++i) {
        double z0, z1;
        boxMullerTransform(&z0, &z1);
        fprintf(file, "%d,%f,%f\n", i + 1, z0, z1);
    }

    fclose(file);

    printf("Pairs of normally distributed random numbers written to gauss_dist.csv\n");

    return 0;
}
```

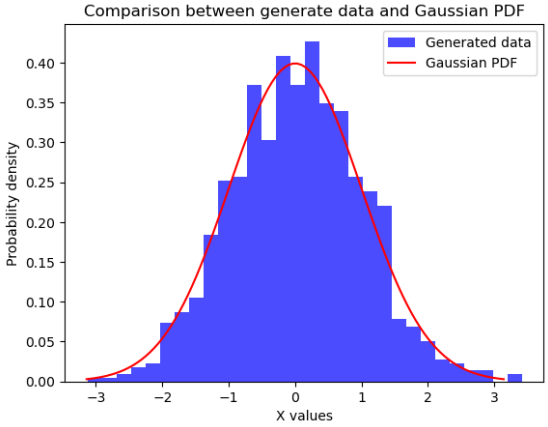


Figure 8: Data sampled from a Gaussian PDF.

For a given mean and sigma so for a distribution defined as $\mathcal{N}(\mu, \sigma^2)$, the algorithm can be generalized using z0 as a variable sampled from $\mathcal{N}(0, 1)$ and returning the variable $x = \mu + \sigma z0$, so shifting and scaling to desidered distribution. Code:

```
    double generateNormalRandomNumber(double mean, double stddev) {
    double u1 = rand() / (RAND_MAX + 1.0);
    double u2 = rand() / (RAND_MAX + 1.0);


    double z0 = sqrt(-2.0 * log(u1)) * cos(2.0 * M_PI * u2);
    // double z1 = sqrt(-2.0 * log(u1)) * sin(2.0 * M_PI * u2);

    // Scale and shift to get the desired normal distribution
    return mean + stddev * z0;
}
```

2.3 Rejection Method

The target function is the following:

$$f(x) = \frac{2}{\sqrt{\pi}} e^{-x^2} \tag{15}$$

and comparison function

$$g(x, p) = \begin{cases} A & \text{if } 0 < x \leq p \\ \frac{A}{p} x e^{p^2 - x^2} & \text{if } x > p \end{cases} \tag{16}$$

Following steps have been implemented:

- generate ξ, u as random number in $[0, 1]$ uniformly distributed where ξ represents the random variable used to compare target function and comparison function, while u is useful to sample x from $g(x)$.

- as second step, data have been sampled from $g(x)$. To do that, $A = \frac{2p}{2p^2+1}$ has been estimated as a function of p imposing $\int_{\mathbb{R}} g(x)dx = 1$; then the cumulative function $G(x)$ has been computed, giving the following analytical form:

$$G(x,p) = \begin{cases} \frac{2p}{2p^2+1}x & \text{if } 0 < x \leq p \\ \frac{2p^2}{2p^2+1} + \frac{1}{2p^1+1}(1 - \exp(p^2 - x^2)) & \text{if } x > p \end{cases} \tag{17}$$

  The inverse of $G(x)$ has been performed giving the following condition:

$$x(u,p) = \begin{cases} \frac{2p^2+1}{2p}u & \text{if } 0 < u \leq \frac{2p^2}{2p^2+1} \\ \sqrt{p^2 - \log(1 - (2p^2+1)u + 2p^2)} & \text{if } u > \frac{2p^2}{2p^2+1} \end{cases} \tag{18}$$

- after that the ratio between target function and comparison function has been performed: if $\xi \geq \frac{f(x)}{1.15g(x)}$ then new u and x have been sampled, in the other case x has been stored.

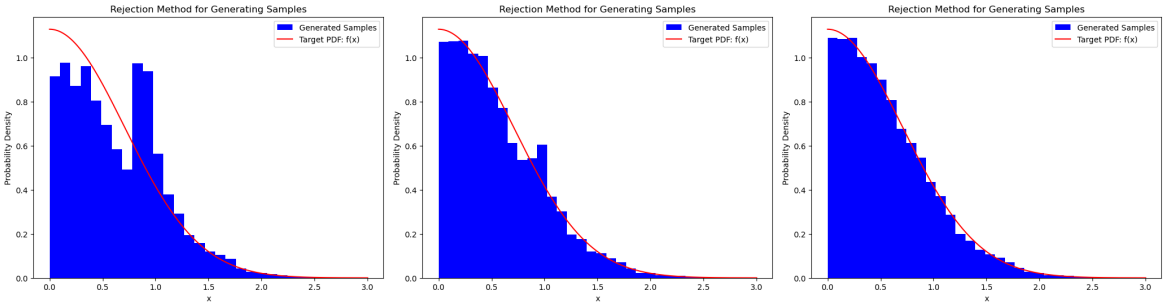
- to conclude data distribution has been visualised through python comparing different value of p. As we can observe in Fig.9 with the value $p = 0.7$ we do have the best perfomance, this is because with lower values the comparison function is not over the target function, so the rejection method doesn't work properly.

Code to generate sample:

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

double random_uniform() {
    return (double)rand() / RAND_MAX;
}

double target_pdf(double x) {
    return sqrt(2.0 / M_PI) * exp(-x * x);
}

double comparison_pdf(double x, double p) {

    if (x < p) {
        return 2*p/(2*p*p+1);
    } else {
        return 2/(2*p*p+1) * x * exp(p * p - x * x);
```

```
    }
}

int main() {
    FILE *file = fopen("reject1.csv", "w");
    if (file == NULL) {
        perror("Errore nell'apertura del file");
        return 1;
    }

    int num_samples = 10000;
    double x, y, u;
    double p = 0.3;
    double A=2*p/(2*p*p+1);
    printf("%f\n", A);
    srand(42);

    for (int i = 0; i < num_samples; i++) {
        while (1) {
            u = random_uniform();
            y = random_uniform();

            if (u < 2 * p / (2 * p * p + 1)) {
                x = (2 * p * p + 1) * u / (2 * p);
            } else {
                x = sqrt(p * p - log(1 - (2 * p * p + 1) * u + 2 * p * p));
            }

            if (y <= target_pdf(x) / (1.15*comparison_pdf(x, p))) {
                break;
            }
        }

        fprintf(file, "%f\n", x);
    }

    fclose(file);

    return 0;
}
```

Visualization of the resaluts for $p = 3, 5, 10$.



(a) $p = 0.3$ (b) $p = 0.5$ (c) $p = 0.7$

Figure 9: Data generated with rejection method using the comparison function (16) .

3 A.4 Importance Sampling

The important sampling principle show that there is a reasonable alternative way to calculate integrals using Montecarlo simulations. Here I report briefly the equations that clearly show the principle.

$$\int_{\mathbb{R}} f(x)dx = \int_{\mathbb{R}} \frac{f(x)}{g(x)} g(x)dx = \langle \frac{f(x)}{g(x)} \rangle_{g(x)} \tag{19}$$

Where we are considering a g(x) which respects the definition of probability density function, the ration between f(x) and g(x) does not diverge and $f(x) \neq g(x)$.

What is useful for Montecarlo approach to calculate integral is the possibility to use another function from which we can sample data and calculate the integral.

3.1 Importance Sampling of a gaussian like distribution

The following function was given:

$$f(x) = g(x)e^{-x^2} \tag{20}$$

where the condition for $g(x)$ was to be slow varying so I choose $g(x) = \sqrt{x}$. The requested calculation was:

$$I = \int_0^\infty f(x)dx \tag{21}$$

The comparison between the crude montecarlo and importance sampling with $W(x) = \sqrt{\frac{2}{\pi}} e^{-x^2}$ for which the integral I become:

$$I \sim \frac{1}{N} \sum_i^N \frac{f(x_i)}{W(x_i)} = \frac{1}{N} \sqrt{\frac{2}{\pi}} \sum_i^N g(x_i) \tag{22}$$

where x_i are the point generated from $W(x)$ which, in this case is a Gaussian, so I can use Box-Muller transform method to sample from a Gaussian. Code:

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <math.h>
    double f(double x) {
        return exp(-x * x) * sqrt(x);
        }
    double W(double x) {
        return sqrt(2.0 / M_PI) * exp(-x * x);
        }
    double random_gaussian() {
        double x;
        double u1 = (double)rand() / RAND_MAX;
        double u2 = (double)rand() / RAND_MAX;
        x = sqrt(-2 * log(u1)) * (sin(2 * M_PI * u2));
        return x;
        }

int main() {
    int num_samples = 100000;
    double integral_crude = 0.0;
    double integral_importance = 0.0;
    double a=0.0;
    double b=3.0;
    srand(42321);

    for (int i = 0; i < num_samples; i++) {
        double x = random_gaussian();
        double u = a+(b-a)*rand() /(double)RAND_MAX;
        printf("x %f\n",x);
        integral_importance += f(fabs(x))/W(x);
        integral_crude += f(u);
    }

    integral_importance =integral_importance*sqrt(M_PI/2)/ (num_samples*2);
    integral_crude =integral_crude *(b-a)/num_samples;

    printf("Importance sampling result: %f\n", integral_importance);
    printf("Crude method result: %f\n", integral_crude);

    return 0;
}

Importance sampling result: 0.646628
Crude method result: 0.616078
```

Importance sampling method is overestimating the result than the crude method. An important point to precise is that to calculate the result using Box-Muller method I had to divide the integral by 2 because it sampled also negative points of $\mathcal{N}(0,1)$ but $g(x) = \sqrt{x}$ does not accept negative values, so to skip the problem I took the modulus, which take two times the desired values, this is the reason why I divided by two. Another point that I think it is right to specify is the fact i choose 3 as interval to calculate the crude Montecarlo integral, this choice is due to the function I choosed, indeed plotting $f(x) = \sqrt{x}e^{-x^2}$ one can see that is different from 0 between 0 and some point near 3, so this is the reason of this choice.

3.2 Impotance Sampling for the Integral $I = \int_0^{\frac{\pi}{2}} cos(x)dx$

The analytical result for this integral is trivial:

$$I = \int_0^{\frac{\pi}{2}} cos(x)dx = 1 \tag{23}$$

So now what I have to show is that the same result as approximately obtainable with importance sampling method. The exercise suggested to use the function $g(x) = a+bx^2$. First a and b has been choosen: $a = 1$ because $cos(0) = 1$ and I'm looking for a function that take into account the important values of the funtion in the considered interval $[0, \frac{\pi}{2}]$; $b = \frac{24}{\pi^3}(1 - \frac{\pi}{2})$ because i looked for a value for which the integral of $g(x)$ is 1. The analytical form for $g(x)$ is the following:

$$g(x) = 1 + \frac{24}{\pi^3}(1 - \frac{\pi}{2})x^2 \tag{24}$$

Then I started applying the importance sampling method, the following steps have been implemented:

- Fix the importance weight $w(x) = \frac{cos(x)}{g(x)}$.
- Generate points from g(x), since G(x), the CDF of g, is not invertible, rejection method has been used. The comparison function taken her is the one used for rejection method, for PDF I refer to eq.16, form cumulative of comparison function eq.17 and for cumulative inverse eq.18. Even if one can show that eq.24 is invertible in the interval $[0, \frac{\pi}{2}]$ the inverse of a cubic function does not present clear so I preferred to choose the non invertible cumulative function case and build the algorithm through rejection method.

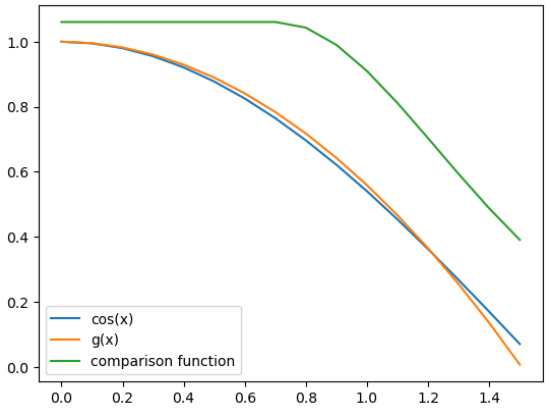


Figure 10: Comparison with three function used to build the algorithm, what is important to show is that the function from which I generate the samples are lower than the compared function.

- Estimate the integral

$$I \sim \frac{1}{N} \sum_{i=1}^{N} w(x_i) \tag{25}$$

  where x_i are the points generated from eq.24.

- Fix the accuracy to 1% taking a convergence approach, i.e. considering the difference with the integral calculated in iteration before and adding iterations until we reach the accuracy o 1%.

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define PI 3.14159265358979323846

double random_uniform() {
    return (double)rand() / RAND_MAX;
}

double target_function(double x) {
    return cos(x);
}

double sampling_distribution(double x) {
```

```
    return 1 + 24 / (PI * PI * PI) * (1 - PI / 2) * x * x;
}

double importance_weight(double x) {
    return cos(x) / sampling_distribution(x);
}

double comparison_pdf(double x, double p) {
    if (x < p) {
        return 2*p/(2*p*p+1);
    } else {
        return 2/(2*p*p+1) * x * exp(p * p - x * x);
    }
}

int main() {
    srand(123456789);

    int num_iterations = 0;
    double integral_estimate = 0.0;
    double accuracy_threshold = 0.01;
    double exact_value = 1.0;
    double previous_estimate = 0.0;
    double p=0.7;
    do {
        double integral_sum = 0.0;


        for (int i = 0; i < 1000; ++i) {
            double x, u, weight, y;

            while (1) {
                u = random_uniform();
                y = random_uniform();

                if (u < 2 * p / (2 * p * p + 1)) {
                    x = (2 * p * p + 1) * u / (2 * p);
                } else {
                    x = sqrt(p * p - log(1 - (2 * p * p + 1) * u + 2 * p * p));
                }

                if (y <= target_function(x) / (1.15*comparison_pdf(x, p))) {
                    break;
                }
```

```
                weight=importance_weight(x);
            }
            integral_sum += weight;
        }

        // Incremento del numero di iterazioni
        num_iterations++;

        integral_estimate += integral_sum / 1000;

        if (num_iterations > 1) {
            double relative_difference = fabs((integral_estimate - previous_estimate) /
            previous_estimate);
            if (relative_difference < accuracy_threshold) {
                break;
            }
        }

        // Aggiornamento della stima precedente
        previous_estimate = integral_estimate;

    } while (1);

    printf("Valore stimato dell'integrale: %f\n", integral_estimate/num_iterations);
    printf("Numero di iterazioni: %d\n", num_iterations);


    double relative_difference = fabs((integral_estimate/num_iterations - exact_value)
    / exact_value);
    printf("Differenza Relativa rispetto al valore esatto: %f\n", relative_difference);

    return 0;
}
Valore stimato dell'integrale: 0.982048
Numero di iterazioni: 40
Differenza Relativa rispetto al valore esatto: 0.017952
```

I reached the accuracy of 2% which is acceptable.

3.3 Importance Sampling of a Heaviside function

In this case Importance Sampling has beel performed for the function:

$$f(x) = \begin{cases} 1 & \text{if } x > T \\ 0 & \text{otherwise} \end{cases} \tag{26}$$

We want to compute the average

$$\langle f(x) \rangle_{\rho(x)} = \int_{\mathbb{R}} f(x)\rho(x)dx \tag{27}$$

with respect to the PDF $\rho(x) = e^{-x}$. Since the PDF does not respect the varying of f(x) it is suggested to compute the integral via importance samplin method using $g(a, x) = ae^{-ax}$.

First of all I will show the analytical results requested in the exercise:

- Mean Analytical Value of the function

$$\begin{aligned} \langle f(x) \rangle_{\rho(x)} &= \int_{\mathbb{R}} f(x)\rho(x)dx \\ &= \int_{T}^{+\infty} e^{-x}dx \\ &= -[e^{-x}]_{T}^{+\infty} \\ &= e^{-T} \end{aligned}$$

- Analytical estimation of variance. Since

$$\langle f(x)^2 \rangle_{\rho(x)} = \langle f(x) \rangle_{\rho(x)}$$

  Then

$$\begin{aligned} \sigma^2(f) &= \langle f(x)^2 \rangle_{\rho(x)} - \langle f(x) \rangle^2_{\rho(x)} \\ &= \langle f \rangle_{\rho}(1 - \langle f \rangle_{\rho}) \end{aligned}$$

- What is requested is the variance for $F = \frac{f\rho}{g}$. It is defined as

$$\begin{aligned}
\sigma^2(a, F(x)) &= \langle F(x)^2 \rangle_{g(x)} - \langle F(x) \rangle^2_{g(x)} \\
&= \langle \frac{f^2(x)\rho^2(x)}{g^2(x)} \rangle_{g(x)} - \langle \frac{f(x)\rho(x)}{g(x)} \rangle^2_{g(x)} \\
&= \int_{\mathbb{R}} \frac{f^2(x)\rho^2(x)}{g^2(x)} g(x) dx - \langle f(x) \rangle^2_{\rho(x)} \\
&= \int_{\mathbb{R}} \frac{f^2(x)\rho^2(x)}{g(x)} dx - e^{-2T} \\
&= \int_T^{+\infty} \frac{e^{-2x}}{ae^{-ax}} dx - e^{-2T} \\
&= \frac{1}{a} \int_T^{+\infty} e^{x(a-2)} - e^{-2T} \\
&= \frac{1}{a} \frac{1}{a-2} [e^{x(a-2)}]_T^{+\infty} - e^{-2T} \\
&= \frac{1}{a} \frac{1}{a-2} [e^{-x(2-a)}]_T^{+\infty} - e^{-2T} \\
&= \frac{1}{a} \frac{1}{a-2} (-) e^{-T(2-a)} - e^{-2T} \\
&= \frac{1}{a} \frac{1}{2-a} e^{-T(2-a)} - e^{-2T} \\
&= \frac{e^{-T(2-a)}}{a(2-a)} - e^{-2T}
\end{aligned}$$

Then as a second step I found a that minimize the variance, which results to be:

$$a_{min} = 1 + \frac{1}{T} - \sqrt{1 + \frac{1}{T^2}} \tag{28}$$

At the end I verify the statistical efficiency comparing the results from different T. For example $T = 3, 5, 10, 20$ performing different statistical tests such as $\frac{\sigma(f)}{\langle f \rangle_\rho}$, $\sigma(F(a*, x))$ and $\frac{\sigma(f)}{\sigma(F(a*,x))}$.

Code:

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

double random_uniform() {
    return (double)rand() / RAND_MAX;
}

double f(double x, double T) {
```

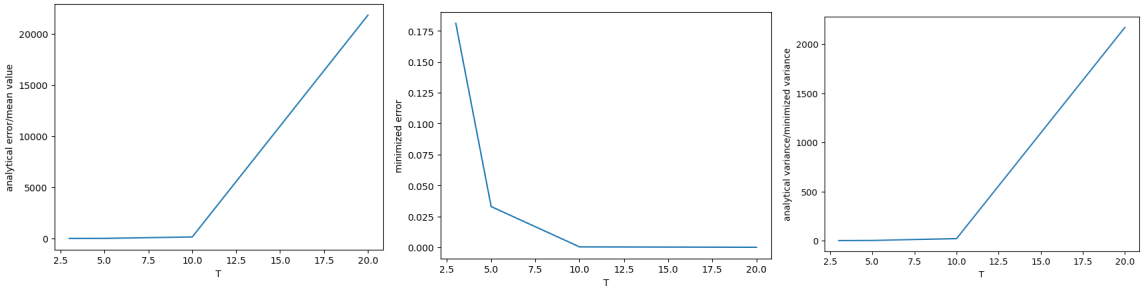
```
    if (x > T) {
        return 1.0;
    } else {
        return 0.0;
    }
}
double roh(double x){
    return exp(-x);
}
double g(double x, double a) {
    return a * exp(-a * x);
}

double cumulative_inv(double x, double a) {
    return -1 / a * log(1 - x);
}

int main() {
    int num_samples = 100000;

    double integral_importance = 0.0;
    double T = 20.0;
    double a = 1 + 1 / T - sqrt(1 + 1 / (T * T));

    srand(42321);

    for (int i = 0; i < num_samples; i++) {
        double u = random_uniform();

        double x = cumulative_inv(u, a);
        integral_importance += f(x, T)*roh(x) / g(x, a);
    }


    integral_importance = integral_importance / num_samples;

    printf("T: %f\n",T);
    printf("Importance sampling result: %.10f\n", integral_importance);
    printf("Expected Mean Value: %.10f\n", exp(-T));
    printf("Relative Error %.10f\n", fabs(integral_importance - exp(-T)) / exp(-T));
    printf("Analitycal Error %.10f\n", exp(-T)*(1-exp(-T)));
    return 0;
}
Importance sampling result: 0.006815
```

```
Expected Mean Value: 0.006738
Relative Error 0.011467
Analitycal Error 0.006693
```

Results:

T	Importance Sampling	Mean Value	Relative Error	Analytical Error	Minimized Variance
3	5×10^{-2}	4×10^{-2}	6×10^{-3}	4×10^{-2}	3×10^{-2}
5	6×10^{-3}	6×10^{-3}	1.1×10^{-3}	6×10^{-3}	1×10^{-3}
10	5×10^{-5}	4×10^{-5}	1.1×10^{-3}	4×10^{-5}	1×10^{-7}
20	2×10^{-9}	2×10^{-9}	2.3×10^{-3}	2×10^{-9}	4×10^{-16}

Table 1: Table with numerical results of the integral estimated with importance sampling and analytically



(a) $\frac{\sigma(f)}{\langle f \rangle_\rho}$ (b) $\sigma(F(a*,x))$ (c) $\frac{\sigma(f)}{\sigma(F(a*,x))}$

Figure 11: Statistical tests .

4 A.5 Markov Chains

4.1 Recurrent Relation as a Master Equation

What is shown is here is that the Recurrent Relation:

$$\mu_n = \mu_{n-1} P \tag{29}$$

is equivalent to write:

$$\mu_n(j) = (1 - \sum_{i \in S, i \neq j} p_{ij}) \mu_{n-1}(j) + \sum_{i \in S, i \neq j} p_{ji} \mu_{n-1}(i) \tag{30}$$

4.1.1 Demonstration

Starting from eq.29 I can write:

$$\begin{aligned}
\mu_n(j) &= \sum_{i \in S} \mu_{n-1}(i) p_{ij} \\
&= \sum_{i \in S, i=j} \mu_{n-1}(i) p_{ij} + \sum_{i \in S, i \neq j} \mu_{n-1}(i) p_{ij} \\
&= \sum_{i \in S, i=j} p_{ji} \mu_{n-1}(i) + \sum_{i \in S, i \neq j} p_{ji} \mu_{n-1}(i) \\
&= p_{jj} \mu_{n-1}(j) + \sum_{i \in S, i \neq j} p_{ji} \mu_{n-1}(i) \\
&= (1 - \sum_{i \in S, i \neq j} p_{ij}) \mu_{n-1}(j) + \sum_{i \in S, i \neq j} p_{ji} \mu_{n-1}(i)
\end{aligned}$$

Where basically:

- in the first line the reccurent relation has been copied;
- in the second line the diagonal part has been divided from the rest of the matrix;
- in the third line the position vector μ and P has been inverted so the matrix has been transposed;
- in the fourth line for the diagonal part the sum is taken out because $i = j$;
- in the fifth line the diagonal has been written as one minus the rest of the matrix because it is a property of stochastic matrix, that the sum over all colomn in each row is equal to one.

cvd.

4.1.2 What does it say this equation?

This equation is similar to the master equation performed for a random walk. Indeed it express the relation between the evolution of states of the system in time and the probability to move from one state to another. Going more into details, left side of the equation relate to state j at instant t_n, while the right side of the equation is divided in two parts, the first sum express the probability to remain in the state j while the second part express the probability to move from state j to state i.

4.2 Markov Chains Graphs

4.2.1 Matrix 1

Give the following matrix:

$$P = \begin{bmatrix} 0 & 0.5 & 0.5 \\ 0.5 & 0 & 0.5 \\ 0.5 & 0.5 & 0 \end{bmatrix}$$

Is explained by the following graph:

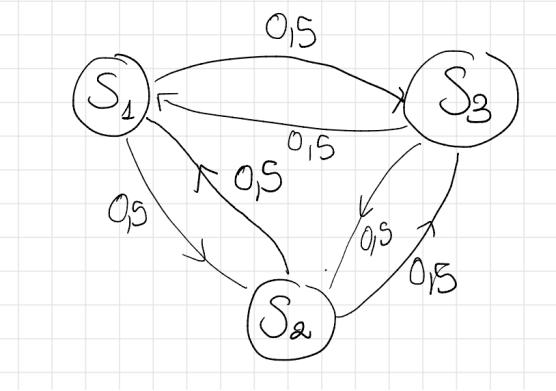


Figure 12: Graph for matrix 1

All of the three states are recurrent.

4.2.2 Matrix 2

Give the following matrix:

$$P = \begin{bmatrix} 0 & 0 & 0.5 & 0.5 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Is explained by the following graph:

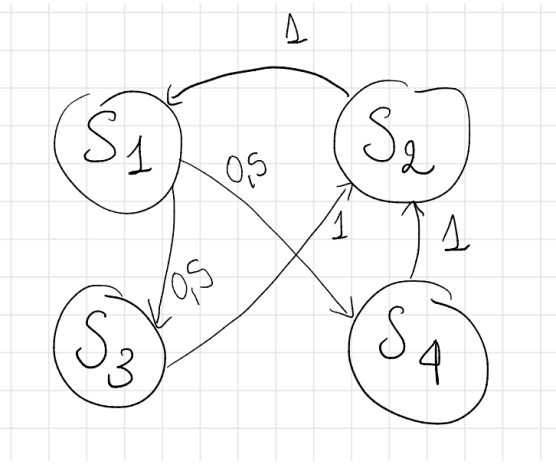


Figure 13: Graph for matrix 2

All of the four states are recurrent.

4.2.3 Matrix 3

Give the following matrix:

$$P = \begin{bmatrix} 0.3 & 0.4 & 0 & 0 & 0.3 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.6 & 0.4 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Is explained by the following graph:

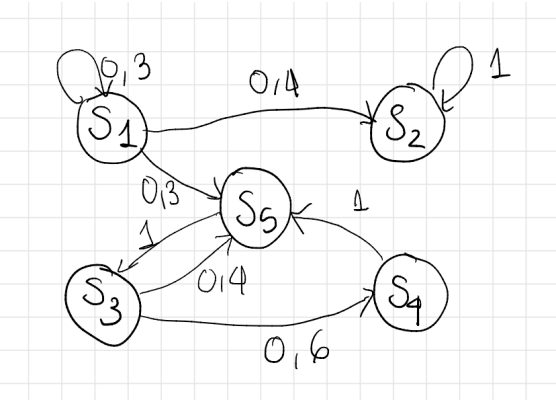


Figure 14: Graph for matrix 3

S1 and S2 are not recurrent, while S3,S4 and S5 are recurrent.

4.3 Irreducible Matrix

4.3.1 Matrix P1

$$P1 = \begin{bmatrix} 0.5 & 0.5 \\ 1 & 0 \end{bmatrix}$$

Code in python to compute P^n and the limit to infinity:

