Numerical Method in Soft Matter

Homework Assignment 1

Anna Garbo

November 17, 2023

1 A1. Sampling random points within D-dimensional domains by hit and miss

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 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;
}
```

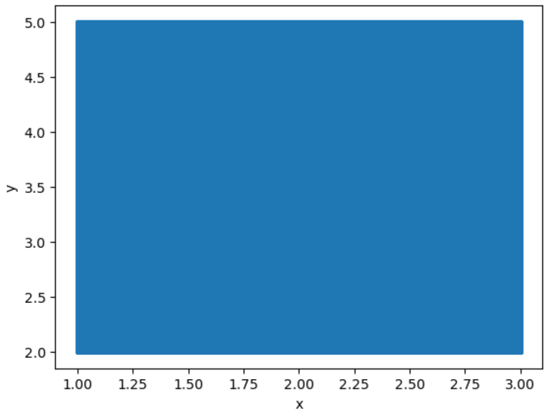


Figure 1: Rectangle of Area $A = L_{ab}L_{cd}$

1.2 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;
```

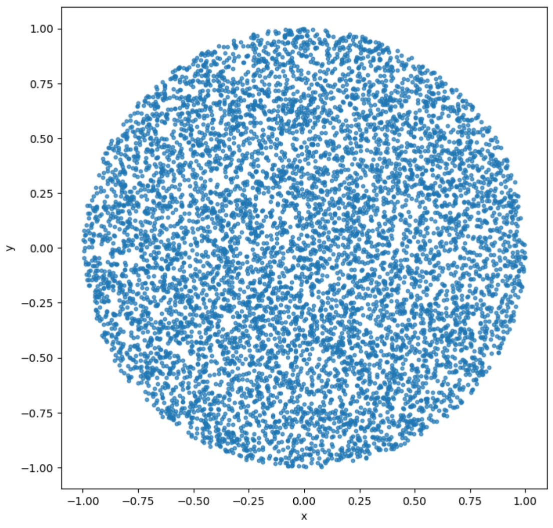


Figure 2: Random points generated in a unitary radius circle.

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

2 A.2 Sampling random numbers from a given distribution

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

2.1.1 $\rho(x) = cx^3$

- 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.

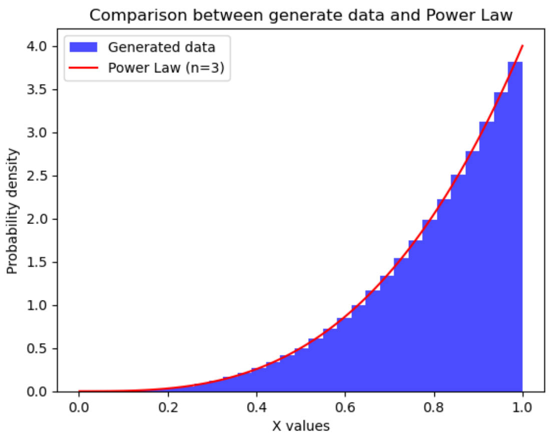


Figure 3: Inverse Sampling Method for PDF $\rho(x) = 4x^3$

2.1.2 $\rho(x) = cx^4$

- 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.

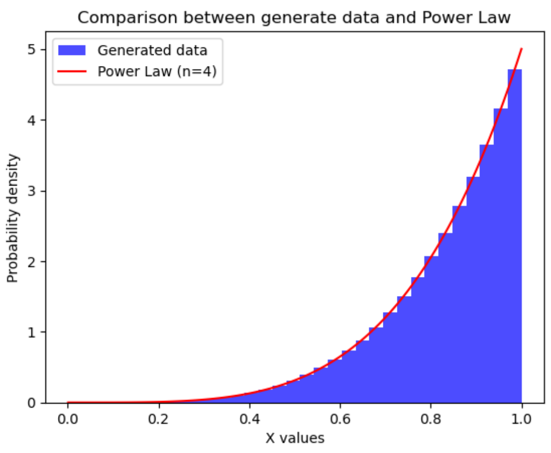


Figure 4: Inverse Sampling Method for PDF $\rho(x) = 5x^4$

2.2 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.