```
import numpy as np


P2 = np.array([[1/2, 1/2 ], [1,0]])

for n in range(1, 6):
    Pn = np.linalg.matrix_power(P2, n)
    print(f'P^{n}:\n{Pn}\n')
```

```
    limit_inf = np.linalg.matrix_power(P2, 100000)
    print(f'Limite mentre n ->               infinity:\n{limit_inf}\n')
Out:
P^1:
[[0.5 0.5]
 [1.  0. ]]

P^2:
[[0.75 0.25]
 [0.5  0.5 ]]

P^3:
[[0.625 0.375]
 [0.75  0.25 ]]

P^4:
[[0.6875 0.3125]
 [0.625  0.375 ]]

P^5:
[[0.65625 0.34375]
 [0.6875  0.3125 ]]

Limite mentre n -> infinity:
[[0.66666667 0.33333333]
 [0.66666667 0.33333333]]
```

4.3.2 Matrix P2

$$P2 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1/4 & 0 & 3/4 \end{bmatrix}$$

Code in python to compute P^n and the limit to infinity:

```
    import numpy as np


    P2 = np.array([[0, 0, 1], [0, 1, 0], [1/4, 0, 3/4]])

    for n in range(1, 6):
        Pn = np.linalg.matrix_power(P2, n)
        print(f'P^{n}:\n{Pn}\n')
```

```
    limit_inf = np.linalg.matrix_power(P2, 100000)
    print(f'Limite mentre n -> infinity:\n{limit_inf}\n')
Out:
P^1:
[[0.   0.   1.  ]
 [0.   1.   0.  ]
 [0.25 0.   0.75]]

P^2:
[[0.25   0.     0.75  ]
 [0.     1.     0.    ]
 [0.1875 0.     0.8125]]

P^3:
[[0.1875   0.       0.8125  ]
 [0.       1.       0.      ]
 [0.203125 0.       0.796875]]

P^4:
[[0.203125   0.         0.796875  ]
 [0.         1.         0.        ]
 [0.19921875 0.         0.80078125]]

P^5:
[[0.19921875 0.         0.80078125]
 [0.         1.         0.        ]
 [0.20019531 0.         0.79980469]]

Limite mentre n -> infinity:
[[0.2 0.  0.8]
 [0.  1.  0. ]
 [0.2 0.  0.8]]
```

4.4 Regular Matrix

4.4.1 Matrix1

$$P = \begin{bmatrix} 1/2 & 1/4 & 1/4 \\ 1/2 & 0 & 1/2 \\ 1/4 & 1/4 & 1/2 \end{bmatrix}$$

Taking the previous code one can show that

```
P^1:
```

```
[[0.5  0.25 0.25]
 [0.5  0.   0.5 ]
 [0.25 0.25 0.5 ]]

P^2:
[[0.4375 0.1875 0.375 ]
 [0.375  0.25   0.375 ]
 [0.375  0.1875 0.4375]]

P^3:
[[0.40625  0.203125 0.390625]
 [0.40625  0.1875   0.40625 ]
 [0.390625 0.203125 0.40625 ]]

P^4:
[[0.40234375 0.19921875 0.3984375 ]
 [0.3984375  0.203125   0.3984375 ]
 [0.3984375  0.19921875 0.40234375]]

P^5:
[[0.40039062 0.20019531 0.39941406]
 [0.40039062 0.19921875 0.40039062]
 [0.39941406 0.20019531 0.40039062]]

Limite mentre n -> infinity:
[[0.4 0.2 0.4]
 [0.4 0.2 0.4]
 [0.4 0.2 0.4]]
```

Then one can say that the matrix is regular since there is a k for which $(P^k)_{ij} > 0$ per ogni i,j.

4.4.2 Matrix2

Same procedure as before:

$$P1 = \begin{bmatrix} 1 & 0 \\ 0.5 & 0.5 \end{bmatrix}$$

```
    P^1:
[[1.  0. ]
 [0.5 0.5]]

P^2:
[[1.   0.  ]
```

```
 [0.75 0.25]]

P^3:
[[1.    0.   ]
 [0.875 0.125]]

P^4:
[[1.     0.    ]
 [0.9375 0.0625]]

P^5:
[[1.      0.     ]
 [0.96875 0.03125]]

Limite mentre n -> infinity:
[[1. 0.]
 [1. 0.]]
```

Then the matrix is not regular.

4.5 Stochastic Matrix with undefined $0 < p < 1$

$$P = \begin{bmatrix} p & 1-p & 0 & 0 \\ 1 & 0 & p & 1-p \\ p & 1-p & 0 & 0 \\ 0 & 0 & p & 1-p \end{bmatrix}$$

To show that is irreducible and aperiodic I used the graph I report here.

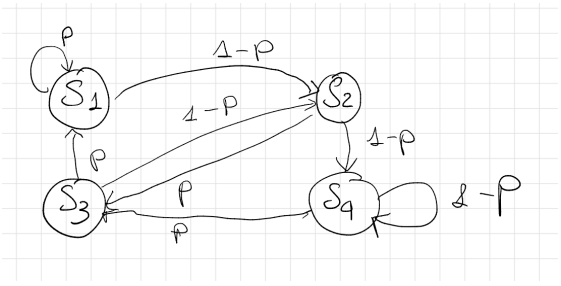


Figure 15: Graph for matrix P

From the graph we can observe that the matrix is irreducible for every $p \in [0, 1]$ Instead to find π compute the eigenvector on python and this is what I found.

```
import numpy as np
p=0.2
```

```
    P = np.array([[p, 1-p, 0, 0],
              [0, 0  , p, 1-p],
              [p, 1-p, 0, 0],
              [0, 0  , p, 1-p]
             ])


    eigenvalues, eigenvectors = np.linalg.eig(P.T)
    stationary_distribution = np.real(eigenvectors[:, 0] / np.sum(eigenvectors[:, 0]))

    print("Stationary Distribution:")
    print(stationary_distribution)
Out: Stationary Distribution:
    [0.04 0.16 0.16 0.64]
```

This is the result for an arbitrary number of $p = 0.2$.

4.6 Stationary Distribution for a Stocastic Matrix

Given the following matrix :

$$P = \begin{bmatrix} 1/2 & 1/2 & 0 \\ 1/4 & 1/2 & 1/4 \\ 0 & 1/2 & 1/2 \end{bmatrix}$$

Using python code the following stationary distribution (π) has been found.

```
    import numpy as np


    P = np.array([[1/2, 1/2, 0],
              [1/4, 1/2,1/4],
              [0, 1/2, 1/2]])


    eigenvalues, eigenvectors = np.linalg.eig(P.T)
    stationary_distribution = np.real(eigenvectors[:, 0] / np.sum(eigenvectors[:, 0]))

    print("Stationary Distribution:")
    print(stationary_distribution)
Out: Stationary Distribution:
    [0.25 0.5  0.25]
```

5 A.6 Simulation of a 2D Ising model by the Metropolis algorithm

In this exercise is required to simulate 2D Ising model using Markov Chain MonteCarlo algorithm. The simulations is run for different lattice configurations $L = 10, 25, 50$ and for different temperature configurations $T = 1.5, 2.5, 3.5$. After the simulation, to verify that the algorithm is showing the expected behaviour after the equilibration time, different quantities will be calculated, such as specific heat and susceptibility. The other characteristic of this system is the critical slow down and the possibility to find the value of critical exponent α and β.

5.1 Program

The steps to build the program to simulate a 2D Ising model by Metropolis algorithm are the following:

- definition of the function to initialise spins: to better evaluate the equilibration times for different configurations, I run different probability to have a bias towards 1 or -1, so for each configurations of temperature and lattice we have 3 simulations: one with random initialization, one with positive initialization and one with negative initialization.

```
// Function to initialize spins with a bias towards 1
void initialize_spins(int spins[][100], int N, double probability_of_1) {
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            spins[i][j] = ((double)rand() / RAND_MAX)
            < probability_of_1 ? 1 : -1;}}}
```

- definition of the function to calculate the energy: the energy has been calculated, following the rule of the Ising model:

$$H = -J \sum_{<i,j>} s_i s_j \tag{31}$$

$$= -J \sum_{<i,j>} s_{i,j}(s_{i+1,j} + s_{i-1,j} + s_{i,j+1} + s_{i,j-1}) \tag{32}$$

taking into account nearest neighbours rule, $J = 1$, and boundary conditions.

```
double calculate_energy(int spins[][100], int N) {
double energy = 0.0;
for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
        energy -= spins[i][j] * (spins[(i+1)%N][j]
```

```
                + spins[(i-1+N)%N[j]+spins[i][(j+1)%N]
                + spins[i][(j1+N)%N]);}}
return energy / 2.0;
}
```

- definition of Glauber step: the flip probability is computed as:

$$p_{flip} = \frac{1}{1 + e^{-2\beta\Delta E}} \tag{33}$$

where

$$\Delta E = 2s_{i,j}(s_{i+1,j} + s_{i-1,j} + s_{i,j+1} + s_{i,j-1}) \tag{34}$$

```
void metropolis_step(int spins[][100], int N, double beta) {
    // Select a random site (i, j)
    int i = rand() % N;
    int j = rand() % N;

    // Calculate the energy change (delta_energy) for flipping the spin
    int delta_energy = 2 * spins[i][j] * (
        spins[(i + 1) % N][j] +         // Interaction with the spin on
        the right
        spins[(i - 1 + N) % N][j] +     // Interaction with the spin on
        the left (with periodic boundary)
        spins[i][(j + 1) % N] +         // Interaction with the spin
        below
        spins[i][(j - 1 + N) % N]       // Interaction with the spin
        above (with periodic boundary)
    );

    // Apply Metropolis-Glauber rule to decide whether to flip the spin
    if (delta_energy <= 0) {
        // If the energy change is negative or zero,
        always accept the spin flip
        spins[i][j] *= -1;
    } else {
        // Otherwise, compute the probability of accepting the flip
        double p_flip = exp(-beta * delta_energy);

        // Flip the spin with probability p_flip
        if ((double)rand() / RAND_MAX < p_flip) {
            spins[i][j] *= -1;
        }
    }
```

```
}
```

- Run the simulation:

```
int main() {
srand((unsigned)time(NULL));  // Random number generator intialization

// Temperatures an Lattice dimension
double temperatures[] = {1.0, 2.5, 3.5};
int sizes[] = {25, 50, 100};
int num_temperatures = sizeof(temperatures) / sizeof(temperatures[0]);
int num_sizes = sizeof(sizes) / sizeof(sizes[0]);

// Probability to have positive spins
    double probability_of_1 = 0.3; // 30% positive spins

// Iterare su tutte le combinazioni di temperature e dimensioni
for (int t = 0; t < num_temperatures; ++t) {
    for (int s = 0; s < num_sizes; ++s) {
        int N = sizes[s];
        double beta = 1.0 / temperatures[t];
        int spins[100][100];  // maximum dimension
        initialize_spins(spins, N, probability_of_1);
    char filename[50];
        snprintf(filename, sizeof(filename),"ising_data_T%.1f_N%d_
        negative.txt", temperatures[t], N);
        FILE *file = fopen(filename, "w");
        if (file == NULL) {
            perror("Error opening file");
            return EXIT_FAILURE;
        }
    // Simulation
    for (int step = 0; step < N_STEPS; ++step) {
        metropolis_step(spins, N, beta);
        double energy_metropolis = calculate_energy(spins, N);
        double magnetization_metropolis = 0.0;
        for (int i = 0; i < N; ++i) {
            for (int j = 0; j < N; ++j) {
                magnetization_metropolis += spins[i][j];
            }
        }
        magnetization_metropolis /= (N * N);  // Normalizzare
```

```
        fprintf(file, "Metropolis %d %lf %lf\n", step,
        energy_metropolis, magnetization_metropolis);
    }fclose(file);}}
return 0;
}
```

5.2 Data Analysis

5.2.1 Data visualisation

First above all, using python libraries we can have a viasualization of the simulation:

From a first observation of the evolution of magnetization and energy over time we can clearly state that for a lower temperature we have smoother curve both for magnetization and energy, this allows to better identify the equilibration time. In a second statement we can say that increasing the dimension of the lattice we also have the increasing of equilibration time.

T/N	**25**	**50**	**100**
1	6200	26000	80000
2.5	5000	25000	80000
3.5	2000	7500	40000

Table 2: Equilibration time for different configuration

5.2.2 Ensemble avrages

Let's go on with data analysis: after the data analysis for each configuration I did and estimation of

- average magnetisation:

$$< M >= \frac{1}{N} \sum_{i=1}^{N} M_i \tag{35}$$

- average energy:

$$< E >= \frac{1}{N} \sum_{i=1}^{N} E_i \tag{36}$$

- specific heat:

$$C = \frac{< E^2 > - < E >^2}{T^2 N} \tag{37}$$

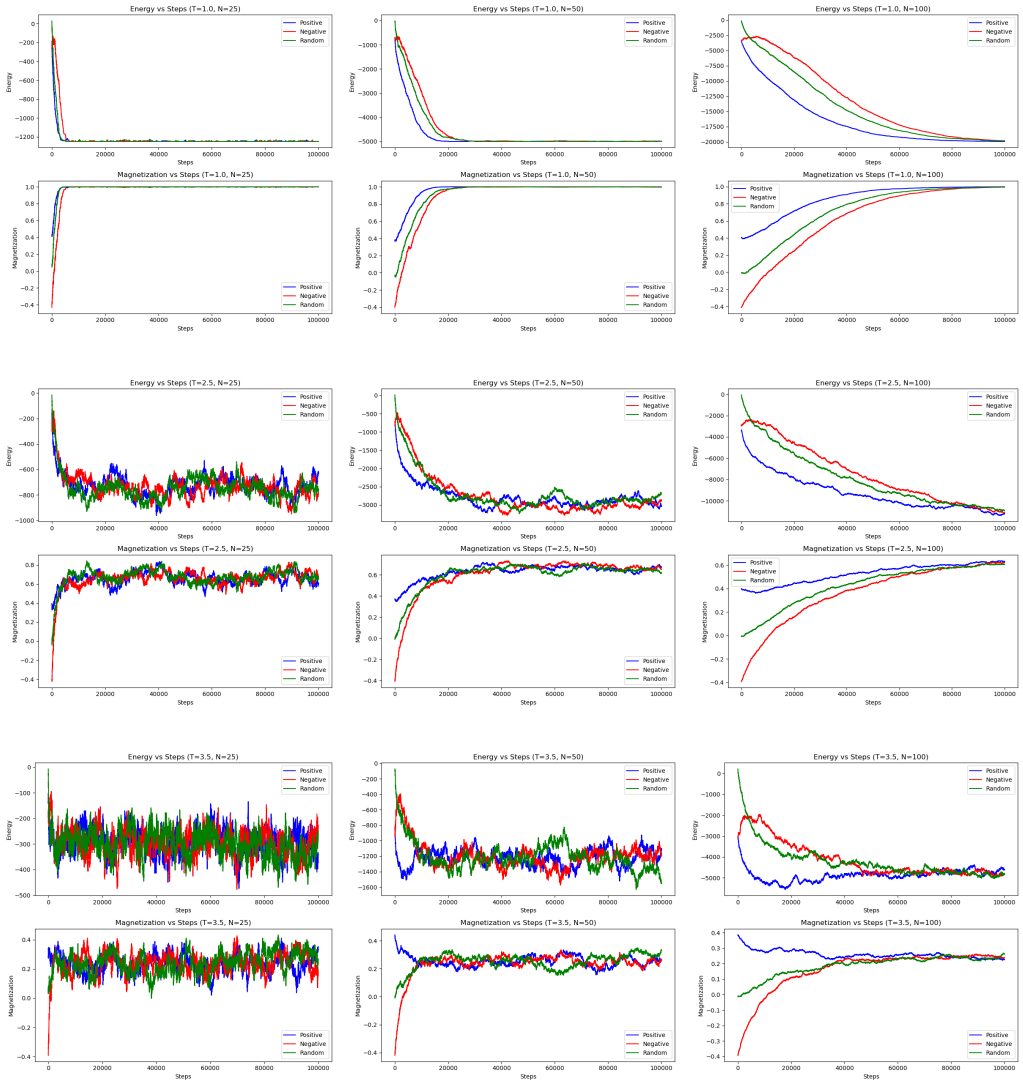


Figure 16: Evolution in time of Energy and Magnetization for each combination of Configurations of temperature and lattice dimension.

- megnetic susceptibility χ

$$\chi = \frac{< M^2 > - < M >^2}{TN} \tag{38}$$

T	N	Average Magnetization	Average Energy	Specific Heat	Susceptibility
1.0	25	0.999447	-1248.624359	0.449979	7.34e-08
1.0	50	0.999322	-4993.415261	1.368091	1.61e-08
1.0	100	0.990969	-19702.546933	184.173583	2.02e-07
2.5	25	0.671540	-746.666512	28.838846	5.36e-05
2.5	50	0.634315	-2822.266821	318.731737	4.29e-05
2.5	100	0.604181	-10684.467967	116.925853	7.25e-07
3.5	25	0.241111	-297.256279	7.597408	4.80e-05
3.5	50	0.255483	-1229.930011	19.105655	6.62e-06
3.5	100	0.237093	-4699.664100	22.138560	4.74e-07

Table 3: Results obtained through simulations with different configurations of temperature and lattice dimensions.

To verify that the results were coherent with the expected results, I plot the specific heat and magnetic susceptibility as a function of temperature, indeed we expect to observe a sort of divergence near the critical temperature, of course since we have only three different values of temperature I do not have a consistent plot but it is still useful to check the results.

As we can observe the only configurations that does not show the correct behaviour is the specific heat at high temperature, but we can observe that in the high temperatures we have a really noisy signal so the calculated values may be affected by this noise, if I had to go into detail of this problem, i would do many more simulations to observe if something would change.

5.3 Integrated correlation time and critical slowing down

In this section I will give an estimation of the autocorrelation time: this are the quantities I have to define:

- auto-correlation function: this function allows us to see how much a current value in a time series depends on the previous values; autocorrelation is defined as:

$$C_O(t) = \int dt'[O(t')O(t'+t) - < O >^2] \tag{39}$$

  $C_O(t)$ is expected to decay to zero fast enough since the sampled configurations are very far apart to be correlated. It is possible to show that $C_O(t)$ goes to zero as

$$C_O(t) \sim e^{-t/\tau^O_{int}} \tag{40}$$

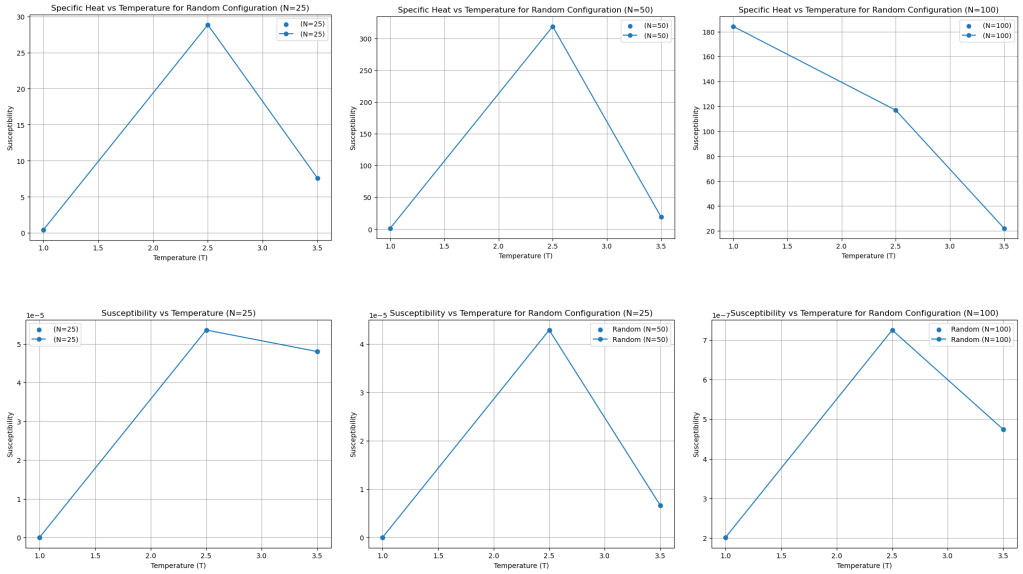


Figure 17: Specific heat and magnetic susceptibility vs temperature, I specify there is a mistake in the image for magnetic susceptibility N=50, it is written N=25 but it refers to N=50.

The carachteristic time τ_{int}^{O} is known as the integrated autocorrelation time of the observable O. Since we are dealing with trajectories that are discrete in time the integral is reprlaced by temporal series:

$$C_O(t) = \frac{1}{t_{max} - t} \sum_{t'=0}^{t_{max}-t} O(t')O(t'+t) \tag{41}$$

- estimation of autocorrelation time τ: it is considered as the first point t where the autocorrelation function C(t) falls below the value $\frac{1}{e}$.
- estimation of the integrated autocorrelation time τ_{int}, calculated as $\tau_{int} \simeq 2\tau$.
- to improve the analysis the estimate of error bars for integrated autocorrelation time can be calculated:

$$S^2_{t_{max}} = \frac{1}{(t_{max} - \tau_{eq} - 1)} \sum_{\tau_{eq}}^{t_{max}} (O_t - O_{t_max})^2 \tag{42}$$

Here I report only the autocorrelation time vs lag for just one configuration but it have been computed for each configuration.

In particular to observe critical slowing down, I want to see if for critical temperature we have an increase of the autocorrelation time, as predicted from the Ising model.

As expected we have a larger integrated time correlation for the temperature near critical one, so the critical slowing down for metropolis algorithm is verified.

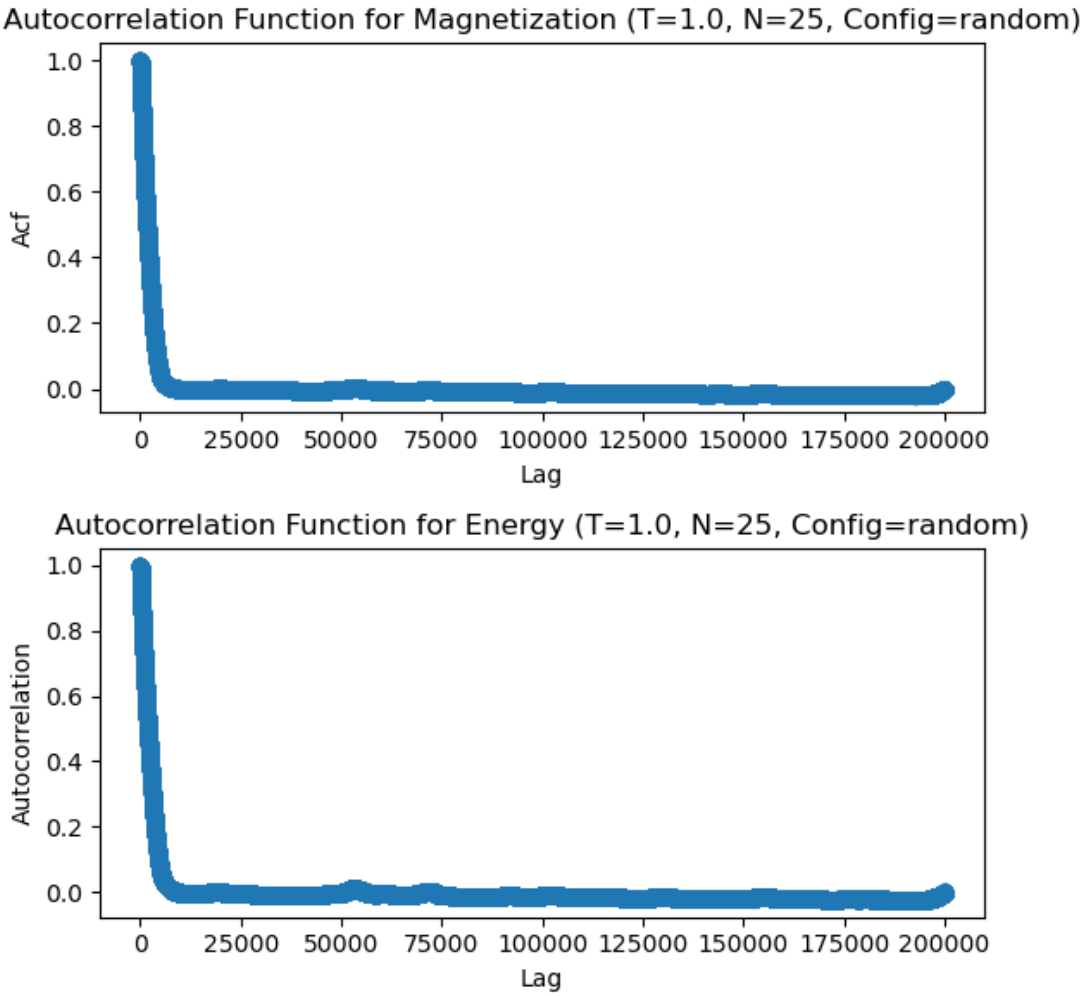


Figure 18: Acutocorrelation for magnetization and Energy $T = 1.0$,$N = 25$

5.4 Finite size analysis and estimates of critical exponent

To perform the Finite size scaling analysis and the estimation of the critical exponent, I went through the following steps:

- Extraction of the specific heat ($Cv(T_c)$), magnetization ($M(T_c)$), and susceptibility ($\chi(T_c)$), from the dataset for the nearest value to the critical temperature $T_c = 2.5$.
- $L = \sqrt{N}$ to represent the system size since we're dealing with 2D Ising model.
- Fit of each observable as a power law of L. The critical exponent are extracted:
  - α/ν from the fit of specific heat;
  - β/ν from the fit of magnetization;

T	N	$\tau_{magnetization}$	τ_{energy}
1	25	5307	6122
1	50	23356	27183
1	100	71908	78551
2.5	25	8143	8870
2.5	50	30388	34583
2.5	100	77592	79398
3.5	25	4908	3750
3.5	50	16773	15993
3.5	100	48661	41805

Table 4: Table of tau mag and tau en values for different T and N.

  - γ/ν from the fit of susceptibility.

- The results are plot in a log-log scale to visualize the power law dependence of each observable on the system size L.

Results:

```
Estimation of alpha/nu: 0.5346685149354818
Estimation of beta/nu: 0.15291756171116824
Estimation of gamma/nu: -1.9692899082322732
```

Expectd Results:

```
alpha/nu=0
beta/nu=0.125
gamma/nu=1.75
```

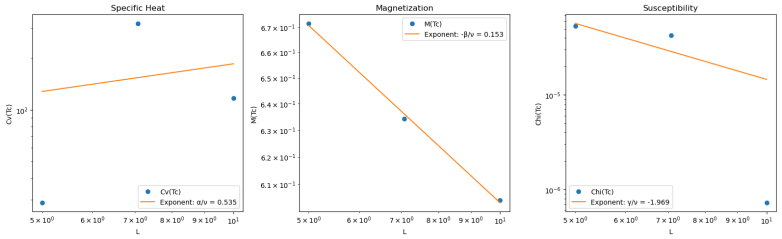


Figure 19: Plot of Specific Heat, Magnetization and Susceptibility vs size of the system in log-log scale.

The estimation of critical exponent doesn't give back a good result if we compare the critical exponents with values given by literature, this could be given by the fact that we have only three points, and also the noise and the uncertainty we have around the critical temperature.

This results are obtained assumin that $\nu = 1$.

6 A7 Advanced simulation of a 2D Ising Model

6.1 Wolff cluster algorithm

This time I choose to write the algorithm in python, The Wollf algorithm is indeed an optimization of the single-flip algorithms. It is based on cluster dynamics, so instead updating individual spins, it updates cluster of correlated spins. The fact I choose to use Python instead of C language and my laptop- which owns 8GB of RAM- was able to perform it in a reasonable time is another signal that the optimisation have been performed. The algorithm goes through the following steps:

- Initialization : A random spin is selected from the lattice.
- Cluster Formation: starting from the chosen spin, a cluster of connected spins with the same value is built. Each tima a neighboring spin is esamined, it is added to the cluser with a certain probability $P_add = 1 - e^{-2\beta}$, which is found imposing the ration of acepptance equal to one, this implies that spins are added to he cluster with a temperature dependent probability;
- Flipping the cluster: once the cluster is formed, all spins in the cluster are flipped.

Code:

```
    import numpy as np
import matplotlib.pyplot as plt


def initialize_lattice(N):
    """Initialize the 2D Ising lattice."""
    return np.random.choice([-1, 1], size=(N, N))

def wolff_cluster_step(lattice, beta):
    """Perform one Wolff cluster update step."""
    N = lattice.shape[0]
    # Pick a random starting spin
    x0, y0 = np.random.randint(N), np.random.randint(N)
    spin_to_flip = lattice[x0, y0]

    # Initialize cluster
    cluster = set()
    stack = [(x0, y0)]
    cluster.add((x0, y0))

    while stack:
        x, y = stack.pop()
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
```

```
            nx, ny = (x + dx) % N, (y + dy) % N
            if lattice[nx, ny] == spin_to_flip and (nx, ny) not in cluster:
                # Probability to add this spin to the cluster
                if np.random.rand() < 1 - np.exp(-2 * beta):
                    cluster.add((nx, ny))
                    stack.append((nx, ny))

    # Flip the spins in the cluster
    for x, y in cluster:
        lattice[x, y] *= -1

def simulate_ising_model(N, beta, n_steps):
    """Run the Wolff cluster algorithm on a 2D Ising model."""
    lattice = initialize_lattice(N)
    for _ in range(n_steps):
        wolff_cluster_step(lattice, beta)
    return lattice

def calculate_magnetization(lattice):
    """Calculate the magnetization of the lattice."""
    return np.sum(lattice) / lattice.size

if __name__ == "__main__":
    # Parameters
    N = 20               # Size of the lattice
    beta = 0.4           # Inverse temperature
    n_steps = 1000       # Number of Wolff updates

    # Run the simulation
    final_lattice = simulate_ising_model(N, beta, n_steps)

    # Calculate and print the magnetization
    mag = calculate_magnetization(final_lattice)
    print(f"Magnetization: {mag}")

    # Optional: Display the final lattice configuration
    import matplotlib.pyplot as plt
    plt.imshow(final_lattice, cmap='gray', interpolation='none')
    plt.title(f'Final Lattice Configuration (Magnetization: {mag:.2f})')
    plt.show()
```

Result:

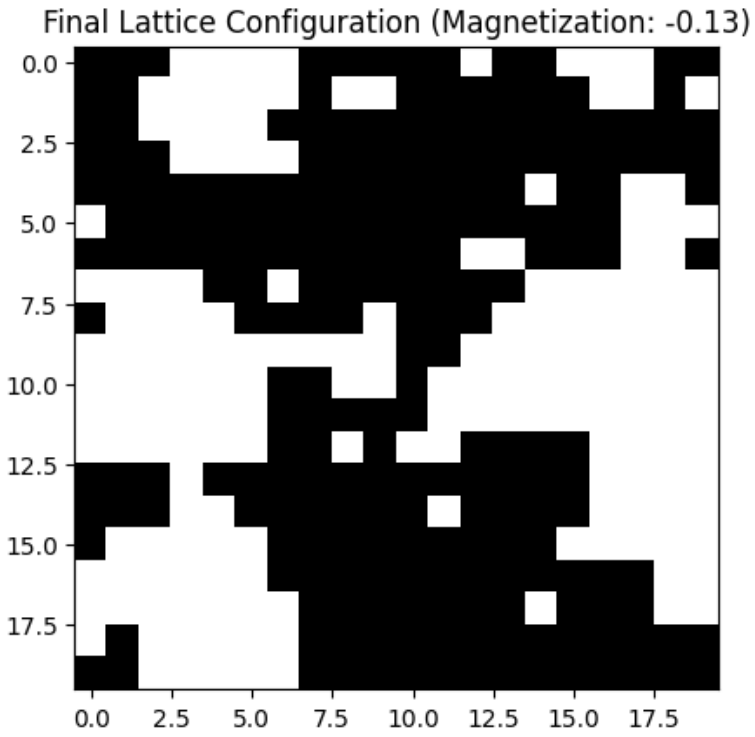


Figure 20: Magnetic configuration a the end of the simulation

6.2 Cluster size statistics

In this section I was asked to collect the cluster sizes of the Wolff algorithm for three temperature: $T_c = \frac{2}{ln(1+\sqrt{2})}$ the critical temperature, $T_h = 2T_c$ a high temperature, and $T_l = T_c/2$. Here I will show the three histogram of the cluster sizes density: This is what

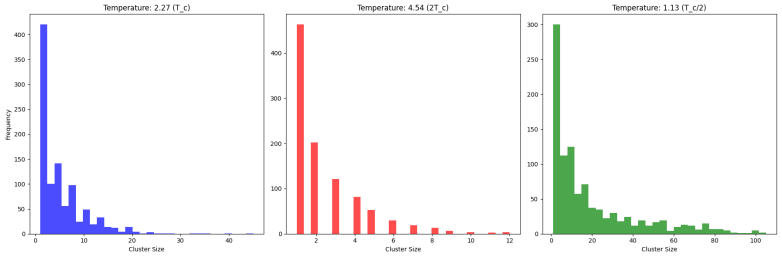


Figure 21: Cluster size density for 3 different value of T

I can observe:

- At T_c the distribution shows a broad range including many medium-sized clusters due the balance between ordered and disordered states.

- At $2T_c$ I can see a distribution skewes toward smaller clusters, as the system is mostly disordered.
- at $T_c/2$ the likelihood of forming large clusters increases significantly, as the spins are mire likely to be aligned due to lower thermal energy. This results in a distribution with higher frequency of larger clusters and a peak that reflects the tendency toward order.

6.3 Autocorrelation time

This point I was asked to perform the simulation at critical temperature T_c for different lattice sizes $L = [5, 50, 500, 5000]$ and then plot in a log-log scale to observe if is scales with lattice size as predicted by theoretical description. We can observe that for the

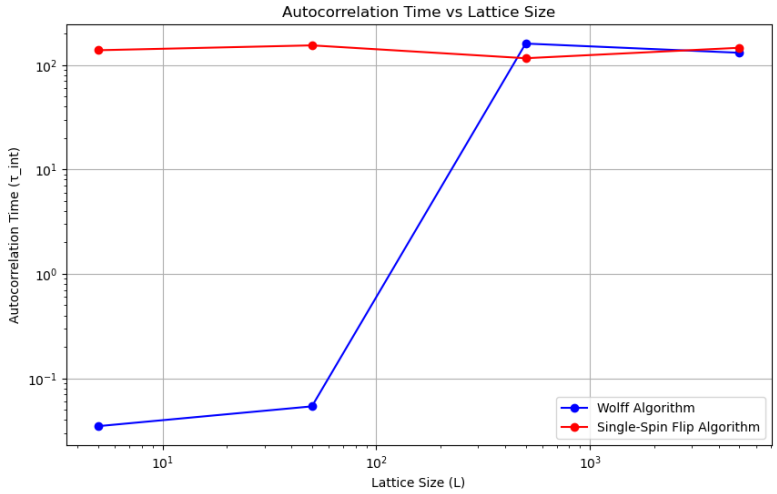


Figure 22: Auto-correlation time for single flip algorithm and Wolff algorithm vs lattice size L, fixed temperature T_c.

single flip algorithm the auto-correlation time is not scaling, instead it is stable at some value of the order 10^3, instead for the simulation runned with Wolff algorithm we have the scaling related to the lattice size.

6.4 Multiple Markov Chains

In this simulation multiple Markov chains are being used as part of a parallel tempering method, a powerful technique, particularly for systems with many local energy minima, such as the Ising model. In the code several independent Ising configurations ("chains") are simulated at different temperature $\beta = 1/T$, these chains evolve using Metropolis algorithm. Each chain samples from Boltzmann distribution at its respective temperature. However, low-temperature chains tend to get suck in local minima, while high-temperature chains explore configuration space more freely. To overcome the problem of chains getting trapped in local energy, the parallel tempering method periodically

attempts to swap configurations between adjacent chains. The swap probability is based on the difference in temperatures of the two chains and the energy difference between their configurations. This method still respect the detailed balance condition. If the energy difference between the configurations and the temperature difference between the chains are favorable a swap happens, otherwise is rejected.

So using this method allows to overcome the critical slowing down problem even at low temperature, where even though Wolff algortihm performs better than Metropolis one we avoid the problem to remain stuck in local minima. Here I report the plot to compare the calculated autocorrelation time betwwen the swap and the non-swapping case.

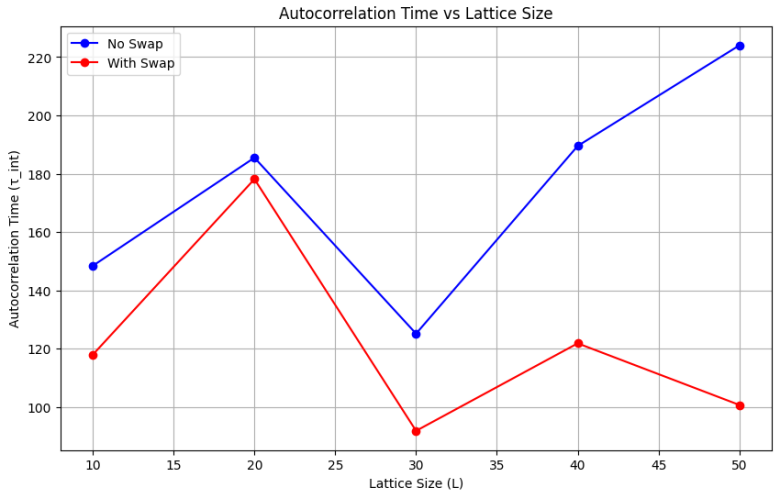


Figure 23: Auto-correlation time vs lattice dimension, comparison between swapping and non-swapping case

We can observe that increasing the lattice dimension we have in increasing of the autocorrelation time time for the non swapping case while the swapping case decrease by increasing the dimension, showing the optimization of this algorithm.

7 A8 Continuous time Markov process

7.1 Gillespie

To simulate a system as a continuous-time Markov process the Gillespie Algorithm have been used since it has the following two properties:

- memoryless: the probability of a particular event happening next depends only on the current state of the system, not on its past history;
- exponential waiting times: the times between events are drawn from an exponential distribution, this is charachteristic of continuos time Markov process, where the time to the next event is exponentially distributed.

It goes also through this two main steps:

- first it extract a random time from the exponential distribution;
- the second step requires choosing the state j to jumo from C, the actual state of the system. The probability $p(i|C)$ of picking randomly a given j must be proportional to the transition rate ω_{Cj}

7.2 Lotka Volterra

Lotka-Volterra model, also known as the predator-prey model, is a system of differential equations that describes the time evolution of two interacting populations: prey and predators. It is described by the following ODEs:

$$\frac{dN_{prey}}{dt} = k_1 N_{prey} - k_2 N_{prey} N_{predators} \tag{43}$$

$$\frac{dN_{predators}}{dt} = k_2 N_{prey} N_{predators} - k_3 N_{predators} \tag{44}$$

where:

- k_1 is the growth rate of the prey population;
- k_2 rate of interaction between prey and predators;
- k_3 natural death rate of predator population.

So the model follows the following reaction rules:

$$X_1 \xrightarrow{k_1} 2X_1 \tag{45}$$

$$X_1 + X_2 \xrightarrow{k_2} 2X_2 \tag{46}$$

$$X_2 \xrightarrow{k_3} 0 \tag{47}$$

where $X_1 = N_{prey}$ and $X_2 = N_{predators}$. They correspond to following transition rates for the Gillespie algoritm:

- to connect the state $C = (X_1, X_2)$ to $C' = (X_1 + 1, X_2)$

$$\omega_1 = k_1 X_1 \tag{48}$$

- to connect the state $C = (X_1, X_2)$ to $C' = (X_1 - 1, X_2 + 1)$

$$\omega_2 = k_2 X_1 X_2 \tag{49}$$

- to connect the state $C = (X_1, X_2)$ to $C' = (X_1, X_2 - 1)$

$$\omega_3 = k_3 X_2 \tag{50}$$

Parameter	Value
k_1 (Prey birth rate)	3.0
k_2 (Predation rate)	0.01
k_3 (Predator death rate)	5.0
Prey population	300
Predator population	500
Total time (T)	100.0

Table 5: Simulation Parameters for Lotka-Volterra Model

The Gillespie algorithm to simulate Lotka-Volterra model go through the following steps:

- parameters setting:
- Initialization of empty list to store time, pray and predator counts;
- main simulation loop:

```
 while t < T and i < MAX_ITER:
rate1 = k1 * prey
rate2 = k2 * prey * predators
rate3 = k3 * predators

total_rate = rate1 + rate2 + rate3
if total_rate <= 0:
    break

r1 = rand_uniform()
r2 = rand_uniform()

t += -np.log(r1) / total_rate

if r2 < rate1 / total_rate:
    prey += 1
elif r2 < (rate1 + rate2) / total_rate:
    prey -= 1
    predators += 1
else:
    predators -= 1
```

- updated of the empty list for each time step.

As you can see I've started from equilibrium configuration which takes to oscillations of the density of the system.

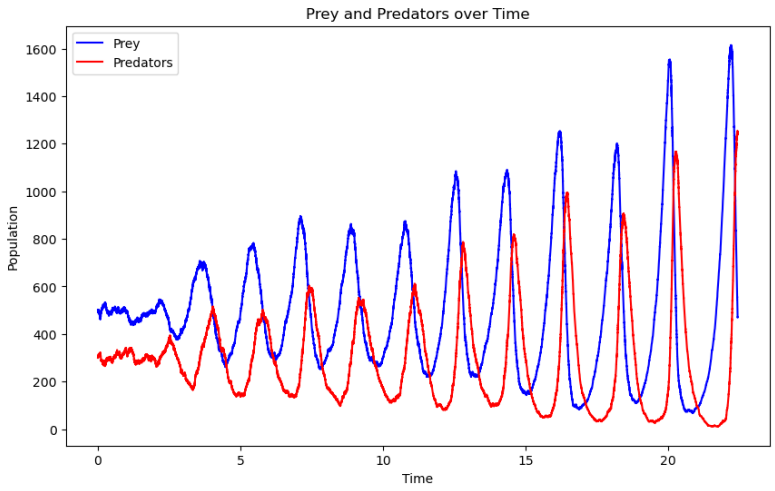


Figure 24: Lotka-Volterra model with initial conditions set as steady state quantity.

To find a configuration in which predators population does extinct we have two possibility: increase K_3, the natural death rate of predators, or decrease initial number of prey, so predators do not have enough to eat and they die out of the system. I choose the second one and this is what I obtained: I must specify that this happened just once

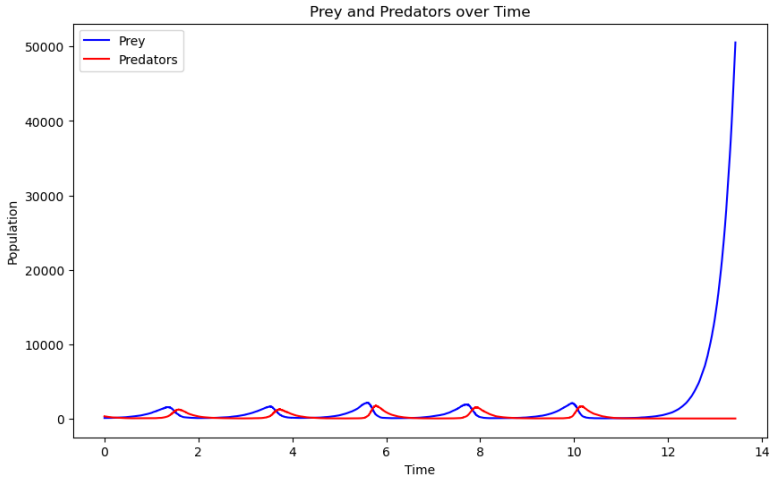


Figure 25: Lotka-Volterra model with initial conditions set decreasing drastically number of prey.

in a while, indeed most of the time I obtained oscillations again, this is caused by the deterministic nature of Lotka-Volterra model: if we do the eigenvalue analysis and we assume that the transition rate are always positive we always obtain complex eigenvalue

which takes us to oscillatory behaviour. The Gillspie algorithm is instead a stochastic algorithm whics is subjected by stochastic noise, and this allows to have not an oscillatory behaviour in which the predator die out and prey explode.

7.3 Brusselator

A brief intriduction about Brusselator model: it is the minimal paradigmatic model for generating chemical periodicity, such as in the Belousov-Zhabotinsky reaction. A state $C = (X, Y)$ of the system represents the numbers X and Y of molecules of two species. The chemical reaction of the Brusselator are [from note CTMP]:

$$0 \xrightarrow{k_1} X \tag{51}$$

$$X \xrightarrow{k_2} 0 \tag{52}$$

$$2X + Y \xrightarrow{k_3} 3X \tag{53}$$

$$X \xrightarrow{k_4} Y \tag{54}$$

which corresponds to the following rates for the Gillespie algorithm:

- to connect the state $C = (X, Y)$ to $C' = (X + 1, Y)$

$$\omega_1 = a\Omega \tag{55}$$

- to connect the state $C = (X, Y)$ to $C' = (X - 1, Y)$

$$\omega_2 = X \tag{56}$$

- to connect the state $C = (X, Y)$ to $C' = (X + 1, Y - 1)$

$$\omega_3 = \frac{1}{\Omega^2} X(X - 1)Y \tag{57}$$

- to connect the state $C = (X, Y)$ to $C' = (X + 1, Y - 1)$

$$\omega_4 = bX \tag{58}$$

Section of the code different from the previous one:

```
while t < T and i < MAX_ITER:
    rate1 = a * volume
    rate2 = X
    rate3 = (X * (X-1) * Y) / (volume**2)
    rate4 = b*X

    total_rate = rate1 + rate2 + rate3 + rate4
```

```
if total_rate <= 0:
    break

r1 = rand_uniform()
r2 = rand_uniform()

t += -np.log(r1) / total_rate

if r2 < rate1 / total_rate:
    X += 1
elif r2 < (rate1 + rate2) / total_rate:
    X -= 1

elif r2 < (rate1 + rate2 + rate3) / total_rate:
    X += 1
    Y -= 1
elif r2 < (rate1 + rate2 + rate3 + rate4) / total_rate:
    X-=1
    Y+=1
else:
    X-=1
```

Initial parameters:

Parameter	**Value**
a	2
b	5
X	3
Y	5
Total time (T)	100.0

Table 6: Simulation Parameters for Brusselator model

Results:

- **Top Left (Volume = 100):** Both species, labeled as x and y, exhibit clear oscillatory dynamics with high noise. The smaller volume increases stochastic effects, leading to irregular oscillations around a common periodic behavior. This is typical for small volumes, where the system is more susceptible to random fluctuations.
- **Top Right (Volume = 1000):** As the volume increases, the oscillations become smoother, indicating a shift towards more deterministic behavior. However, some noise is still visible, suggesting that stochasticity persists. The periodicity of the system is clearer in this case.

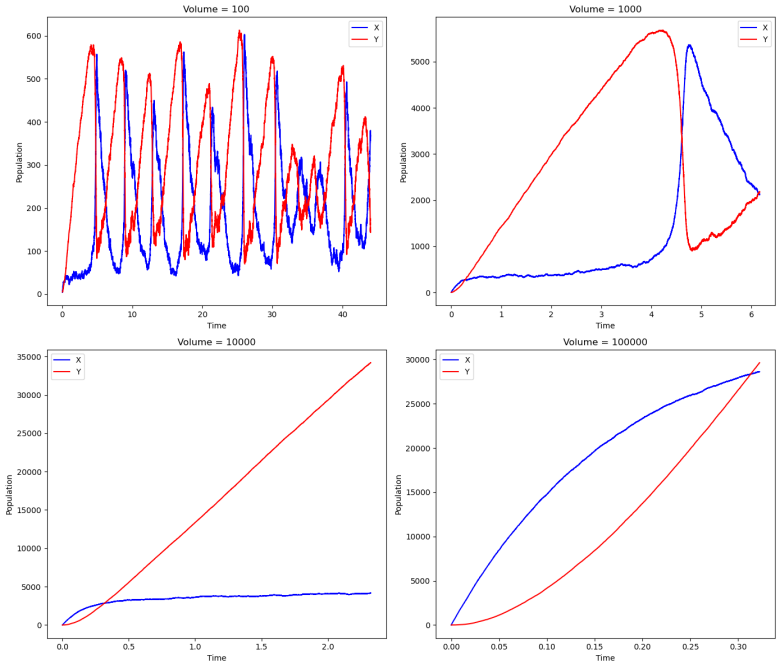


Figure 26: Running of Gillespie algorithm to simulate Brusselator model for diferent Volume $\Omega = [10^2, 10^3, 10^4, 10^5]$

- **Bottom Left (Volume = 10000):** For this larger volume, the dynamics of x and y show significantly less noise, with y growing at a much faster rate than x. The system appears to stabilize, with x reaching a steady state while y continues to increase, possibly due to saturation effects in the nonlinear terms of the Brusselator model.

- **Bottom Right (Volume = 100000):** This panel shows almost entirely deterministic behavior with smooth trajectories. Both species are converging towards their final steady states, though the dynamics are much slower. The overall shape suggests that the Brusselator system is reaching equilibrium in a large-volume, deterministic limit.

In summary, as the volume increases from 100 to 100,000, the stochastic fluctuations decrease, and the system transitions from noisy oscillations to smoother, more deterministic behavior. This is expected in models like the Brusselator, where the volume acts as a control parameter for the noise level in the system's dynamics.

8 A9 Off Lattice simulation basics

8.1 Reduce Units

8.1.1 Temperature Conversion from Reduce Units to SI Units

Given

- $\sigma_{Ar} = 3,41$Å, $\sigma_{Kr} = 3,38$Å
- $\frac{\epsilon_{Ar}}{k_B} = 119,8K$, $\frac{\epsilon_{Kr}}{k_B} = 164K$
- reduced temperature $T* = 2$

The reduced temperature T* is defined as

$$T* = \frac{T}{\epsilon/k_B} \tag{59}$$

where T is the temperature in kelvin and ϵ/k_B is the characteristic temperature for the substance. To find the actual temperatures for Argon and Krypton, we can rearrange the :

$$T = T * \frac{\epsilon}{k_B} \tag{60}$$

For Argon $T_{Ar} = 239,6K$, for Krypton $T_{Kr} = 328K$.

8.1.2 Conversion of the Time Step to SI Units

The time step is a quantity which is often given in reduced units. The reduced time step is defined as:

$$\Delta t* = \Delta t\sqrt{\frac{\epsilon}{m\sigma^2}} \tag{61}$$

where $\Delta t* = 0,001$, m is the mass of the particle, σ is the characteristic size, ϵ is the depth of the potential wall.

Here it will be calculated for Argon and for Krypton. Considering constant value written above, and $m_{Ar} = 39,948g/mol$, $m_{Kr} = 83,798g/mol$:

$$\Delta t_{Ar} = 2,15fs \tag{62}$$

$$\Delta t_{Kr} = 2,25fs \tag{63}$$

8.1.3 Friction Coefficient and Dynamical Viscosity in Reduced Units

Here friction coefficient and dynamical viscosity will be written in reduced units, so in this case they are both dimensionless. Friction Coefficient:

$$\xi* = \xi\frac{\sigma^2 m}{\epsilon} \tag{64}$$

Dynamical Viscosity

$$\eta* = \eta\frac{\sigma^3}{\sqrt{m\epsilon}} \tag{65}$$

8.2 Off lattice Monte Carlo

Here I report the main part of the code that can be used for more specific implementation.

- Parameters

Symbol	Value	Description
L	10	Lattice size (10x10x10)
N	L^3	Total number of particles
$d_{\max}$	0.1	Maximum displacement
T	1.0	Temperature

Table 7: Simulation Parameters

- Structure definition

```
typedef struct {
double x, y, z;
} Particle;
```

- applying boundary condition

```
// Apply periodic boundary conditions
double apply_PBC(double coord) {
    if (coord >= 1.0) coord -= 1.0;
    if (coord < 0.0) coord += 1.0;
    return coord;}
```

- definition of function specific of the model

```
    // Compute energy of a single particle (interaction with its neighbors)
double particle_energy(int i) {
    double energy = 0.0;
    // Here, we could implement some potential function based
    on particle positions
    // For example, Lennard-Jones potential, or any other interaction model
    return energy;
}
// Compute total energy of the system
double total_energy() {
    double energy = 0.0;
    for (int i = 0; i < N; i++) {
        energy += particle_energy(i);
    }
```

```
    return energy;
}
```

- perform a Monte Carlo trial move, first a single move than N trial moves

```
// Perform a single Monte Carlo trial move
void monte_carlo_move() {
    // Select a random particle
    int i = rand() % N;
    Particle old_pos = particles[i];

    // Propose random displacement in x, y, z directions
    double dx = dmax * (2.0 * rand_uniform() - 1.0);
    double dy = dmax * (2.0 * rand_uniform() - 1.0);
    double dz = dmax * (2.0 * rand_uniform() - 1.0);

    // Compute energy before move
    double old_energy = particle_energy(i);

    // Displace particle
    particles[i].x = apply_PBC(particles[i].x + dx);
    particles[i].y = apply_PBC(particles[i].y + dy);
    particles[i].z = apply_PBC(particles[i].z + dz);

    // Compute energy after move
    double new_energy = particle_energy(i);

    // Metropolis acceptance criterion
    double delta_E = new_energy - old_energy;
    if (delta_E > 0 && exp(-delta_E / (K_B * temperature)) < rand_uniform()) {
        // Reject the move (restore the old position)
        particles[i] = old_pos;
    }
}

// Perform a Monte Carlo sweep (N trial moves)
void monte_carlo_sweep() {
    for (int i = 0; i < N; i++) {
        monte_carlo_move();
    }
}
```

8.3 Off lattice Monte Carlo of Hard Spheres

Hard sphere model is a paradigmatic model in Soft Matter. The interaction energy, between two hard spheres if the separation distance is larger of the particle σ and infinite otherwise.

Implementation:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define N 100           // Number of particles
#define sigma 1.0       // Diameter of hard sphere

typedef struct {
    double x, y, z;
} Particle;

Particle particles[N];
double L;  // Simulation box length
double temperature = 1.0;  // Temperature (irrelevant for hard spheres)

// Initialize random number generator
double rand_uniform() {
    return (double)rand() / RAND_MAX;
}

// Apply periodic boundary conditions
double apply_PBC(double coord) {
    if (coord >= L) coord -= L;
    if (coord < 0.0) coord += L;
    return coord;
}

// Compute squared distance between two particles
with periodic boundary conditions
double distance_squared(Particle p1, Particle p2) {
    double dx = fabs(p1.x - p2.x);
    double dy = fabs(p1.y - p2.y);
    double dz = fabs(p1.z - p2.z);

    if (dx > L / 2) dx = L - dx;
    if (dy > L / 2) dy = L - dy;
    if (dz > L / 2) dz = L - dz;
```

```
    return dx * dx + dy * dy + dz * dz;
}

// Check for overlap
int has_overlap(int i) {
    for (int j = 0; j < N; j++) {
        if (i != j) {
            double dist_sq = distance_squared(particles[i], particles[j]);
            if (dist_sq < sigma * sigma) {
                return 1;  // Overlap detected
            }
        }
    }
    return 0;  // No overlap
}

// Compute total energy
double total_energy() {
    for (int i = 0; i < N; i++) {
        if (has_overlap(i)) {
            return 1e6;  // Very large energy for overlap
        }
    }
    return 0.0;  // Zero energy if no overlaps
}

// Perform a single Monte Carlo trial move
int monte_carlo_move(double dmax) {
    int i = rand() % N;
    Particle old_pos = particles[i];

    // Propose random displacement in x, y, z directions
    double dx = dmax * (2.0 * rand_uniform() - 1.0);
    double dy = dmax * (2.0 * rand_uniform() - 1.0);
    double dz = dmax * (2.0 * rand_uniform() - 1.0);

    // Displace particle
    particles[i].x = apply_PBC(particles[i].x + dx);
    particles[i].y = apply_PBC(particles[i].y + dy);
    particles[i].z = apply_PBC(particles[i].z + dz);

    // Check for overlaps after the move
    if (has_overlap(i)) {
```

```
        // Reject the move (restore the old position)
        particles[i] = old_pos;
        return 0;  // Move rejected
    }
    return 1;  // Move accepted
}

// Perform a Monte Carlo sweep (N trial moves)
void monte_carlo_sweep(double dmax, int* accepted_moves, int* total_moves) {
    for (int i = 0; i < N; i++) {
        (*total_moves)++;
        *accepted_moves += monte_carlo_move(dmax);
    }
}

// Initialize the particle positions randomly within the 3D box
void initialize_particles() {
    for (int i = 0; i < N; i++) {
        particles[i].x = rand_uniform() * L;
        particles[i].y = rand_uniform() * L;
        particles[i].z = rand_uniform() * L;
    }
}

// Initialize particles in a simple cubic lattice
void initialize_cubic_lattice() {
    double spacing = sigma;  // Set the spacing to the diameter
    for (int i = 0; i < N; i++) {
        particles[i].x = (i % 10) * spacing;  // Assume a 10x10x1 lattice
        particles[i].y = ((i / 10) % 10) * spacing;
        particles[i].z = (i / 100) * spacing;
    }
}

int main() {
    // Seed random number generator
    srand(time(NULL));

    // Array of dmax values and densities
    double dmax_values[] = {0.01, 0.1, 0.5, 0.75, 1.0};
    double densities[] = {0.05, 0.3, 0.5, 1.0};
    int num_dmax = sizeof(dmax_values) / sizeof(dmax_values[0]);
    int num_densities = sizeof(densities) / sizeof(densities[0]);
    int num_realizations = 10; // Number of realizations
```

```
    for (int d = 0; d < num_densities; d++) {
        for (int dm = 0; dm < num_dmax; dm++) {
            double rho = densities[d];
            L = cbrt(N / rho);  // Set simulation box size based on density

            double dmax = dmax_values[dm];
            double total_energy_sum = 0.0;
            int total_accepted_moves = 0;
            int total_moves = 0;

            for (int realization = 0;
            realization < num_realizations; realization++) {
                initialize_particles(); // For random initialization
                // initialize_cubic_lattice();
                // Uncomment for cubic lattice initialization

                int num_sweeps = 1000; // Number of Monte Carlo sweeps
                for (int sweep = 0; sweep < num_sweeps; sweep++) {
                    int accepted_moves = 0;
                    monte_carlo_sweep(dmax, &accepted_moves, &total_moves);
                    total_accepted_moves += accepted_moves;
                    total_energy_sum += total_energy();

                    // Print energy every 100 sweeps
                    if (sweep % 100 == 0) {
                        //printf("Density: %.2f, dmax: %.2f, Sweep %d,
                        Total Energy: %f\n",
                               //rho, dmax, sweep, total_energy());
                    }
                }
            }

            double acceptance_ratio = (double)total_accepted_moves
            / total_moves;
            printf("Density: %.2f, dmax: %.2f, Acceptance Ratio: %.4f,
            Average Energy: %f\n",
                   rho, dmax, acceptance_ratio,
                   total_energy_sum / num_realizations);
        }
¢    }
return 0;
}
```

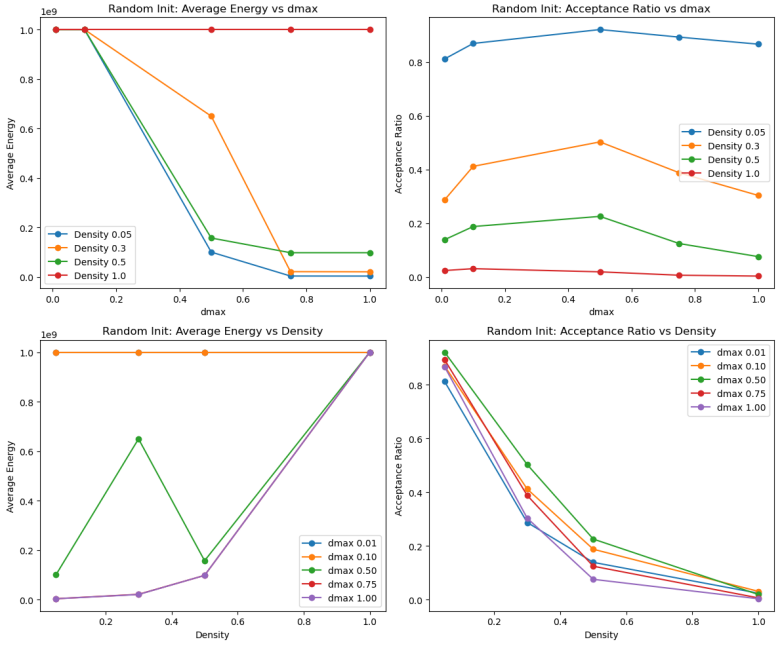


Figure 27: Results from hard spheres model simulation

The plots represent results from a hard spheres model simulation with varying parameters: density and $d_{\max}$, where $d_{\max}$ likely represents the maximum displacement in the system.

- **Top Left: Average Energy vs. $d_{\max}$ for Different Densities.** The average energy decreases drastically as $d_{\max}$ increases for lower densities (0.05 and 0.3), reaching almost zero around $d_{\max} = 0.5$. For higher densities (0.5 and 1.0), the energy shows a slower decline and stabilizes at lower values, indicating that higher density systems maintain a more constrained energy state despite the increase in $d_{\max}$.

- **Top Right: Acceptance Ratio vs. $d_{\max}$ for Different Densities.** The acceptance ratio initially increases for most densities up to $d_{\max} = 0.5$ and then decreases as $d_{\max}$ increases further. Systems at higher densities exhibit a significantly lower acceptance ratio, suggesting that fewer configurations are accepted as the system becomes more crowded and constrained.

- **Bottom Left: Average Energy vs. Density for Different $d_{\max}$.** At low $d_{\max}$, the average energy remains high for all densities. However, for larger $d_{\max}$ values (e.g., 0.5, 0.75), the energy rapidly decreases as density increases, except for some cases like $d_{\max} = 0.5$, where a peak is observed at intermediate density values.

- **Bottom Right: Acceptance Ratio vs. Density for Different $d_{\max}$.** The

acceptance ratio shows a decreasing trend as density increases. Systems with lower $d_{\max}$ values generally have higher acceptance ratios across all densities, while for higher $d_{\max}$ values, the ratio drops sharply as density increases.

In summary, the simulation results indicate that increasing $d_{\max}$ leads to a decrease in average energy and acceptance ratio, particularly in lower-density systems. Higher densities result in more constrained behavior with lower acceptance ratios, emphasizing the impact of particle crowding in hard spheres models.

For the **second requirement** the modification to the code was in the initialisation cubic lattice function:

```
// Initialize particles in a simple cubic lattice with spacing sigma
void initialize_cubic_lattice() {
    double spacing = sigma;  // Set the spacing to the diameter
    int L = ceil(pow(N, 1.0 / 3.0)); // Determine the cube root for dimensions

    for (int i = 0; i < N; i++) {
        particles[i].x = (i % L) * spacing;  // x position
        particles[i].y = ((i / L) % L) * spacing; // y position
        particles[i].z = (i / (L * L)) * spacing; // z position
    }
}
```

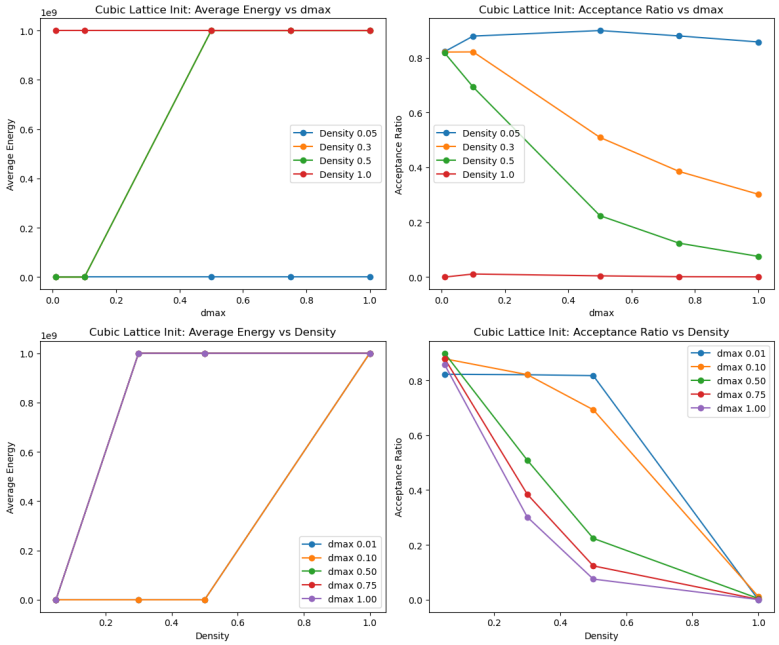


Figure 28: Results from hard spheres model simulation with fixed distance σ

The plots in Fig.28 illustrate the results from a hard spheres model simulation for varying density and $d_{\max}$, where $d_{\max}$ represents the maximum displacement in the system.

- **Top Left: Average Energy vs. $d_{\max}$ for Different Densities.** The average energy remains high when $d_{\max}$ is small but sharply decreases as $d_{\max}$ increases, especially for lower densities (e.g., 0.05 and 0.3). For densities like 0.5 and 1.0, the energy shows a less steep decrease, indicating that denser systems retain more energy despite increases in $d_{\max}$.
- **Top Right: Acceptance Ratio vs. $d_{\max}$ for Different Densities.** The acceptance ratio shows a non-linear behavior with respect to $d_{\max}$. At lower densities (e.g., 0.05), the acceptance ratio remains high across most $d_{\max}$ values, while at higher densities, the ratio decreases significantly as $d_{\max}$ increases, reflecting the impact of crowding on particle movement and configuration acceptance.
- **Bottom Left: Average Energy vs. Density for Different $d_{\max}$.** For small $d_{\max}$, the average energy remains nearly constant across all densities. However, for larger $d_{\max}$ values (0.5 and beyond), the energy decreases with increasing density, although for $d_{\max} = 0.5$, an unexpected energy peak occurs at intermediate densities.
- **Bottom Right: Acceptance Ratio vs. Density for Different $d_{\max}$.** The acceptance ratio decreases as density increases, particularly for higher $d_{\max}$ values. For lower $d_{\max}$, the acceptance ratio is initially high but gradually diminishes with density, while higher $d_{\max}$ leads to a steeper decline in acceptance, indicating the challenge of finding valid configurations in denser systems.

In conclusion, these results highlight that increasing $d_{\max}$ leads to a rapid decrease in both average energy and acceptance ratio, particularly at higher densities. The system becomes increasingly constrained, with fewer configurations accepted as the density rises, which is typical of the hard spheres model where particle overlap is strictly prohibited.

8.4 Off lattice Monte Carlo of Lennard Jones particles

As required in this exercise two simulations of Lennard-Jones are performed, one with $T* = 2$ (above the critical temperature) and one at $T* = 0,9$ (well below the critical temperature). I report here the theoretical assumptions to perform this simulation, from the exercise text: the Lennard-Jones interaction is another paradigmatic interaction potential, often used to implement "self-avoidance" in MD simulations or as paradigmatic example of gas-liquid phase separation. It reads:

$$V_{\mathrm{LJ}}(r) = \begin{cases} 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] & \text{for } r < \sigma_{\mathrm{cut}} \\ 0 & \text{for } r \geq \sigma_{\mathrm{cut}} \end{cases} \tag{66}$$

where $\sigma = 1$, $\epsilon = 1$ are the units of length and energy, and σ_{cut} is the so-called cut-off length, for this exercise it is set, to half of the box size. The cut-off length is set to

reduce computational costs by neglecting interactions beyond this distance. To account for the ignores interactions beyond the cutoff, two tail corrections are introduced

- Energy Tail Correction:

$$u_{tail} = \frac{8}{3}\pi\rho[\frac{1}{3}(\frac{\sigma}{\sigma_{cut}})^9 - (\frac{\sigma}{\sigma_{cut}})^3] \tag{67}$$

- Pressure tail correction:

$$P_{tail} = \frac{16}{3}\pi\rho^2[\frac{2}{3}(\frac{\sigma}{\sigma_{cut}})^9 - (\frac{\sigma}{\sigma_{cut}})^3] \tag{68}$$

The aim of the exercise is to verify the physical characteristic of the system in the pressure-density plane where the correction is applied.

Now I will report the structure of the program in c++ used to simulate the system.