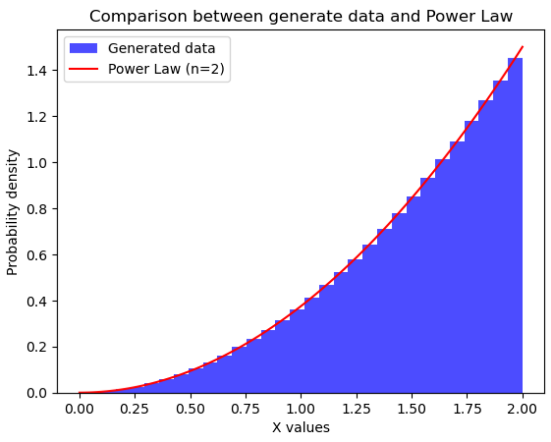


Figure 5: Inverse Sampling Method for PDF $\rho(x) = \frac{3}{8}x^2$

3 A.3 Sampling via transformation of coordinates

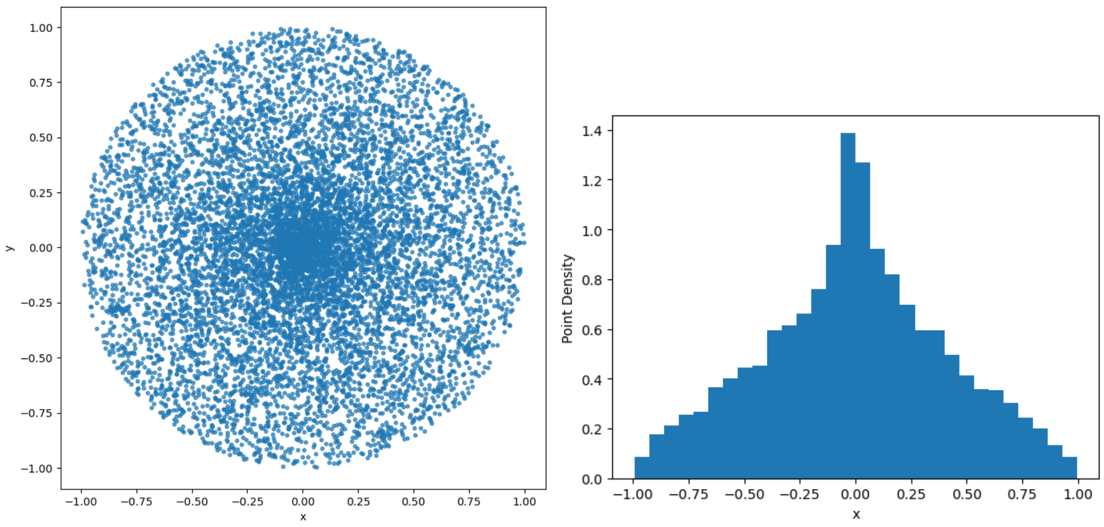
3.1 Sampling uniformly points within a unit radius disk

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

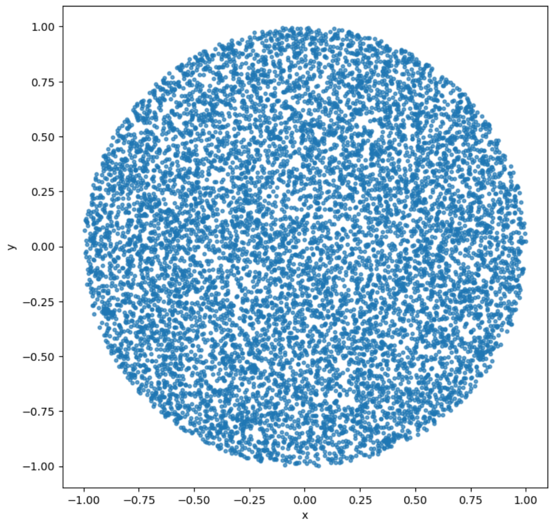
(b) Data Distribution

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

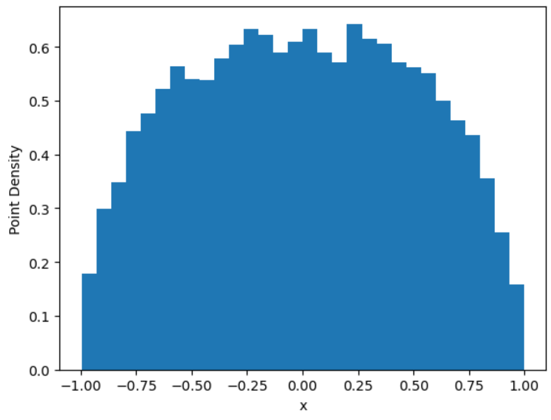
The principle reason why this algorithm does not generated data uniformly in a unit radius disk is that r gives an higher probability for points near the center of the disk instead of points more external. This fact give the hints to find a probability density that represents this variability. Indeed 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.