- Main functions:
  - Initialisation the positions of the particles randomly within the box.

```
void init_Positions(double **X) {
// Initialize the matrix with random values for positions x, y, z
for (int i = 0; i < mySys.NPart; i++) {

X[i][0] = ran38(&(mySys.seed)) * mySys.box_x; // x position
X[i][1] = ran38(&(mySys.seed)) * mySys.box_y; // y position
X[i][2] = ran38(&(mySys.seed)) * mySys.box_z; // z position

}
}
```

  - Computation of energy : Calculates the total energy of the system based on the particle positions and includes a correction for the energy neglected beyond the cut-off distance.

```
double compute_energy(double **X){
double en = 0.0;
double dx= 0., dy= 0., dz = 0.;

// lennard jones potential energy
double r = 0.0;  // distance between particle i and j
for(int i = 0; i< mySys.NPart; i++){
    for(int j = 0; j< mySys.NPart; j++){

        if (i != j){
        //r = sqrt( pow(X[i][0]-X[j][0],2)
```

```
        + p[1]-X[j][1],2) + pow(X[i][2]-X[j][2],2));
        dx = X[i][0]-X[j][0];
        dy = X[i][1]-X[j][1];
        dz = X[i][2]-X[j][2];
        dx -= round(dx / mySys.box_x) * mySys.box_x;
        dy -= round(dy / mySys.box_y) * mySys.box_y;
        dz -= round(dz / mySys.box_z) * mySys.box_z;
        r = sqrt( pow(dx,2) + pow(dy,2) + pow(dz,2));

        if(r != 0 && r < mySys.sigma_cut){
        en += 4 * mySys.eps * (pow((mySys.sr),12) - pow((mySys.sigma / r),6));
    }

        else if(r >=  mySys.sigma_cut){
        en +=  0}}}
```

– Energy Updating: Updates the system's energy when a particle is moved and calculates the difference in energy resulting from this change.

```
double update_energy(double **X, double **X_new, double en_old, int i_star){

double en_new = en_old; // was computed on X
double en_diff  = 0;
double dx= 0., dy= 0., dz = 0.;

    // potential energy
    double r = 0.0;  // distance between particle i and j
    for(int i = 0; i< mySys.NPart; i++){

        dx = X[i][0]-X[i_star][0];
        dy = X[i][1]-X[i_star][1];
        dz = X[i][2]-X[i_star][2];

        dx -= round(dx / mySys.box_x) * mySys.box_x;
        dy -= round(dy / mySys.box_y) * mySys.box_y;
        dz -= round(dz / mySys.box_z) * mySys.box_z;

    r = sqrt( pow(dx,2) + pow(dy,2) + pow(dz,2));
        //cout << r << endl;
    //r = sqrt( pow(X[i][0]-X[i_star][0],2) + pow(X[i][1]-X[i_star][1],2)
    +pow(X[i][2]-X[i_star][2],2));
    if (r !=0 && r <= mySys.sigma_cut && r != 0){
        en_new -= 4 * mySys.eps *
        (pow((mySys.sigma / r),12) -pow((mySys.sigma / r),6));
```

```
        en_diff -= 4 * mySys.eps *
        (pow((mySys.sigma / r),12) -pow((mySys.sigma / r),6));
    }
    else if (r > mySys.sigma_cut){
        en_new -= 0;
        en_diff -=0;
    }
}

for(int i = 0; i< mySys.NPart; i++){

        dx = X_new[i][0]-X_new[i_star][0];
        dy = X_new[i][1]-X_new[i_star][1];
        dz = X_new[i][2]-X_new[i_star][2];
        dx -= round(dx / mySys.box_x) * mySys.box_x;
        dy -= round(dy / mySys.box_y) * mySys.box_y;
        dz -= round(dz / mySys.box_z) * mySys.box_z;
     r = sqrt( pow(dx,2) + pow(dy,2) + pow(dz,2));

    //r = sqrt( pow(X_new[i][0]-X_new[i_star][0],2)
    + pow(X_new[i][1]X_new[i_star][1],2)
    + pow(X_new[i][2]-X_new[i_star][2],2));
    if (r !=0 && r <= mySys.sigma_cut){
        en_new += 4 * mySys.eps * (pow((mySys.sigma / r),12)
        -pow((mySys.sigma / r),6));
        en_diff += 4 * mySys.eps * (pow((mySys.sigma / r),12)
        -pow((mySys.sigma / r),6));
    }
    else if (r > mySys.sigma_cut){
        en_new += 0;
        en_diff += 0;}}
return en_diff;
}
```

– Pressure :The function computes the pressure of the system by summing the contributions of pairwise interactions to the virial term. It then calculates the pressure using both the ideal gas law and the virial term, and adjusts the result with a tail correction for interactions beyond the cut-off distance.

```
double compute_Pressure(double **X){
double virial = 0.0;          // const + virial term
double pressure = 0.0 ;  // tail correction
double dx= 0., dy= 0., dz = 0.;
```

```
// lennard jones potential energy
double rij = 0.0;  // distance between particle i and j
    for(int i = 0; i< mySys.NPart; i++){
        for(int j = 0; j <i; j++){
            if (j != i) {
                dx = X[i][0]-X[j][0];
                dy = X[i][1]-X[j][1];
                dz = X[i][2]-X[j][2];
                dx -= round(dx / mySys.box_x) * mySys.box_x;
                dy -= round(dy / mySys.box_y) * mySys.box_y;
                dz -= round(dz / mySys.box_z) * mySys.box_z;
                rij = sqrt( pow(dx,2) + pow(dy,2) + pow(dz,2));
    virial +=  48 * mySys.eps * ( 0.5*pow(mySys.sigma / rij,6) -
    pow(mySys.sigma / rij, 12));}}}

//Pres += mySys.NPart * mySys.T / pow(mySys.box_x,3); //
pressure = (mySys.NPart*mySys.T - virial/3.)/
(mySys.box_x*mySys.box_y*mySys.box_z);
// Add the tail correction term
 pressure += (16.0 / 3.0) * M_PI * pow(mySys.density, 2.) *
 (2. / 3.  pow(mySys.sigma / mySys.sigma_cut, 9.)
 - pow(mySys.sigma  mySys.sigma_cut, 3.));

// cout << mySys.density << "  " << rij << "   " << mySys.eps << endl;
return pressure;
}
```

- Montecarlo simulations:

```
time_t startTime = time(0);
   for(int real = 0; real < mySys.realiz; real++){
   // ********* Call the initialization function *********
       init_Positions(X);
       copy(X,X_new);
       mySys.energy = compute_energy(X);
       cout << "realiz -> " << real << endl;
           for(int t = 0; t < mySys.NSteps; t++){
               for (int i = 0; i < mySys.NPart; i++){
                   // Pick one rnd particle
                   i_star = ran38(&(mySys.seed)) * mySys.NPart;
                   // For each direction propose a displacement rnd in
                   [ disp_max, disp_max]
```

```
    for (int j = 0; j < 3; j++){
            proposed_position = X[i_star][j] - mySys.disp_max +
            ran38(&(mySys.seed)) * (2.0 * mySys.disp_max);
            // Apply PBC
            X_new[i_star][j] = proposed_position
            - floor(proposed_position / mySys.box_x) *
            mySys.box_x;}

    // Metropolis test
    //mySys.energy_curr = update_energy(X, X_new,
    mySys.energy, i_star);
    diff_ene = update_energy(X, X_new, mySys.energy, i_star);
    // mySys.energy_curr - mySys.energy ;
    ratio = exp((-diff_ene)/beta);
    u = ran38(&(mySys.seed));
    // uniform real in [0,1]

    // 1) = check metropolis
    if (diff_ene <= 0.0 || u < ratio) {
    for (int j = 0; j < 3; j++){
            temp = X_new[i_star][j];
            X[i_star][j] = temp;
        }
        // update energy
        mySys.energy += diff_ene;

        // update acceptance ratio
        accept_ratio_t[t] += 1;}

    // if 1) = false
    else {
        for (int j = 0; j < 3; j++){
            // if the test fails, restore the X_new array
            pre-displacement-values
            temp = X[i_star][j];
            X_new[i_star][j] = temp;}}}

// keep track of the new energy at every timestep
energy_t[t] += mySys.energy / mySys.realiz;

// write new configuration to file
if (real==0 && nP==2 ){
    filemc << mySys.NPart << endl; //  ovito format requirement
    filemc << endl;                //  ovito format requirement
```

```
                for (int n = 0; n < mySys.NPart; n++){
                    filemc << "S" << n  << "\t" <<  X[n][0] <<
                    "\t" << X[n][1] << "\t" << X[n][2] << endl;
                }}} // close timesteps
            // compute pressure at the end of every realization
            P += compute_Pressure(X) / mySys.realiz;
            } // close realiz
        time_t endTime = time(0);

        // total simulation time
        cout << "total simulation time: " << endTime - startTime << endl;

        // update pressure
        filemc5 << mySys.density << "\t" << P << endl;
        cout << "pressure -> " << P << endl;
    } // close density loop
    // write acceptance ratio to file
    for (int i = 0; i < mySys.NSteps ; i++){
        filemc3 << accept_ratio_t[i]/ (mySys.realiz * mySys.NPart) << endl;
        }

    // write energy per timestep to file
    for (int i = 0; i < mySys.NSteps ; i++){
        filemc4 << energy_t[i] << endl;
    }


        // ********* Deallocate memory for the matrix *********
        for (int i = 0; i <  mySys.NPart; i++)
        {
            free(X[i]);
        }
        free(X);
```

In the end I will show a comparison between data generated by this simulation and data given in the excercise instruction.

9 A10 Integration Schemes

9.1 Exercise 1

9.1.1 Request a

Consider the first integrator (Euler integrator):

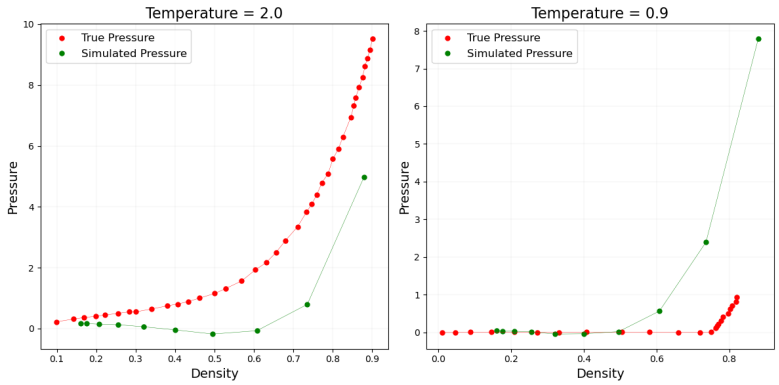


Figure 29: Comparison between Lennard-Jones with given data and the algorithm above.

$$\begin{aligned} p(t+\Delta t) &= p(t) - x(t)\Delta t \\ x(t+\Delta t) &= x(t) + p(t)\Delta t \end{aligned} \tag{69}$$

This can be expressed in matrix form as:

$$\begin{pmatrix} x(t+\Delta t) \\ p(t+\Delta t) \end{pmatrix} = \begin{pmatrix} 1 & \Delta t \\ -\Delta t & 1 \end{pmatrix} \begin{pmatrix} x(t) \\ p(t) \end{pmatrix} \tag{70}$$

Let $\mathbf{Y} = \begin{pmatrix} x \\ p \end{pmatrix}$ and the transformation matrix M be defined as:

$$M = \begin{pmatrix} 1 & \Delta t \\ -\Delta t & 1 \end{pmatrix}$$

The determinant of M can be computed as:

$$\det(M) = 1 \cdot 1 - (-\Delta t) \cdot \Delta t = 1 + \Delta t^2 \tag{71}$$

Since $\Delta t^2 > 0$, we have $\det(M) > 1$.
Now, consider the second integrator:

$$\begin{aligned} p(t+\Delta t) &= p(t) - x(t)\Delta t \\ x(t+\Delta t) &= x(t) + p(t+\Delta t)\Delta t \end{aligned} \tag{72}$$

In matrix form, this is represented as:

$$\begin{pmatrix} x(t+\Delta t) \\ p(t+\Delta t) \end{pmatrix} = \begin{pmatrix} 1-\Delta t^2 & \Delta t \\ -\Delta t & 1 \end{pmatrix} \begin{pmatrix} x(t) \\ p(t) \end{pmatrix} \tag{73}$$

The transformation matrix for this integrator is:

$$M' = \begin{pmatrix} 1-\Delta t^2 & \Delta t \\ -\Delta t & 1 \end{pmatrix}$$

The determinant of M' is given by:

$$\det(M') = 1 - \Delta t^2 - \Delta t^2 = 1 \tag{74}$$

For the first integrator, we find that $\det(M) > 1$. This indicates that the volume of the phase space is expanding under the dynamics defined by this integrator. An expanding volume in phase space often leads to numerical instability, which can cause the system to diverge from its true trajectory over time.

On the other hand, for the second integrator, we find that $\det(M') = 1$. This implies that the volume of the phase space remains constant. This is a desirable property in numerical integration as it suggests that the integrator preserves the symplectic structure of the phase space, leading to better long-term stability and conservation properties.

In conclusion, the differences in the determinants tell us about the stability and the qualitative behavior of the two algorithms when integrated over time. The first integrator may lead to growth in numerical errors, while the second integrator better maintains the physical properties of the system.

9.1.2 Request b

Let's consider Euler symplectic integrator:

$$\begin{aligned} p(t + \Delta t) &= p(t) - x(t)\Delta t \\ x(t + \Delta t) &= x(t) + p(t + \Delta t)\Delta t \end{aligned} \tag{75}$$

We want to identify an Hamiltonian which is constant in time, this means $H'(t) = H'(t + \Delta t)$, so we should find $H(t + \Delta t) = H(t)$ and also the term $\frac{px}{2}$ should conserve in time . Let's veirfy it:

$$H'(t + \Delta t) = \frac{p(t + \Delta t)^2}{2} + \frac{x(t + \Delta t)^2}{2} - \frac{p(t + \Delta t)x(t + \Delta t)\Delta t}{2} \tag{76}$$

$$= \frac{1}{2}\left[(p(t) - x(t)\Delta t)^2 + (x(t)(1 - \Delta t^2) + p(t)\Delta t)\right] - \tag{77}$$

$$\frac{(p(t) - x(t)\Delta t)(x(t)(1 - \Delta t^2) + p(t)\Delta t)\Delta t}{2} \tag{78}$$

Expanding the kinetic term:

$$(p(t) - x(t)\Delta t)^2 = p(t)^2 - 2p(t)x(t)\Delta t + x(t)^2\Delta t^2$$

Now, expanding the potential term:

$$(x(t) + p(t)\Delta t - x(t)\Delta t^2)^2 = x(t)^2 + 2x(t)(p(t)\Delta t - x(t)\Delta t^2) + (p(t)\Delta t - x(t)\Delta t^2)^2$$

If we do the same for the last term:

$$\frac{p(t + \Delta t)x(t + \Delta t)\Delta t}{2} = \frac{\Delta t}{2}[(p(t) - x(t)\Delta t)(x(t)(1 - \Delta t^2) + p(t)\Delta t)\Delta t] \tag{79}$$

$$\sim \frac{p(t)x(t)}{2}\Delta t + O(\Delta t^2) \tag{80}$$

taking out terms of higher order than Δt since it is going to zero.This verifies that the Hamiltonian H' is conserved in time when using the symplectic integrator. Therefore, we conclude that:

$$H'(t) = H'(t + \Delta t)$$

This demonstrates that the shadow Hamiltonian defined retains its form and is constant in time under the dynamics specified by the Euler symplectic integrator.

9.1.3 Request c

Code in python:

```
import numpy as np
import matplotlib.pyplot as plt

#Parameters
delta_t1 = 1e-3
delta_t2 = 1e-2
T = 10
n_steps1 = int(T / delta_t1)
n_steps2 = int(T / delta_t2)

# Initial Conditions
x0 = 1.0
p0 = 0.0

def exact_solution(t):
    """Exact solution per x(t) e p(t)"""
    x_exact = np.cos(t)
    p_exact = -np.sin(t)
    return x_exact, p_exact

def euler_integrator(delta_t, n_steps):
    """Standard Euler integration"""
    x = np.zeros(n_steps + 1)
    p = np.zeros(n_steps + 1)
    t = np.linspace(0, T, n_steps + 1)

    x[0] = x0
    p[0] = p0

    for i in range(n_steps):
        p[i + 1] = p[i] - x[i] * delta_t
        x[i + 1] = x[i] + p[i] * delta_t
```

```
    return x, p, t

def symplectic_euler_integrator(delta_t, n_steps):
    """Simplectic Euler integration"""
    x = np.zeros(n_steps + 1)
    p = np.zeros(n_steps + 1)
    t = np.linspace(0, T, n_steps + 1)

    x[0] = x0
    p[0] = p0

    for i in range(n_steps):
        p[i + 1] = p[i] - x[i] * delta_t
        x[i + 1] = x[i] + p[i + 1] * delta_t

    return x, p, t

#Numerical integration
x_euler1, p_euler1, t1 = euler_integrator(delta_t1, n_steps1)
x_symplectic1, p_symplectic1, t1 = symplectic_euler_integrator(delta_t1, n_steps1)

x_euler2, p_euler2, t2 = euler_integrator(delta_t2, n_steps2)
x_symplectic2, p_symplectic2, t2 = symplectic_euler_integrator(delta_t2, n_steps2)

# exact solution
x_exact1, p_exact1 = exact_solution(t1)
x_exact2, p_exact2 = exact_solution(t2)

#relativ error
error_euler1 = np.abs((x_euler1 - x_exact1) / x_exact1)
error_symplectic1 = np.abs((x_symplectic1 - x_exact1) / x_exact1)
print(error_euler1)
print(error_symplectic1)
error_euler2 = np.abs((x_euler2 - x_exact2) / x_exact2)
error_symplectic2 = np.abs((x_symplectic2 - x_exact2) / x_exact2)
print(error_symplectic2)
```

Result: As expected and we can observe it in Fig.29 we have that the more accurate integrator is the Euler symplectic one.

9.1.4 Request d

Code:

```
import numpy as np
```

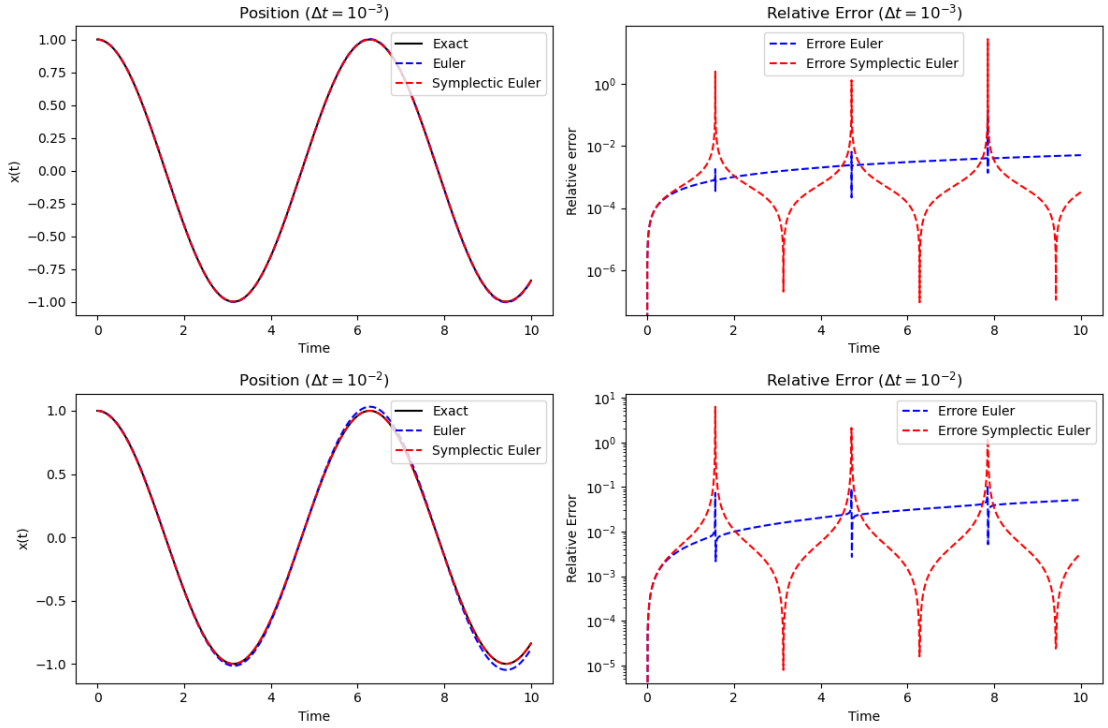


Figure 30: Harmonic oscillator solutions found as the exact solution, numerical simulation with euler integrator, numerical integration with euler symplectic integrator.

```
import matplotlib.pyplot as plt

# Parameters
dt1 = 1e-3
dt2 = 1e-2
T = 10
n_steps1 = int(T / dt1)
n_steps2 = int(T / dt2)
t1 = np.linspace(0, T, n_steps1 + 1)
t2 = np.linspace(0, T, n_steps2 + 1)

# Initial conditions
x0 = 1.0
p0 = 0.0

# Functions to calculate the Hamiltonian energy H and the shadow Hamiltonian H_tilde
def compute_H(x, p):
    return 0.5 * p**2 + 0.5 * x**2
```

```
def compute_H_tilde(x, p, dt):
    return 0.5 * p**2 + 0.5 * x**2 - p * x * dt

# Initialize arrays to store the results
x_euler1, p_euler1 = np.zeros(n_steps1 + 1), np.zeros(n_steps1 + 1)
x_symplectic1, p_symplectic1 = np.zeros(n_steps1 + 1), np.zeros(n_steps1 + 1)
H_euler1, H_symplectic1, H_tilde1 = np.zeros(n_steps1 + 1), np.zeros(n_steps1 + 1), np.ze

x_euler2, p_euler2 = np.zeros(n_steps2 + 1), np.zeros(n_steps2 + 1)
x_symplectic2, p_symplectic2 = np.zeros(n_steps2 + 1), np.zeros(n_steps2 + 1)
H_euler2, H_symplectic2, H_tilde2 = np.zeros(n_steps2 + 1), np.zeros(n_steps2 + 1), np.ze

# Initial conditions
x_euler1[0], p_euler1[0] = x0, p0
x_symplectic1[0], p_symplectic1[0] = x0, p0
H_euler1[0] = compute_H(x0, p0)
H_symplectic1[0] = compute_H(x0, p0)
H_tilde1[0] = compute_H_tilde(x0, p0, dt1)

x_euler2[0], p_euler2[0] = x0, p0
x_symplectic2[0], p_symplectic2[0] = x0, p0
H_euler2[0] = compute_H(x0, p0)
H_symplectic2[0] = compute_H(x0, p0)
H_tilde2[0] = compute_H_tilde(x0, p0, dt2)

# Integration using classical Euler and symplectic Euler methods for dt1
for i in range(n_steps1):
    # Classical Euler
    p_euler1[i + 1] = p_euler1[i] - x_euler1[i] * dt1
    x_euler1[i + 1] = x_euler1[i] + p_euler1[i] * dt1
    H_euler1[i + 1] = compute_H(x_euler1[i + 1], p_euler1[i + 1])

    # Symplectic Euler
    p_symplectic1[i + 1] = p_symplectic1[i] - x_symplectic1[i] * dt1
    x_symplectic1[i + 1] = x_symplectic1[i] + p_symplectic1[i + 1] * dt1
    H_symplectic1[i + 1] = compute_H(x_symplectic1[i + 1], p_symplectic1[i + 1])
    H_tilde1[i + 1] = compute_H_tilde(x_symplectic1[i + 1], p_symplectic1[i + 1], dt1)

# Integration using classical Euler and symplectic Euler methods for dt2
for i in range(n_steps2):
    # Classical Euler
    p_euler2[i + 1] = p_euler2[i] - x_euler2[i] * dt2
    x_euler2[i + 1] = x_euler2[i] + p_euler2[i] * dt2
```

```
H_euler2[i + 1] = compute_H(x_euler2[i + 1], p_euler2[i + 1])

# Symplectic Euler
p_symplectic2[i + 1] = p_symplectic2[i] - x_symplectic2[i] * dt2
x_symplectic2[i + 1] = x_symplectic2[i] + p_symplectic2[i + 1] * dt2
H_symplectic2[i + 1] = compute_H(x_symplectic2[i + 1], p_symplectic2[i + 1])
H_tilde2[i + 1] = compute_H_tilde(x_symplectic2[i + 1], p_symplectic2[i + 1], dt2)
```

Result:

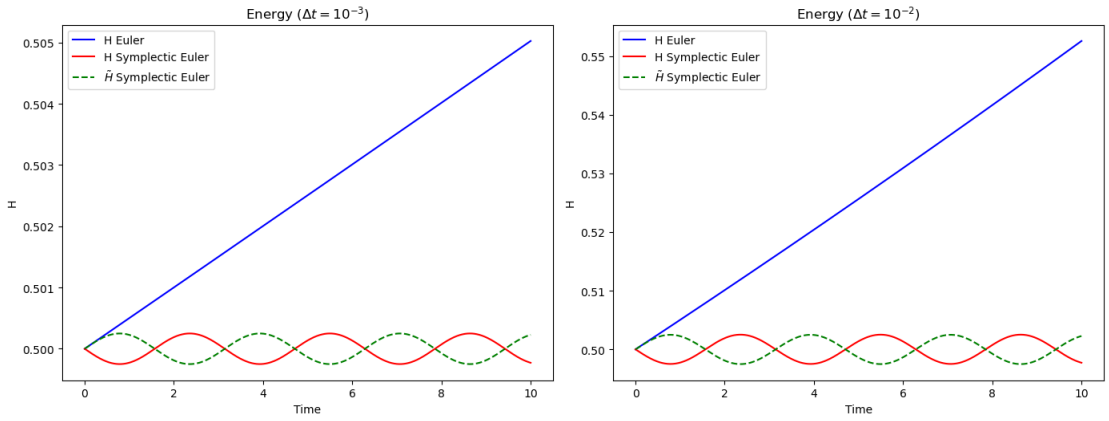


Figure 31: Hamiltonian and Shadow Hamiltonian with Euler symplectic integrator and Hamiltonian with standard Euler integrator.

The behaviour we are observing is correct and reflects a fundamental difference between the two integratos we are comparing. The classical Euler integrator is not a symplectic integrator, meaning it does not preserve the phase-space structure of Hamiltonian systems over time. For Hamiltonian systems, one of the main consequences of this is that energy (the Hamiltonian) is not conserved. This results in the Hamiltonian computed with the classical Euler integrator increasing linearly over time, which is a clear sign of "long-term energy drift." The symplectic Euler integrator, on the other hand, preserves the system's symplectic structure, meaning that, while it doesn't exactly conserve the energy at every step, it preserves the general properties of the Hamiltonian system. Both of hamiltonians oscillates around a constant value and do not show any energy drift.

9.2 Exercise 2

The system being simulated is a harmonic oscillator, governed by the equation of motion:

$$\ddot{x} = -\frac{k}{m}x \tag{81}$$

where x is the position of the particle, k is the spring costant, m is the mass of the particle.

9.2.1 Verlet Integration

Verlet integration is a method to integrate the equations of motion without explicitly using velocities. Insteat, it updates positions based on the current position and the one from the previous time step. It is an energy conserving method that's stable for simulating systems with periodic motion. The update rule is:

$$x_{i+1} = 2x_i - x_{i-1} - \frac{k}{m} x_i \Delta t^2 \tag{82}$$

velocities are not directly calculated with this method, but can be approximated by a finite difference: $v_i = \frac{x_{i+1} - x_{i-1}}{2\Delta t}$. Code:

```
# Verlet Algorithm
def verlet(x0, v0, dt, n_steps):
    x = np.zeros(n_steps)
    v = np.zeros(n_steps)

    x[0] = x0
    x[1] = x0 + v0 * dt - 0.5 * k/m * x0 * dt**2  # Use initial velocity to compute x[1]

    for i in range(1, n_steps - 1):
        x[i+1] = 2 * x[i] - x[i-1] - k/m * x[i] * dt**2

    # Velocity can only be computed from the second to the second-to-last point
    v[1:-1] = (x[2:] - x[:-2]) / (2 * dt)

    return x, v
```

9.2.2 Velocity Verlet Integration

The velocity Verlet method is an improved version of Verlet that explicitly tracks boyh positions and velocities. It provides better accuracy and direct access to velocities. The updates rules are:

- position

$$x_{i+1} = x_i + v_i \Delta t + \frac{1}{2} a_i \Delta t^2 \tag{83}$$

  where $a_i = -\frac{k}{m} x_i$;

- acceleration

$$a_{i+1} = -\frac{k}{m} x_{i+1} \tag{84}$$

- velocity

$$v_{i+1} = v_i + \frac{1}{2}(a_i + a_{i+1})\Delta t \tag{85}$$

Code:

```
# Velocity Verlet Algorithm
def velocity_verlet(x0, v0, dt, n_steps):
    x = np.zeros(n_steps)
    v = np.zeros(n_steps)
    a = np.zeros(n_steps)

    # Initial conditions
    x[0] = x0
    v[0] = v0
    a[0] = -k/m * x0  # Initial force (acceleration)

    for i in range(n_steps - 1):
        # Step 1: Half kick (update momentum)
        v[i] += 0.5 * a[i] * dt  # Half-step velocity update

        # Step 2: Full step position update (drift)
        x[i + 1] = x[i] + v[i] * dt  # Update position

        # Step 3: Compute forces at the new position
        a_next = -k/m * x[i + 1]  # Compute new acceleration (force)

        # Step 4: Complete the kick (update momentum)
        v[i + 1] = v[i] + 0.5 * (a[i] + a_next) * dt  # Full-step velocity update
        a[i + 1] = a_next  # Store the new acceleration for next iteration

    return x, v
```

9.3 Exercise 3

Code:

```
import numpy as np
import matplotlib.pyplot as plt

# Constants
omega = 1 # Angular frequency
dt = 0.1  # Time step
T = 10 # Total time
n_steps = int(T / dt)
```

```
# Initial conditions
q0 = 1.0  # Initial position
p0 = 0  # Initial momentum

# Exact solution for comparison
def exact_solution(t):
    return q0 * np.cos(omega * t)+(p0/omega)*np.sin(omega*t),
    p0 * np.cos(omega * t) - q0 * omega * np.sin(omega * t)

# Corrected Velocity Verlet Algorithm
def velocity_verlet(q0, p0, dt, n_steps):
    q = np.zeros(n_steps)
    p = np.zeros(n_steps)
    E = np.zeros(n_steps)

    q[0] = q0
    p[0] = p0
    E[0] = 0.5 * (p[0]**2 + omega**2 * q[0]**2)  # Initial energy

    for i in range(n_steps - 1):
        p_half = p[i] - 0.5 * omega**2 * q[i] * dt  # Half-step momentum
        q[i + 1] = q[i] + p_half * dt               # Full step position
        p[i + 1] = p_half - 0.5 * omega**2 * q[i + 1] * dt  # Full step momentum

        E[i + 1] = 0.5 * (p[i + 1]**2 + omega**2 * q[i + 1]**2)  # Energy at each
        time step

    return q, p, E

# Standard Verlet Algorithm
def standard_verlet(q0, p0, dt, n_steps):
    q = np.zeros(n_steps)
    p = np.zeros(n_steps)
    E = np.zeros(n_steps)

    q[0] = q0
    p[0] = p0
    E[0] = 0.5 * (p[0]**2 + omega**2 * q[0]**2)  # Initial energy

    # First step position calculation
    q[1] = q0 + p0 * dt - 0.5 * omega**2 * q0 * dt**2

    for i in range(1, n_steps - 1):
        q[i + 1] = 2 * q[i] - q[i - 1] - omega**2 * q[i] * dt**2  # Update position
```

```
        E[i] = 0.5 * (p[i]**2 + omega**2 * q[i]**2)
    # Velocity calculation using central difference
    p[1:-1] = (q[2:] - q[:-2]) / (2 * dt)



    return q, p, E

# Run simulations
q_velocity_verlet, p_velocity_verlet, E_velocity_verlet =
velocity_verlet(q0, p0, dt, n_steps)
q_standard_verlet, p_standard_verlet, E_standard_verlet =
standard_verlet(q0, p0, dt, n_steps)

# Time array for plotting
t = np.linspace(0, T, n_steps)

# Exact solution for comparison
q_exact, p_exact = exact_solution(t)

# Calculate relative energy difference
E0_velocity = E_velocity_verlet[0]  # Initial energy for velocity verlet
relative_energy_difference_velocity_verlet =
abs(E_velocity_verlet - E0_velocity) / E0_velocity

E0_standard = E_standard_verlet[0]  # Initial energy for standard verlet
relative_energy_difference_standard =
abs(E_standard_verlet - E0_standard) / E0_standard
```

Result: we can clearly observe that Verlet velocity algorithm has an energy discrepancy that is an order lower than standard Verlet algorithm. Instead, even if the Verlet velocity algorithm is more stable, they both show the same discrepancy to exact solution that increases in time step

10 A11 Interaction potential and thermostats

10.1 Exercise 1

The temperature of the system T_K can be linked to the kinetic energy K by the relation:

$$< T_K >= \frac{2 < K >}{3Nk_B} \tag{86}$$

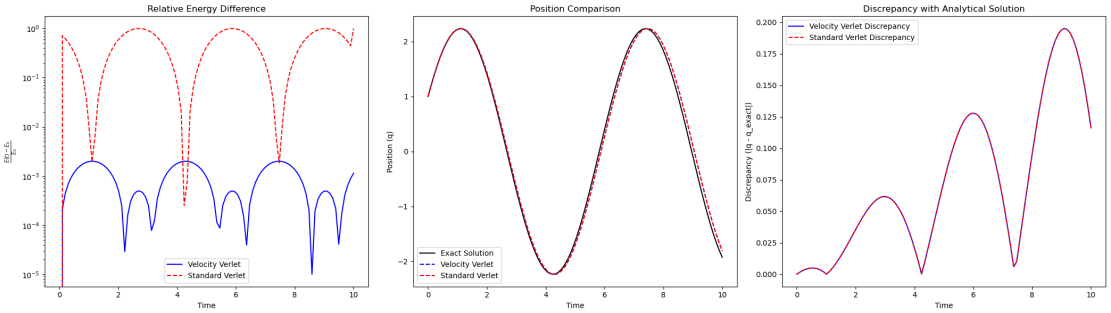


Figure 32: Comparison between Verlet Velocity and Verlet algorithm

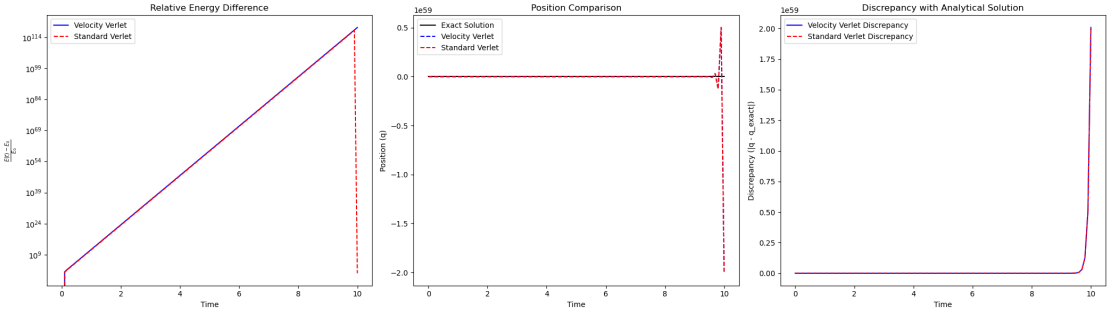


Figure 33: Comparison between Verlet Velocity and Verlet algorithm for $\omega\Delta t > 2$,$\Delta t = 0,1$ and $\omega = 25$

where k_B is the Boltzmann constant and N is the number of particles. The temperature fluctuation is given by the variance:

$$\sigma^2_{T_K} =< T_K^2 > - < T_K >^2 \tag{87}$$

The average kinetic energy $< K >$ for a classical system in three dimensions is given by:

$$< K >= \frac{3}{2} N k_B < T_K > \tag{88}$$

Since the fluctuation of the kinetic energy is given by:

$$\sigma^2_K =< K^2 > - < K >^2 \tag{89}$$

the temperature fluctuations can be related to kinetic energy fluctuations as:

$$\sigma^2_{T_K} = \frac{4}{9N^2 k_B^2} \sigma^2_K \tag{90}$$

For a classical ideal gas in the canonical ensemble it is known that:

$$\sigma^2_K = \frac{2}{3} < K >^2 \tag{91}$$

Then:

$$\sigma_{T_K}^2 = \frac{4}{9N^2k_B^2} \cdot \frac{2}{3}\left(\frac{3}{2}Nk_B\langle T_K\rangle\right)^2 \tag{92}$$

$$= \frac{2}{3}\langle T_K\rangle^2 \tag{93}$$

Finally, the relative fluctuation in temperature is:

$$\frac{\sigma_{T_K}^2}{< T_K >^2} = \frac{2}{3N} \tag{94}$$

10.2 Exercise 2

In this exercise er are going to observe simulations of N particles of mass $m = 1$ that are confined to move within a cubic box of length L (with periodic boundary condition), that interact with each other through Lennard-Jones potential:

$$V_{LJ}(r) = 4\epsilon((\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^6) \tag{95}$$

where $\sigma = 1$, $\epsilon = 1$. In these simulations, as suggestes $L = 10$ and $\rho = \frac{N}{V} = 0,2$. This exercise has been carried on with LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator). This is the program runned on LAMMPS with requirements:

```
# 1) Initialization
units lj
dimension 3
atom_style atomic
pair_style lj/cut 3.92
boundary p p p

# 2) System definition
region simulation_box block -5 5 -5 5 -5 5
create_atoms 1 random 200 341341 simulation_box


# 3) Simulation settings
mass 1 1
pair_coeff 1 1 1.0 1.0


#enforce momentum to zero
velocity all create 1.0 87287 dist gaussian
velocity all zero linear #enforce zero momentum

# 4) Initial Minimization
```

```
minimize 1.0e-4 1.0e-6 1000 10000

# 5) Visualization
thermo 10
thermo_style custom step temp pe ke etotal press

# 6) Set up and run NVE dynamics with Langevin thermostat
fix mynve all nve
fix mylgv all langevin 1.0 1.0 0.1 1530917
timestep 0.005

# 7) Compute and output RDF
compute myRDF all rdf 100
fix rdf all ave/time 50 10 500 c_myRDF[*] file rdf15_output.dat mode vector

# 8) Run the simulation
run 10000
```

This is the result.

The RDF provides insights into the structure and organization of the particle system. The peak at r=sigma suggests that the particles tend to form clusters or bonds with their nearest neighbors. The decay of the RDF at larger distances indicates that the system becomes more disordered at longer length scales. The effect of the cutoff on the RDF highlights the importance of choosing an appropriate cutoff value in simulations involving the Lennard-Jones potential. A too-small cutoff can lead to inaccurate results, while a too-large cutoff can increase the computational cost of the simulation.

10.3 Exercise 3

For thi exercise I choose to simulate Berendsen thermostat and Noosehover thermostat through LAMMPS, the code was the same as the previous one except for this condition:

- Berendsen thermostat

  ```
  # 6) Set up and run dynamics with Berendsen thermostat
  fix berendsen all temp/berendsen 2.0 2.0 0.5
  ```

- Noosehover thermostat:

  ```
  # 6) Set up and run dynamics with Nose-Hoover thermostat
  fix nosehoover all nvt temp 2.0 2.0 0.5
  ```

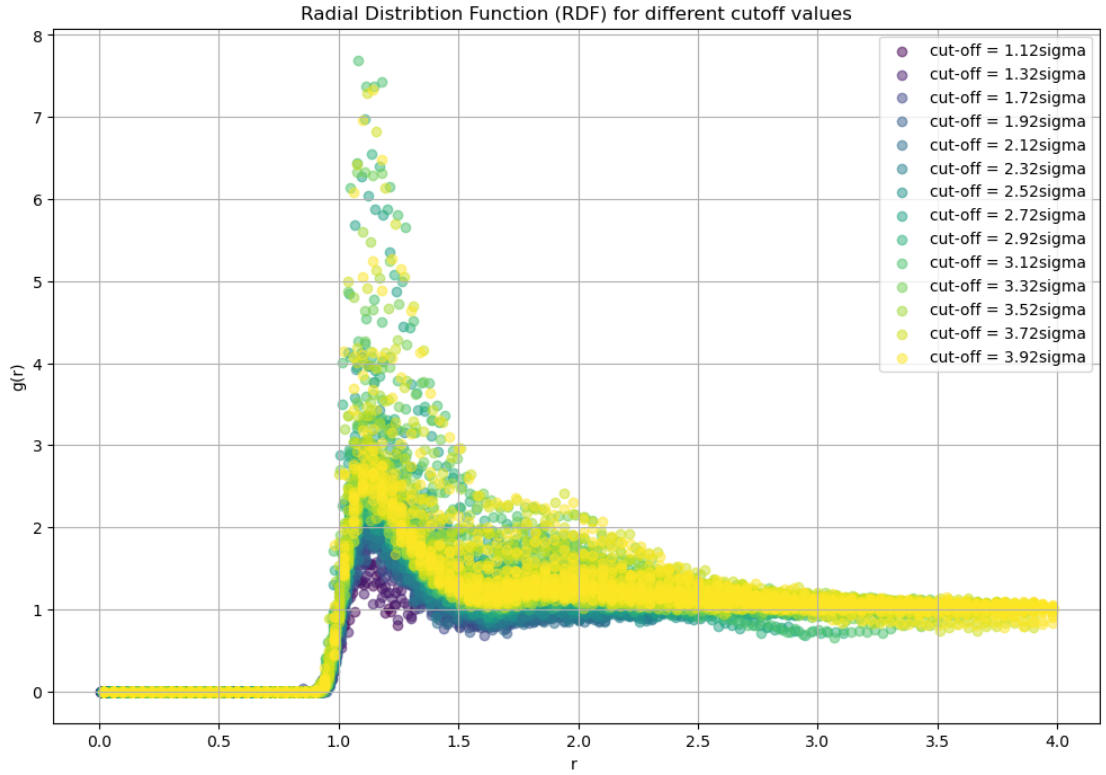


Figure 34: Radial Distribution function for different cut-off values

I run both simulations for $N = [200, 150, 50]$ and this is what I obtain: where consistency

Thermostat	N	Kinetic Energy Consistency	Pressure Consistency	Mean Kinetic Energy	Theoretical Kinetic Energy	Mean Pressure	Theoretical Pressure
Berendsen	200	Yes	Yes	3.37	3.38	0.35	0.34
Nose-Hoover	200	Yes	Yes	2.9	3.31	0.32	0.33
Berendsen	150	Yes	No	3.36	3.38	0.25	0.34
Nose-Hoover	150	Yes	No	3.31	3.33	0.25	0.33
Berendsen	50	No	No	3.32	3.38	0.1	0.34
Nose-Hoover	50	Yes	No	2.9	3.35	0.09	0.33

Figure 35: Table of system information about pressure and Energy consistency

means that I obtain a difference between the value obtained through the simulation and the theoretical one, smaller than 0.02.

The kinetic energy is consistently found to be in agreement with the equipartition theorem for all simulations, regardless of the thermostat or number of particles. This suggests that the thermostats are effectively maintaining the desired temperature and that the system is reaching thermal equilibrium.

The pressure consistency with the virial expansion shows a more complex picture: For

N=200, both thermostats yield consistent results. For N=150, both thermostats deviate from the virial expansion, indicating potential issues in the pressure calculation or the accuracy of the virial expansion at this particle number. For N=50, both thermostats show significant deviations from the virial expansion, suggesting that the virial expansion becomes less accurate at low particle numbers.

Overall, the simulations demonstrate the effectiveness of both Berendsen and Nose-Hoover thermostats in maintaining the desired temperature and ensuring the system reaches thermal equilibrium. However, the pressure calculations exhibit discrepancies with the virial expansion, especially at lower particle numbers. This suggests that caution should be exercised when using the virial expansion for small systems or when analyzing the pressure behavior in this particular simulation setup.

11 A12 Langevin and Brownian Dynamics

11.1 Validity of the Langevin Approach

11.1.1 Minimum size of the object for which Langevin description is acceptable

To approach this problem, I need to establish a connection between the time step of the simulation and the physical properties of the object (specifically its size). The Langevin equation is valid when the correlation time of the heat bath fluctuations (τ_{coll}) is much smaller that the time scales of the dynamics we are interested in. Given that:

- Correlation time of water fluctuations: $\tau_{coll} \approx 1ps$
- Time step of the algorithm $\Delta t \geq 10\tau_{coll} = 10^{11}s - 10^{10}s$

Assumptions:

- Thermal Energy k_BT, where $k_B = 1.38 \times 10^{-23}J/K$ is the Boltzman constant and $T \approx 300K$.
- Density of the object: I assume it has the same density as water ($\rho = 1000kg/m^3$).
- The object is spherical

The time scale fot the momentum relaxation of a particle of mass m is given by:

$$\tau_{relax} = \frac{m}{\gamma} \tag{96}$$

where $\gamma = 6\pi\eta r$ is the friction coefficient for a sphere of radius r, and η is the the viscosity of the fluid (for water $\eta \approx 10^{-3}Pas$).

The mass m of a sphere with density ρ and radius r is:

$$m = \frac{4}{3}\pi r^3\rho \tag{97}$$

Substituting the expression for the mass into formula 96

$$\tau_{relax} = \frac{\frac{4}{3}\pi r^3 \rho}{6\pi\eta r} \tag{98}$$

$$= \frac{2\rho r^2}{9\eta} \tag{99}$$

Since $\tau_{coll} < \tau_{relax}$ and $\Delta t \geq 10\tau_{coll} = 10^{11}s - 10^{10}s$, the condition I take into the account is

$$\frac{2\rho r^2}{9\eta} \geq 10^{-11}s \tag{100}$$

Which allows to observe the condition for r:

$$r \geq \sqrt{\frac{9\eta 10^{-10}}{2\rho}} \tag{101}$$

Substuting the numerical values:

$$r \geq 6.7 \times 10^{-6}m \tag{102}$$

11.1.2 Diffusion time scale for a particle of diameter $\sigma = 10^{-8}m$

The Stokes-Einstein equation for the diffusion coefficient D is:

$$D = \frac{k_B T}{6\pi\eta r} \tag{103}$$

where $r = \frac{\sigma}{2}$. The diffusion time scale for a particle to diffuse over a distance equal to its own diameter is given by:

$$\tau_{diff} = \frac{\sigma^2}{D} \tag{104}$$

Substituting with our data $\tau_{diff} \approx 4.55 \times 10^{-7}s$, while $\tau_{relax} \approx 5.56 \times 10^{-12}s$. Since the condition for the Langein approach is that $\tau_{coll} << \tau_{diff} << \tau_{relax}$ and in this case $\tau_{relax} << \tau_{diff}$ the Langevin description is not compatible with the time scale.

11.1.3 When Brownian motion becomes negligible ($\sigma = 5\mu m$)

When $\sigma = 5\mu m$:

- $\tau_{relax} \approx 1.39 \times 10^{-6}s$
- $\tau_{diff} \approx 2.27 \times 10^{2}s$

Since $\tau_{relax} << \tau_{diff}$ the brownian motion is less relevant compared to viscous forces: For larger particles, viscous forces become dominant compared to Brownian motion.

11.2 Simple Brownian Motion

Brownian motion describes the random motion of particles suspended in a fluid, as a result of their collisions with fast-moving molecules in the fluid. This phenomenon can be observed in many physical systems and is a crucial example of diffusion processes. In this report, we investigate the behavior of Brownian particles in both the overdamped and underdamped limits.

The purpose of this analysis is to simulate the system using the Stochastic Velocity Verlet algorithm and explore how varying the temperature and friction coefficient affects the mean squared displacement (MSD). The simulation is conducted in dimensionless units, with $m = 1$, $\sigma = 1$, and $\epsilon = 1$.

The simulation is conducted within a system of size $L = 10\sigma$, with periodic boundary conditions. We simulate the underdamped limits of Brownian motion. For the underdamped case, the Stochastic Velocity Verlet algorithm is employed.

The velocity Verlet algorithm is modified to account for stochastic forces and friction:

$$x_{t+\Delta t} = x_t + v_t \Delta t + \frac{1}{2} a_t (\Delta t)^2, \tag{105}$$

$$v_{t+\Delta t} = v_t + \frac{1}{2}(a_t + a_{t+\Delta t})\Delta t, \tag{106}$$

where $a = -\gamma v + \eta$, with η representing the stochastic noise term drawn from a Gaussian distribution.

The mean squared displacement is calculated as:

$$\langle x^2(t) \rangle = \frac{1}{N} \sum_{i=1}^{N} x_i(t)^2,$$

where $x_i(t)$ is the position of particle i at time t.

We vary the temperature in the range $0.1 \leq T \leq 2$ and the friction coefficient γ in the range $0.1 \leq \gamma \leq 100$, using 5-10 values for each case.

11.2.1 Mean Squared Displacement for Different Temperatures

Figure 36 shows the MSD as a function of time for several temperatures. As expected, increasing the temperature leads to an increase in MSD, as the particles gain more kinetic energy.