3.2 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;
}
```

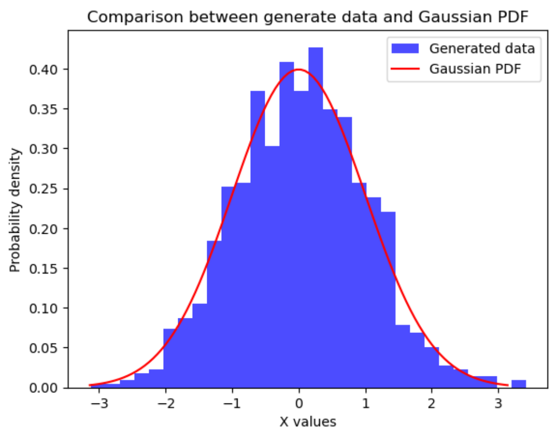


Figure 8: Data sampled from a Gaussian PDF.

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;
}
```

3.3 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:

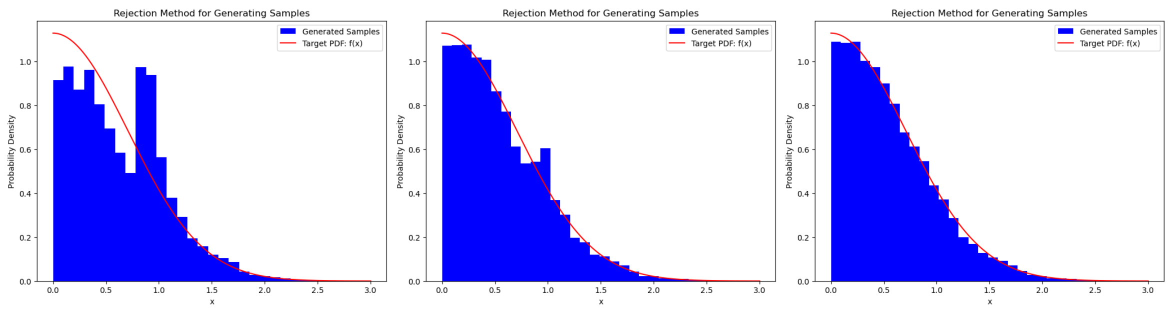
$$x(u,p) = \begin{cases} \frac{2p^2+1}{2p}u & \text{if } 0 < u \leq \frac{2p^2}{2p^2+1} \\ \sqrt{p^2 - \log(1-(2p^2+1)u + 2p^2)} & \text{if } u > \frac{2p^2}{2p^2+1} \end{cases} \tag{18}$$

- 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.11 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) .

4 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.

4.1 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.

4.2 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.

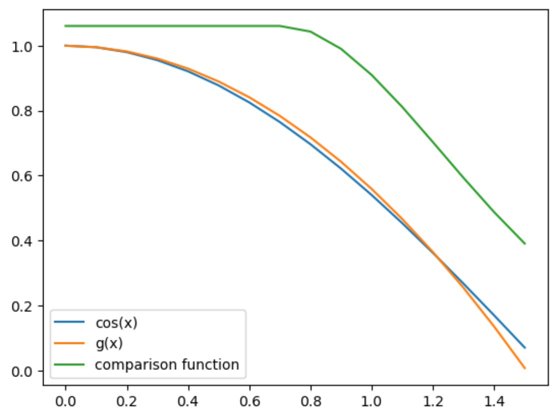


Figure 10: Comparison with three function used to build the algorithm, what is important to show is that the function from which I generate the samples are lower than the compared function.

- 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.