For low temperatures (e.g., $T = 0.1$), the MSD grows slowly over time, while for higher temperatures (e.g., $T = 2$), the particles diffuse more rapidly, leading to a steeper increase in MSD.

11.2.2 Effect of Friction Coefficient

Figure 37 illustrates the effect of varying the friction coefficient γ on the MSD. For low values of γ, the particle motion is less damped, resulting in a faster initial growth of MSD. However, for high γ values, the motion is more diffusive and constrained.

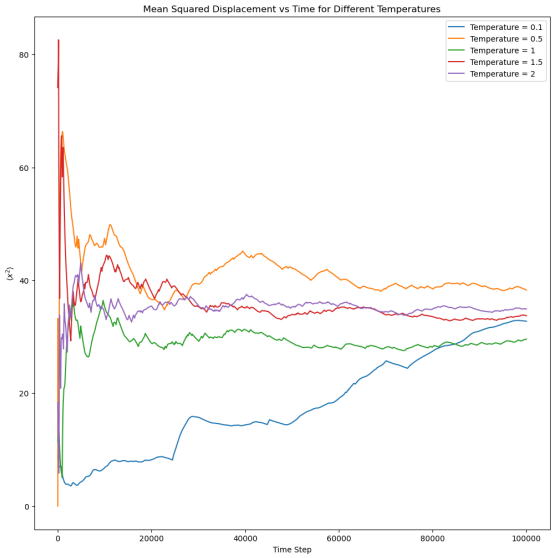


Figure 36: Mean Squared Displacement vs Time for different temperatures.

As the temperature increases, the particles exhibit greater kinetic energy, which is reflected in the steeper MSD curves. This result is consistent with the theoretical prediction that MSD grows faster with higher temperatures.

In terms of friction, lower values lead to more ballistic motion, as there is less resistance to particle movement. On the other hand, high friction introduces stronger damping, which causes the system to transition to a diffusive regime faster.

Further investigations could explore other integrators or extend the analysis to multi-particle systems with interactions.

11.2.3 Diffusion Coefficient

The diffusion coefficient by Green-Kubo relation is given by:

$$D = \int_0^\infty < v(0)v(t) > dt \tag{107}$$

Since the diffusion coefficient is given by the integration of autocorrelation coefficient we can also visualise the velocity autocorrelation function.

Result from the algorthim:

```
Diffusion coefficient (Green-Kubo): 0.009265410954977092
```

11.2.4 Code

Here I report the version with also the calculation of the diffusion coefficient via Green-Kubo relation.

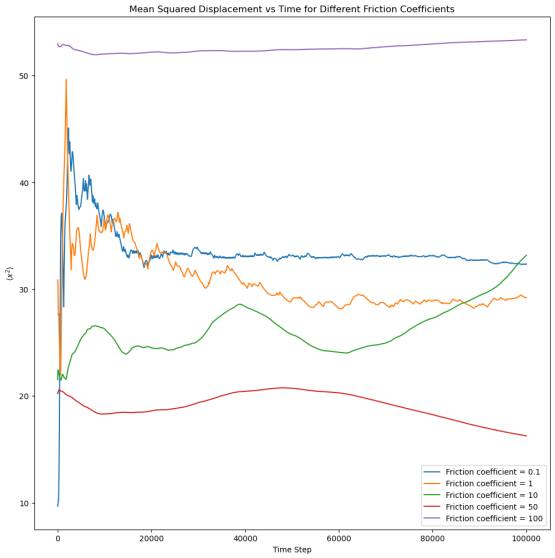


Figure 37: Mean Squared Displacement vs Time for different friction coefficients.

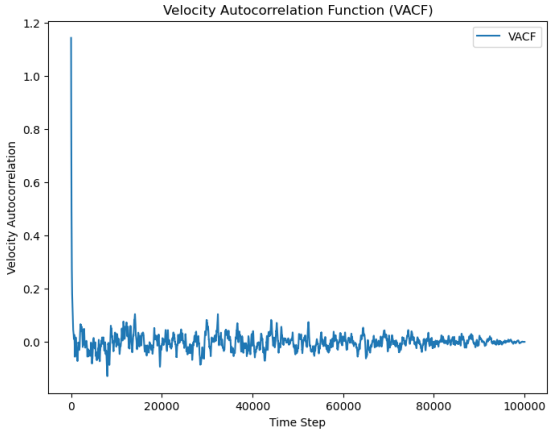


Figure 38: Velocity auto correlation function

```
import numpy as np
import matplotlib.pyplot as plt

def simulate_brownian_motion_with_vacf(T, gamma, L=10, dt=0.01, n_steps=100000):
    # Parameters
    D = T
    m = 1

    # Initial conditions
    x = np.random.uniform(0, L)
```

```
v = np.random.normal(0, np.sqrt(D * dt))
a = - (gamma / m) * v  # Initial acceleration

# Arrays to store results
positions = np.zeros(n_steps)
squared_displacement = np.zeros(n_steps)
velocities = np.zeros(n_steps)  # To store velocities for VACF

for step in range(n_steps):
    # Store position, velocity, and squared displacement
    positions[step] = x
    squared_displacement[step] = x**2
    velocities[step] = v

    # Update position
    x = x + v * dt + 0.5 * a * (dt ** 2)

    # Compute new acceleration
    xi = np.random.normal(0, 1)
    a_new = - (gamma / m) * v + np.sqrt(2 * D / dt) * xi

    # Update velocity
    v = v + 0.5 * (a + a_new) * dt

    # Update acceleration for the next step
    a = a_new

    # Apply periodic boundary conditions
    if x < 0:
        x += L
    elif x >= L:
        x -= L

# Calculate mean squared displacement <x^2> over time
mean_squared_displacement = np.cumsum(squared_displacement) / (np.arange(n_steps) + 1

# Compute the velocity autocorrelation function (VACF)
vacf = np.correlate(velocities, velocities, mode='full') / n_steps
vacf = vacf[n_steps - 1:]  # Take the positive lag part

# Calculate the diffusion coefficient using Green-Kubo relation (integral of VACF)
diffusion_coefficient = np.sum(vacf) * dt

return mean_squared_displacement, vacf, diffusion_coefficient
```

```
# Simulate Brownian motion with VACF for temperature T=1 and gamma=1
msd, vacf, diffusion_coefficient = simulate_brownian_motion_with_vacf(T=1, gamma=1)
```

11.3 Overdamped colloid in an harmonic trap

In this exercise, I have simulated the dynamics of an overdamped particle in a two-dimensional harmonic trap, centered at a point $\mathbf{r}_0$, under periodic boundary conditions. The aim of the simulation was to study the behavior of the particle's position as a function of the trap stiffness K, friction coefficient γ, and temperature T.

The governing potential for the harmonic trap is given by:

$$V(\mathbf{r}) = \frac{1}{2}K(\mathbf{r} - \mathbf{r}_0)^2, \tag{108}$$

where K is the trap stiffness and $\mathbf{r}_0$ is the center of the trap.

The simulation was performed using Langevin dynamics in the overdamped regime, with the particle's velocities initialized from the equilibrium distribution. The periodic boundary conditions ensured that the particle remains within the simulation box.

We model the motion of a single particle subjected to the forces from the harmonic trap and random thermal noise. The Langevin equation describing the motion of the particle in two dimensions is:

$$\mathbf{v} = -\gamma\mathbf{v} - \frac{K}{\gamma}(\mathbf{r} - \mathbf{r}_0) + \sqrt{\frac{2T}{\gamma}}\xi(t), \tag{109}$$

where:

- $\mathbf{v}$ is the particle velocity,
- γ is the friction coefficient,
- K is the trap stiffness,
- T is the temperature,
- $\xi(t)$ represents Gaussian random noise with zero mean and unit variance.

The particle's position was updated at each time step, and periodic boundary conditions were applied to ensure the particle remains within the simulation box of size L.

The particle's motion was simulated using the following steps:

1. Initialize the position $\mathbf{r}(0)$ of the particle and its velocity from the equilibrium distribution.
2. At each time step, update the velocity using the Langevin equation:

$$\mathbf{v} \leftarrow \mathbf{v} - \gamma\mathbf{v}\,\Delta t - \frac{K}{\gamma}(\mathbf{r} - \mathbf{r}_0)\,\Delta t + \sqrt{\frac{2T}{\gamma}}\,\xi(t)\,\sqrt{\Delta t}. \tag{110}$$

3. Update the particle's position:

$$\mathbf{r} \leftarrow \mathbf{r} + \mathbf{v}\,\Delta t. \tag{111}$$

4. Apply periodic boundary conditions to ensure $\mathbf{r} \in [0, L]$.

The code used for the simulation is given below:

```
import numpy as np
import matplotlib.pyplot as plt

# Parameters
L = 20
m = 1
epsilon = 1
n_steps = 100000
dt = 0.01

def simulate_particle(K, gamma, T, initial_position):
    # Initialize
    x = initial_position
    v = np.random.normal(0, np.sqrt(2*T),2)
    r0 = np.array([5,5])

    # Store positions and velocities
    positions = np.zeros((n_steps, 2))

    for step in range(n_steps):
        # Store position
        positions[step] = x

        # Langevin dynamics update
        xi = np.random.normal(0, 1, 2)

        v = v - gamma*v*dt -  (K) * (x - r0) * dt / gamma + np.sqrt(2 * T / gamma) * xi *
        x = x + v * dt

        # Apply periodic boundary conditions
        x = np.mod(x, L)

    return positions
```

The positions of the particle were recorded at each time step, and the average position and variance along one axis were computed. These quantities were studied as functions

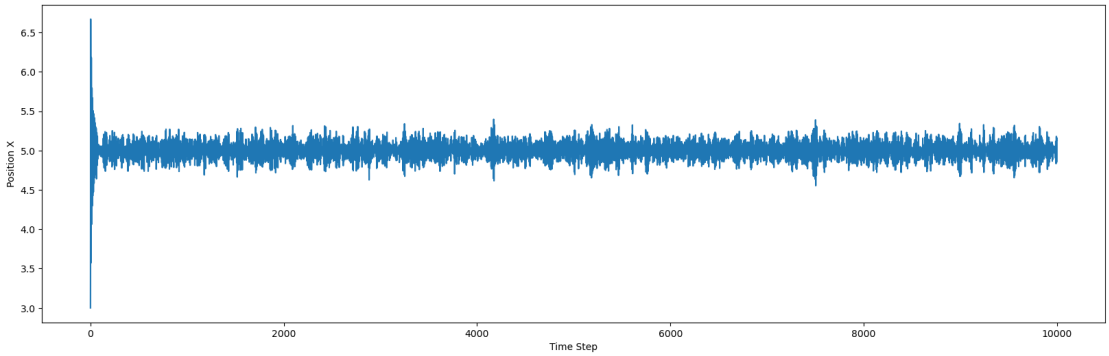
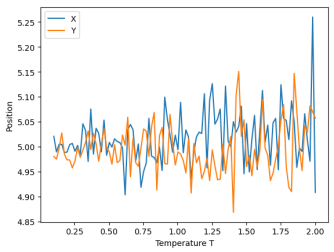


Figure 39: Position of the overdamped particle in the harmonic trap along the x-axis over time.

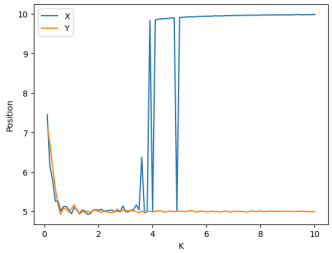
of K, γ, and T. Below is a sample plot of the particle's position along the x-axis over time.

We observed that as K increases, the particle tends to remain closer to the center of the harmonic trap, reducing the variance of the position. Similarly, increasing γ causes the particle to experience stronger damping, leading to slower motion and reduced fluctuation from the trap center. Increasing the temperature T results in more random motion, as expected.

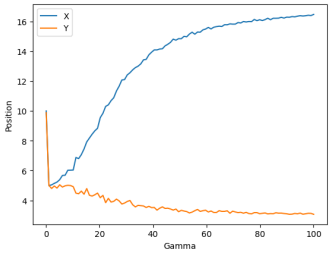
As required the netxt step is to vary T, K, and γ to observe what happens to position and its variance by varyung this parmeters.



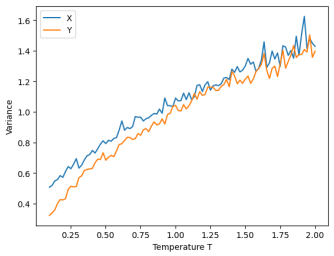
(a) T vs average position



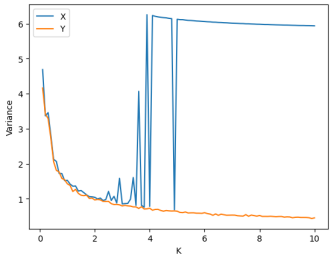
(b) K vs average position



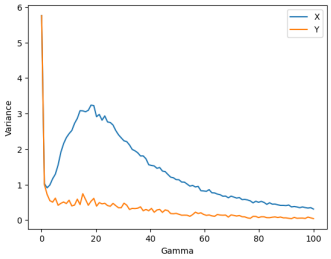
(c) γ vs average position



(d) T vs position variance



(e) K vs position variance



(f) γ vs Position variance

Figure 40: Average position as function of T, K and γ.

Let's comment the behaviour of the system:

- in Fig.40a we can observ that increasing the temperature the position oscillstes more, and this is coherent with the fact that variance increase with temperature as we can see in Fig.40d;
- A higher trap stiffness results in a narrower distribution of the particle's position around the equilibrium point. This is because a stiffer trap exerts a stronger restoring force, limiting the particle's movement. (Fig.40b, Fig.40e)
- a higher friction coefficient dampens the particle's motion, causing it to spend more time near the equilibrium point. This is evident in the reduced fluctuations in the average position, as we can observe in Fig.40c. The variance of the position decreases with increasing friction. This is due to the damping effect, which reduces the particle's ability to deviate significantly from the equilibrium point.

The second requirment was to build another harmonic trap and to observe time swapping between the two: this is what I've implemented:

```
import numpy as np
import matplotlib.pyplot as plt

def potential(r):
    K = 1
    r0 = np.array([0, 0])
    return 0.5 * K * np.sum((r - r0)**2)

def simulate_fpt(initial_position, num_trials=1000, dt=0.01):
    fpt_times = []

    for _ in range(num_trials):
        position = np.array(initial_position)
        time = 0

        while True:
            # Random displacement due to thermal fluctuations
            noise = np.random.normal(0, np.sqrt(dt), size=2)
            position += noise

            # Apply periodic boundary conditions
            position = position % 20

            # Check distances to traps
            dist_to_first = np.linalg.norm(position - np.array([0, 0]))
            dist_to_second = np.linalg.norm(position - np.array([2, 0]))
```

```
            if dist_to_second < dist_to_first:
                fpt_times.append(time)
                break

            time += dt

    return np.mean(fpt_times), np.var(fpt_times)

# Vary initial positions
initial_positions = np.linspace(0.5, 1.5, 10)
avg_fpt = []
var_fpt = []

for pos in initial_positions:
    avg, var = simulate_fpt((pos, 0))
    avg_fpt.append(avg)
    var_fpt.append(var)

# Plotting results
plt.figure()
plt.errorbar(initial_positions, avg_fpt, yerr=np.sqrt(var_fpt), fmt='o')
plt.xlabel('Initial Position')
plt.ylabel('Average First Passage Time')
plt.title('Average First Passage Time vs Initial Position')
plt.show()
```

This is the output:

12 A13 Reweighing techniques

12.1 Change of Measure

I want to prove that:

$$< O >_\pi = < O\frac{\pi}{g} >_g = \frac{< O\frac{e^{-\beta E}}{g} >_g}{< \frac{e^{-\beta E}}{g} >_g} \tag{112}$$

for a canonical distribution π at inverse temperature β, where g is another probability distribution function.

The canonical distribution π is given by

$$\pi(x) = \frac{e^{-\beta E(x)}}{Z} \tag{113}$$

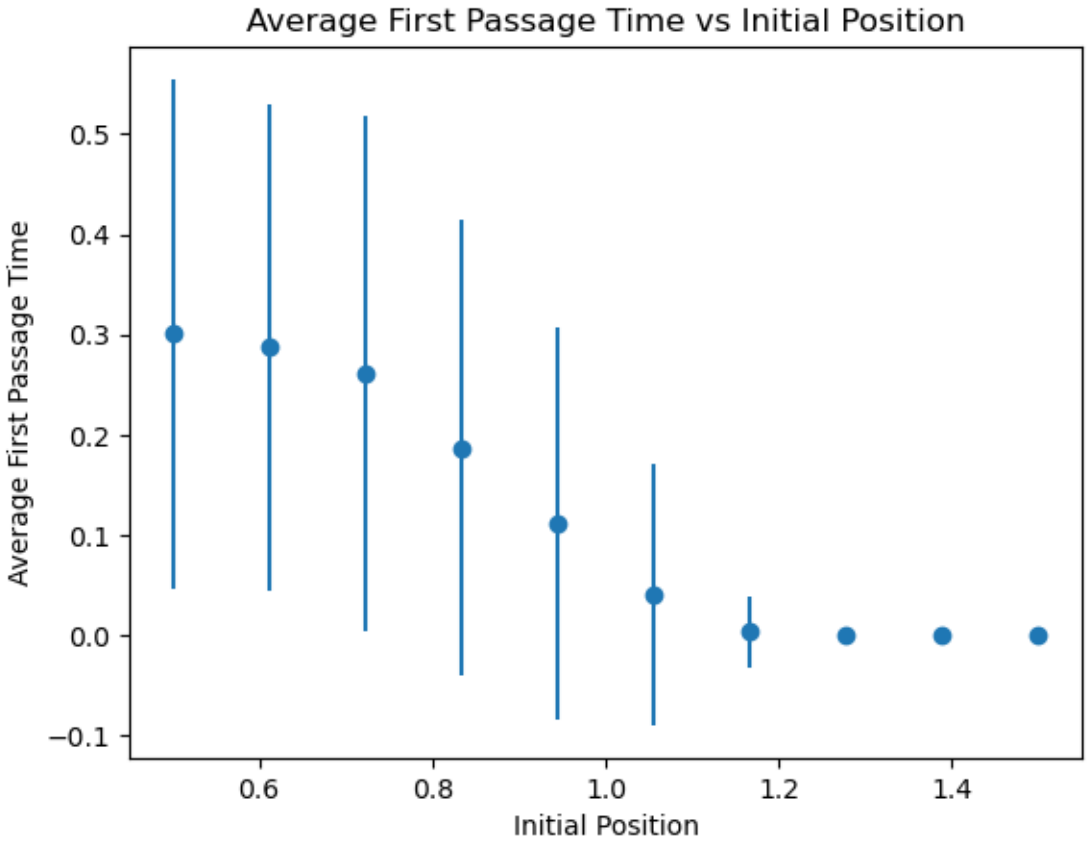


Figure 41: Swapping time as the function of initial position.

where $Z = \int e^{-\beta E(x)} d\mu(x)$. The expectation value of an observable $O(x)$ is given by

$$< O >_\pi = \int O(x)\pi(x)d\mu(x) \tag{114}$$

. Now I substitute the expression of $\pi(x)$ in $< O >_\pi$, obtaining

$$< O >_\pi = \frac{1}{Z} \int O(x)e^{-\beta E(x)} d\mu(x) \tag{115}$$

Consider now $< O\frac{\pi}{g} >_g$. This is given by:

$$< O\frac{\pi}{g} >_g = \int O(x)\frac{\pi(x)}{g(x)}g(x)d\mu(x) \tag{116}$$

$$= \int O(x)\pi(x)d\mu(x) \tag{117}$$

$$= \int O(x)\frac{\frac{e^{-\beta E(x)}}{Z}}{g(x)}g(x)d\mu(x) \tag{118}$$

$$= \int O(x)\frac{e^{-\beta E(x)}}{\int e^{-\beta E(x)}d\mu(x)}\frac{g(x)}{g(x)}d\mu(x) \tag{119}$$

$$= \frac{\int O(x)e^{-\beta E(x)}g(x)d\mu(x)}{\int e^{-\beta E(x)}g(x)d\mu(x)} \tag{120}$$

$$= \frac{< O\frac{e^{-\beta E}}{g} >_g}{< \frac{e^{-\beta E}}{g} >_g} \tag{121}$$

where $\mu(x)$ is the measure of x.

12.2 Extrapolating Average Internal Energy $U(\beta)$ Using the Single Histogram Method in the Two-Dimensional Ising Model

The two-dimensional Ising model is a classic statistical mechanics model that illustrates phase transitions and critical phenomena. In this report, we apply the Single Histogram Method (SHM) to extrapolate the average internal energy $U(\beta)$ from energy samples obtained at one specific inverse temperature β_i to a range of temperatures covering $\beta_1, \ldots, \beta_K$. The goal is to compare the extrapolated values with the average energy directly obtained from simulations at various β_k.

12.2.1 Methodology

To implement the SHM, we first load the energy data collected from simulations at a reference temperature. The following steps outline the process:

1. **Data Loading**: We read the energy data from the specified file, assuming that the second column contains the energy values.

2. **Histogram Calculation:** : A histogram of the energy distribution is constructed for the reference temperature.

3. **Partition Function**: For each target β value, we calculate the partition function $Z(\beta)$ and use it to compute $U(\beta)$ based on the histogram of energies from the reference temperature. In this case have been used 3 target temperature, in order to effort the effect of the mestimation method using different beta as reference to estimate the others.

The equations used are:

$$E(\beta) = \frac{\sum_E EN(E)e^{(\beta'-\beta)E}}{\sum_j N(E)e^{(\beta'-\beta)E}} \tag{122}$$

12.2.2 Code

```
def calculate_average_energy_single_histogram(file_path, beta_values, n_bins=250):
    # Load the energy data from the file for the reference temperature (T = 2.5)
    data_reference = np.loadtxt(file_path)
    en = data_reference[:, 1]  # Assuming the second column contains energy values

    beta=1/2.9
    # Create a histogram of energy for the reference temperature
    hist_ref, bin_edges_ref = np.histogram(en, bins=n_bins, density=True)
    bin_centers_ref = 0.5 * (bin_edges_ref[:-1] + bin_edges_ref[1:])

    U_beta = []

    # Loop over the beta values and calculate U(beta)
    for beta_target in beta_values:
        # Calculate Z(beta) for the reference temperature
        Z_ref = np.sum(hist_ref * np.exp((beta-beta_target) * bin_centers_ref))

        # Calculate P(E_j | beta) for the target temperature
        P_E_given_beta_target = (hist_ref * np.exp((beta-beta_target)
        * bin_centers_ref))

        # Calculate U(beta) for the target temperature
        U_beta_target = np.sum(bin_centers_ref * P_E_given_beta_target
        )/np.sum(hist_ref*np.exp((beta-beta_target)*bin_centers_ref ))

        U_beta.append(U_beta_target/256)

    return U_beta

# Generate beta values for temperatures from 1 to 3.2
T_values = np.linspace(1, 4, 88)
beta_values = 1.0 / T_values  # Inverse temperature

# Calculate average energies for each beta
average_energies = calculate_average_energy_single_histogram("rew_T2.9_N16.txt", beta_val

# Load energy data from energy.txt for overlay

T_energy = df_results['Temperatura']  # Assuming first column is temperature
U_energy = df_results['Energia Media'] # Assuming second column is average energy
```

12.2.3 Results

Using the above methodology, we computed the average energies $U(\beta)$ for a range of inverse temperatures. The results were plotted against the average energies obtained directly from the simulation data.

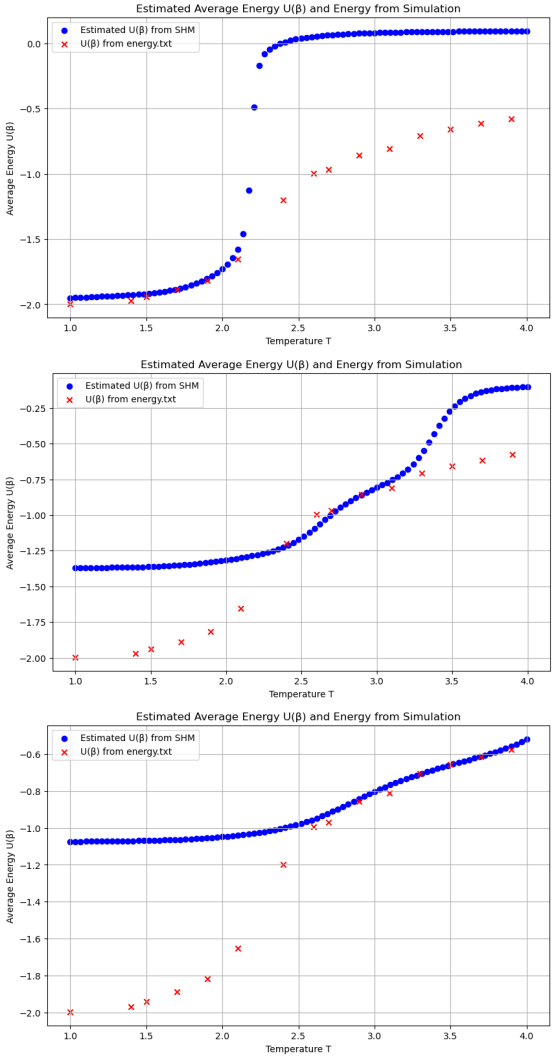


Figure 42: Extrapolation of Internal Energy via Single Histogram Method

In order to have a more accuracy of the application of the mathod, I virified that density of energy did overlap with each other(Fig.43)

The plotted results show the estimated average energy $U(\beta)$ derived from the SHM (blue points) compared to the average energy obtained from the simulation data (red crosses).

It is observed that as the target β deviates from the reference β_i, the accuracy of

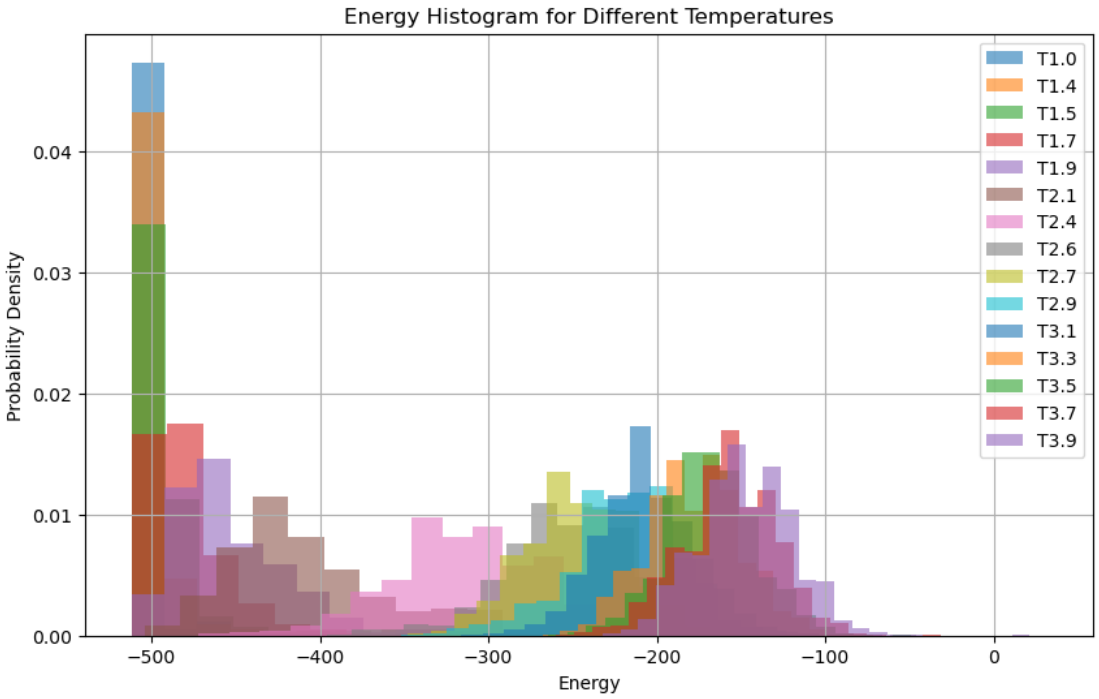


Figure 43: Sovrapposition of density of energies

the extrapolated $U(\beta)$ tends to decrease. This is expected, as larger discrepancies in temperature can lead to less overlap in the energy distributions. The prediction $U(\beta)$ varies depending on the choice of β_i; hence, careful selection of the reference temperature is crucial for accurate extrapolation. Another Important observation is pointed by the fact in this 3 different simulations I've used 3 different initial temperature to avluate the others, and staring form different values of given beta we can see that the method covers the simulations value only for specific values. In this sence, for one of them I plotted the discrepancy betwwen the estimated internal energy through SHM and the simulations value in relation of the difference between temperatures used to apply the method and other temperatures (Fig.44). The application of the Single Histogram Method provides a useful tool for estimating the average internal energy $U(\beta)$ across a range of temperatures. However, the results highlight the sensitivity of this method to the choice of the reference temperature, underscoring the importance of careful parameter selection in computational physics studies.

12.3 Multiple Histogram Method

The multiple Histogram method represents an optimized usage of the information gathered by different simulations, for calculating the density of states $\rho(E)$, and in tur, use ρ to estimate quantities at values of the parameters not used in Monte Carlo sampling. Here the formulation for the partition function, the estimation of energy and the specific

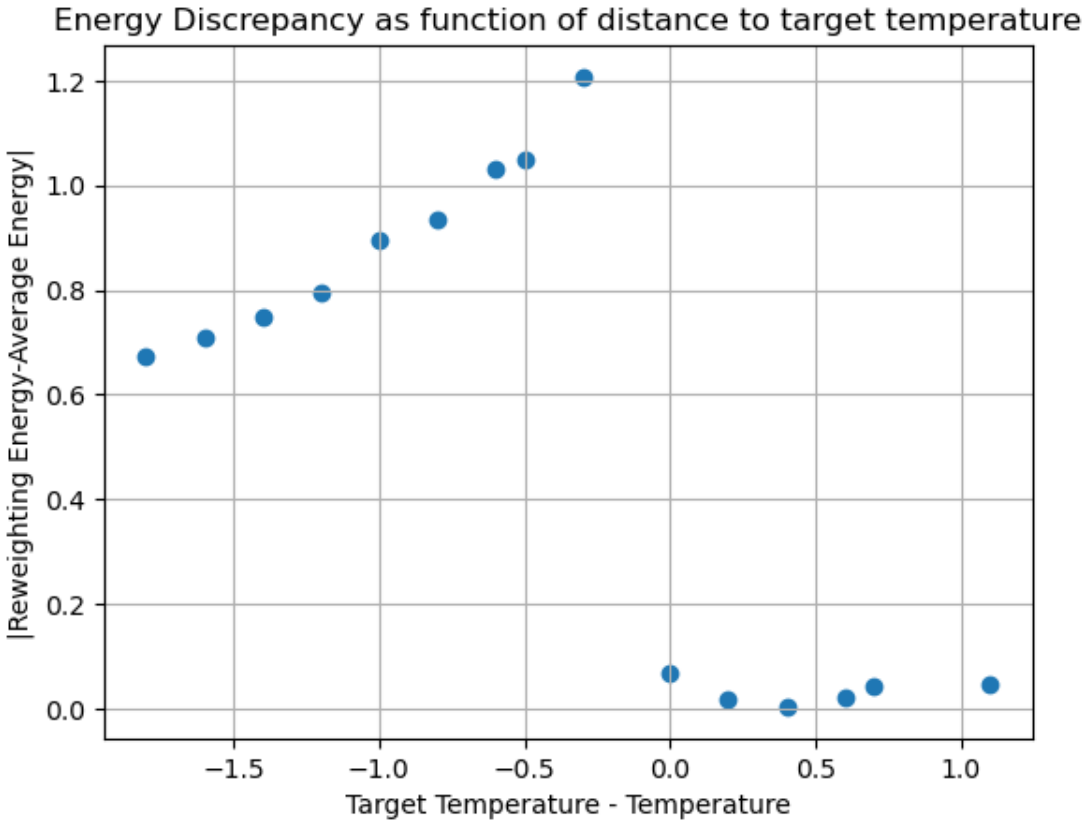


Figure 44: Discrepancy between the SHM method and simulation estimation of internal energy.

heat:

$$Z(\beta) = \sum_{i,n} \frac{1}{\sum_j M_j Z_j^{-1} e^{(\beta-\beta_j)E_{i,n}}} \tag{123}$$

$$E(\beta) = \frac{1}{Z(\beta)} \sum_{i,n} \frac{E_{i,n}}{\sum_j M_j Z_j^{-1} e^{(\beta-\beta_j)E_{i,n}}} \tag{124}$$

$$C(\beta) = \frac{1}{N\beta^2} [< E^2 >_\beta - < E >_\beta^2] \tag{125}$$

Where:

- β is the target inverse temperature at which we want to calculate the partition function.
- i and n index the configurations sampled during the simulations, where $E_{i,n}$ represents the energy of the n-th configuration in the i-th energy bin.

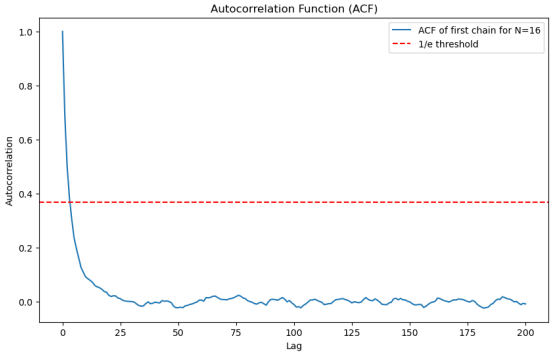


Figure 45: Autocorrelation Function

- The outer sum $\sum_{i,n}$ sums over all configurations and energy levels to calculate the total partition function at β.
- M_j is the number of measurements or configurations collected at inverse temperature β_j during the simulation.
- Z_j is the estimated partition function at inverse temperature β_j, which is computed iteratively in the Multiple Histogram Method.
- The inner sum over j runs over all the temperatures β_j at which simulations were performed. This sum accounts for the contribution of the sampled energies from different temperatures to the partition function at the target temperature β.
- The exponential factor $e^{(\beta-\beta_j)E_{i,n}}$ adjusts the Boltzmann weights of the energy levels, allowing the method to reweight the energy distribution from temperature β_j to the target temperature β.

12.3.1 Methodology

To apply the method the same passages mentioned in SHM methodology have been used. In this case an other estimation of the time correlation have been added. Then the procedure have been applied for each dimension of the lattice to observe how would it change the estimation of specific heat.

12.3.2 Code

Function to estimate partition functions fpr different temperatures

```
def compute_partition_functions(data16, T_critic, num_iterations=500,
tolerance=1e-5,verbose=True):
    # Define M vector
    M = []
    for j in range(len(data16)):
```

```
        M.append(data16[j].shape[0])

    # Define the betas
    T = np.linspace(T_critic, 1.5 * T_critic, 7)
    betas = 1 / T
    num_temperatures = len(T)

    # Initializing partition functions Z for each temperature
    Z_old = np.ones(num_temperatures)  # Start with ones to avoid division by zero
    Z_new = np.ones(num_temperatures)

    # Main loop for iterations
    for iteration in range(num_iterations):
        print(f"Iteration {iteration + 1}")
        for k in range(num_temperatures):  # Compute Z at a given temperature k
            summation = 0
            for i in range(num_temperatures):
                for n in range(data16[i].shape[0]):
                    denominator_sum = 0
                    for j in range(num_temperatures):
                        E_in = data16[i][n]
                        denominator_sum += M[j] * (1 / Z_old[j]) * np.exp((betas[k] - bet
                    summation += 1.0 / denominator_sum
            Z_new[k] = summation

        # Check for convergence
        if convergence(Z_new, Z_old) < tolerance**2:
            break

        # Update Z_old with Z_new for the next iteration
        Z_new = rescale_z(Z_new)
        Z_old = Z_new.copy()
        if verbose:
            print(f"Z_new after iteration {iteration + 1}: {Z_new}")

    # Final Z values
    Z_final = Z_new
    print(f"Final Z values: {Z_final}")
    return Z_final
```

Function to calculate partition function for target temperature

```
def compute_Z_beta(num_temperatures, M, data20, Z_k, betas, beta):
    Z_beta = 0.0
```

```
    for i in range(len(Z_k)):
        for n in range(data20[i].shape[0]):
            denominator = 0.0
            for j in range(len(Z_k)):
                denominator += M[j] * np.exp((beta - betas[j]) * data20[i][n]) / Z_k[j]
            Z_beta += 1.0 / denominator

    return Z_beta
```

Function to calculate first moment of Energy

```
def compute_E_beta(data, num_temperatures,  M, betas, Z_k, Z_beta, beta):
    E_beta = 0.0

    for i in range(len(Z_k)):
        for n in range(data[i].shape[0]):
            denominator = 0.0
            for j in range(len(Z_k)):
                denominator +=  M[j] / Z_k[j] * np.exp((beta - betas[j]) * data[i][n])
            E_beta += data[i][n] / denominator

    E_beta /= Z_beta
    return E_beta
```

Function to calculate the second moment of Energy

```
def compute_E2_beta(data_e, num_temperatures, M, betas, Z_k, Z_beta, beta):
    E2_beta = 0.0

    for i in range(len(Z_k)):
        for n in range(data_e[i].shape[0]):
            denominator = 0.0
            for j in range(len(Z_k)):
                denominator += M[j] / Z_k[j] * np.exp((beta - betas[j]) * data_e[i][n])
            E2_beta += data_e[i][n]**2 / denominator

    E2_beta /= Z_beta
    return E2_beta
```

12.3.3 Results

As we can observe we have better results than before, with a similarities between estimated values and simulated values (Fig. 46).

Regarding Fig.47 the curves display a clear peak for each system size, corresponding to the **critical temperature** T_c. The peak becomes higher and sharper for larger system sizes, as expected for a phase transition. This reflects the **divergence of specific heat**

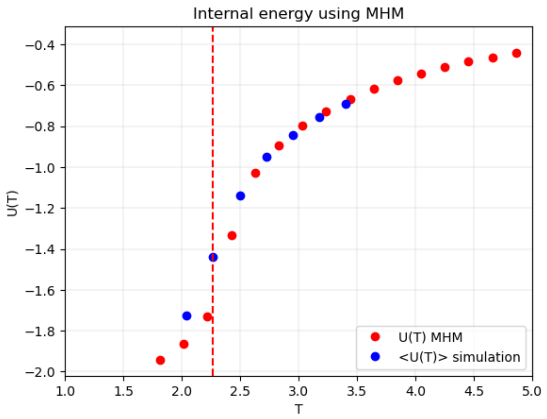


Figure 46: Interna Energy using MHM

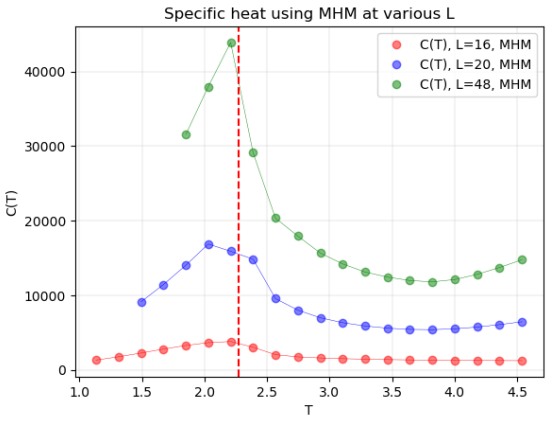


Figure 47: Specific Heat for different lattice dimension

near T_c as the system approaches the transition. The graph highlights how T_c converges to the critical value expected for large systems, with the peak becoming sharper for increasing values of L. This is typical of systems undergoing a phase transition, such as the **2D Ising model**. Beyond T_c, the specific heat decreases rapidly, a behaviour often observed in systems undergoing phase transitions. However, for $L = 48$, this trend is more noticeable, showing a sharper drop in the post-critical regime.

13 A14 Langevin Simulation of many particles

13.1 Cell list

In this document, we compare two implementations of the cell list used in the simulation: one for the "big" particle (particle 0) and another for the smaller particles. We also specify the forces acting on the particles and provide details on their initialization and

numerical integration.

13.1.1 Initialization

During the initialization phase, the particle position matrix $\mathbf{x}$ places the big particle (particle 0) at the center of the simulation box. All other smaller particles are randomly distributed within the box. The trap, which affects only particle 0, is also positioned at the center and moves linearly with velocity v_{trap}.

13.1.2 Implementation Overview

Cell List for Particle 0

The algorithm for particle 0 focuses on interactions between this "big" particle and the smaller particles within a defined cutoff distance. The force acting on particle 0, $\mathbf{F}_0$, comprises two primary components:

- **Harmonic Trap Force:** This is given by

$$\mathbf{F}_{\text{trap}} = k_{\text{trap}} \cdot (\mathbf{x}_{\text{trap}} - \mathbf{x}_0), \tag{126}$$

  where $\mathbf{x}_{\text{trap}}$ and $\mathbf{x}_0$ are the positions of the trap and particle 0, respectively.

- **Interaction Force:** The second component arises from interactions with smaller particles located in neighboring cells, denoted as Dv_0. These interaction forces are described by an exponential potential:

$$\mathbf{F}_{\text{int}} = f_{p0} \cdot e^{-r^2 \cdot \text{inv}_0}, \tag{127}$$

  where r^2 represents the distance squared between particle 0 and the periodic image of smaller particle i. The constants are defined as $f = Q \cdot \epsilon_0$ and $\text{inv} = \frac{1}{p_0 \sigma_0^2}$.

  It is important to note that the force applied to particle 0, $\mathbf{F}_0$, is subtracted from the force applied to the smaller particles, ensuring that the forces are equal in magnitude and opposite in direction.

Cell List for Smaller Particles

The smaller particles also interact among themselves, characterised by a Gaussian repulsion term:

$$\mathbf{F}_{\text{int,ss}} = f_p \cdot \exp(-r^2 \cdot \text{inv}), \tag{128}$$

where r^2 represents the distance squared between two smaller particles i and ii in neighbouring cells.

In cases where small particles are organized into polymers of length $L_p > 1$, they experience polymer bond interactions characterized by a spring-like force between adjacent particles in the polymer chain:

$$\mathbf{F}_{\text{poly}} = k_{\text{pol}} \cdot \mathbf{v}, \tag{129}$$

where $\mathbf{v}$ is the shortest distance vector between consecutive particles ii and i, considering periodic boundary conditions.

Numerical Integration

All particle positions, including the big particle, undergo a numerical integration step using the Euler method. This integration step incorporates deterministic forces and stochastic noise into each particle's position update:

$$\mathbf{x}_i(t + \Delta t) = \mathbf{x}_i(t) + \mu \cdot \mathbf{F}_i + \mathbf{dB}_i, \tag{130}$$

where:

- μ is a mobility constant,
- $\mathbf{dB}_0 = \text{stoch}_0 \cdot \mathcal{N}(0, 1)$ for particle 0,
- $\mathbf{dB}_i = \text{stoch} \cdot \mathcal{N}(0, 1)$ for small particles.

This step introduces thermal fluctuations into the dynamics of the particles.

13.1.3 Comparison of the Two Implementations

Efficiency

- **Particle 0:** The cell list for particle 0 is efficient because it only considers neighboring cells of the big particle. This localized search reduces the number of calculations, as interactions are only computed with nearby smaller particles that fall within the cutoff distance.
- **Smaller Particles:** The implementation for smaller particles is also efficient because it reduces the complexity of searching through all particles. By leveraging the cell list, each small particle only interacts with its direct neighbors, minimizing the number of pairwise interactions considered.

Memory and Data Structure

Both implementations utilize a similar data structure for the cell list, maintaining low memory overhead. However, the interaction data for smaller particles may grow larger in terms of the number of entries as more small particles are added.

Complexity

The complexity for both implementations is approximately $O(N)$ for filling the cell lists and $O(N)$ for interaction checks, but the exact number of interactions will depend on particle density and arrangements. The cell list for particle 0 is simpler in construction, as it checks a known number of interactions. In contrast, the smaller particles' implementation manages more potential neighbor checks.

Determining the "best" algorithm depends on the specific goals of the simulation:

- If the primary focus is on the interaction between a single big particle and many small particles, the cell list for particle 0 is likely sufficient.

- For simulations involving complex interactions among many small particles, the smaller particles' implementation would likely perform better due to its focused approach based on proximity.

In practical applications, testing both implementations under expected simulation conditions can provide insights into which performs better for specific cases.

13.2 Active Matter

To effectively integrate the active dumbbells into the simulation code, we will to add the related sections at specific points.

- Declare Data Structure:

```
// Global variables section
int active_dumbbells[N/2][2]; // Each row holds indices
//of particles i and i+1
double spring_constant;
double rest_length = 0.5; // Rest length of the harmonic spring
```

- Initialise Active Dumbbells

```
// Inside Init() function
    for (int i = 0; i < N/2; i++) {
         active_dumbbells[i][0] = 2 * i;     // Particle i
         active_dumbbells[i][1] = 2 * i + 1; // Particle i+1
        }
```

- Calculate Forces in $MD_step()$ Spring Force Calculation

```
// Inside MD_step() function, after initializing forces for other particles
    for (int d = 0; d < D; d++) {
        // For each dumbbell
        for (int i = 0; i < N/2; i++) {
            int p1 = active_dumbbells[i][0];
            int p2 = active_dumbbells[i][1];

            // Calculate the vector between the two particles
            double v[D];
            for (int j = 0; j < D; j++) {
                v[j] = x[p2][j] - x[p1][j];
            }
```

```
            // Calculate the distance and force
            double r = Mod(v); // Function to calculate the magnitude
            double force_magnitude = spring_constant * (r - rest_length);

            // Apply the spring force to both particles
            for (int j = 0; j < D; j++) {
                F[p1][j] -= force_magnitude * (v[j] / r);
                // Force on particle 1
                F[p2][j] += force_magnitude * (v[j] / r);
                // Force on particle 2
            }
        }
    }
```

Propulsive Force Calculation

```
    // Propulsive Force Calculation
    for (int i = 0; i < N/2; i++) {
        int p1 = active_dumbbells[i][0];
        int p2 = active_dumbbells[i][1];

        // Calculate the propulsion direction
        double v[D];
        for (int j = 0; j < D; j++) {
            v[j] = x[p2][j] - x[p1][j];
        }

        // Apply the propulsion force
        double r = Mod(v); // Get the distance
        for (int j = 0; j < D; j++) {
            F[p1][j] += propulsion_force_magnitude * (v[j] / r);
            // Propulsion on particle 1
        }
    }
```

13.3 Active Matter and Diffusion of the probe

To modify the simulation to remove the harmonic trap and study the diffusion of the probe in a bath on $N/2$ dummbells we need to gothrough the following steps:

- Remove the harmonic trap.
- Set up the simulations dor active Dumbbells as in the previous requirement.

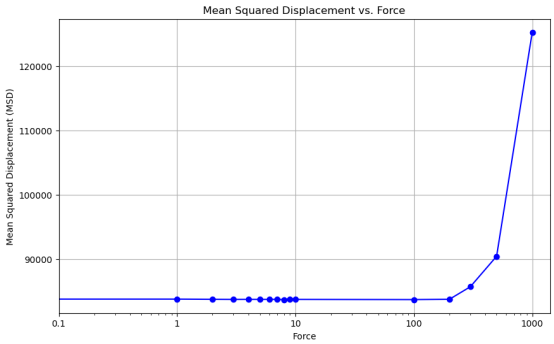


Figure 48: Mean Squared Displacement vs Force Intensity

- Update the MSD Calculation in the sampling Loop

```
double msd; // Mean square displacement
// After updating positions in the sampling loop
double dx = x[0][0] - x0_initial[0]; // x0_initial is
the initial position of the probe
double dy = x[0][1] - x0_initial[1]; // If in 2D, add this
msd += dx*dx + dy*dy; // Update MSD
msd /= num_samples; // num_samples is the number of sampling points
```

This is the result: At low forces, the probe's motion is dominated by the interaction with the dumbbells, which may restrict its movement and lead to a lower MSD. As force increases, the propulsion from the active dumbbells allows the probe to move more freely, which contributes to a higher MSD. At very high forces, the probe might experience enough propulsion to move significantly further away from its initial position, resulting in an increase in MSD.