4.3 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

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

- 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:

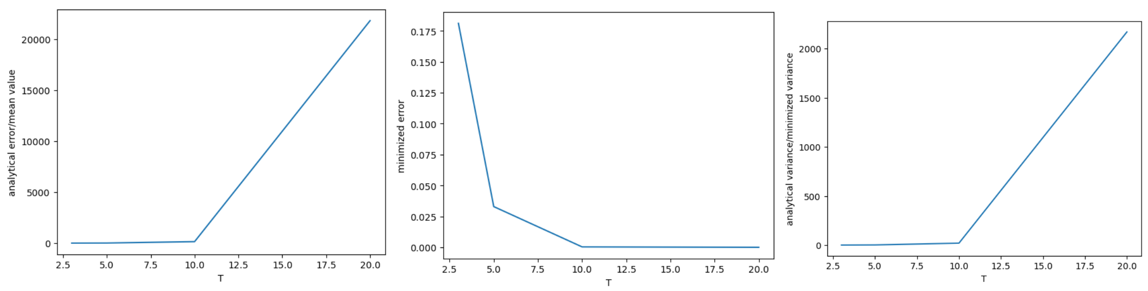
```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

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 .

5 A.5 Markov Chains

5.1 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:

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

5.1.1 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.

5.1.2 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.

5.2 Markov Chains Graphs

5.2.1 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:

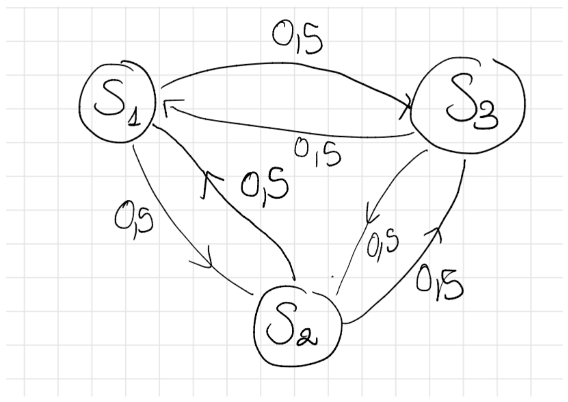


Figure 12: Graph for matrix 1

All of the three states are recurrent.

5.2.2 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:

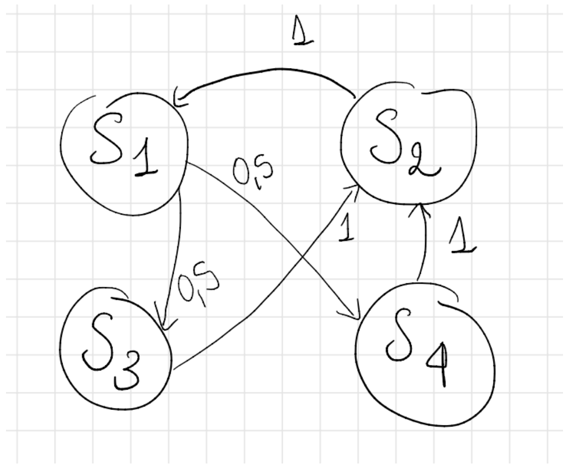


Figure 13: Graph for matrix 2

All of the four states are recurrent.

5.2.3 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:

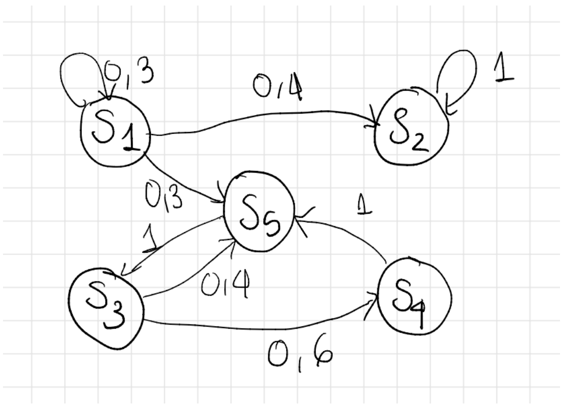


Figure 14: Graph for matrix 3

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

5.3 Irreducible Matrix

5.3.1 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]]
```

5.3.2 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]]
```

5.4 Regular Matrix

5.4.1 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.

5.4.2 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.

5.5 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.

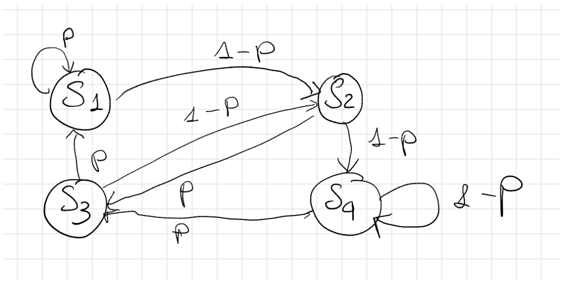


Figure 15: Graph for matrix P

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 $p = 0.2$.

5.6 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]
```