APMTH207 HW 1 Probability, Sampling & Monte Carlo Integration

Elie Attias

September 2022

Question 1: From Ensemble Interpretation to Kolmogorov Axioms

Let $P(\omega)$ be the probability of an event $\omega \subset \Omega$ where we denote the sample space as Ω. We define an ensemble as a finite collection of n models where each model is labelled with an index $i \in \{0, 1, \cdots, n-1\}$.

1. First of all, since the indicator function is positive everywhere, we must have that the probability of any ensemble member is positive: for any $A \subset \Omega$, for any $i \in \{1, \cdots, n\}$, we must have that $\chi_A(i) \geq 0$ hence the first axiom of probability holds true for ensemble interpretation.

2. It is clear that $\forall i \in \{0, 1, \cdots, n-1\}$ we have that $\chi_\Omega(i) = 1$. This gives that:

$$P(\Omega) = \sum_{i=1}^{n} \frac{\chi_\Omega(i)}{n} = \sum_{i=1}^{n} \frac{1}{n} = n \cdot \frac{1}{n} = 1$$

   Hence, the second axiom of probability must hold true for ensemble interpretation.

3. We define a family of pairwise disjoint events $(A_i)_{i \geq 1}$ in the event space Ω.

   Let $i, j \in \{1, \cdots, n\}$ such that $i \neq j$. Let $I := A_1 \cup A_2 \cup \cdots$. Define $B = A_2 \cup A_3 \cup \cdots$. Then $I = A_1 \cup B$. We note that for any s in $\{1, \cdots, n\}$, we have that $\chi_{A_i \cup A_j}(s) = \chi_{A_i}(s) + \chi_{A_j}(s) - \chi_{A_i \cap A_j}(s) = \chi_{A_i}(s) + \chi_{A_j}(s)$ since A_i and A_j are pairwise disjoint. Hence, we have that:

$$P(A_1 \cup A_2 \cup \cdots) = P(A_1 \cup B) = \sum_{i=1}^{n} \frac{\chi_{A_1 \cup B}(i)}{n} = \sum_{i=1}^{n} \frac{\chi_{A_1}(i) + \chi_B(i)}{n} = \sum_{i=1}^{n} \frac{\chi_{A_1}(i)}{n} + \sum_{i=1}^{n} \frac{\chi_B(i)}{n} = P(A_1) + P(B)$$

   Extending this reasoning to B, we have that:

$$P(A_1 \cup A_2 \cup \cdots) = P(A_1) + P(A_2) + \cdots$$

Question 2: Mutating Genome

We consider an organism with a genome in which mutations happen as a Poisson process with rate ν. Let $X(t)$ be the random variable associated to the number of mutations taking place during a time interval: $[0, t)$ where $t > 0$. For $k \in \mathbb{N}$, the probability that k mutations take place is $P(X(t) = k) = \frac{(\nu t)^k e^{-\nu t}}{k!}$.

a)

Let $t > 0$.Considering that mutations happen as a Poisson process with rate ν, we have that the probability that the genome does not obtain any new mutations within the time interval $[0, t)$ is:

$$P(X(t) = 0) = \frac{(\nu t)^0 e^{-\nu t}}{0!} = e^{-\nu t}$$

b)

Let $T \in \mathbb{N}$. The expected number of mutations during a time period T is:

$$\begin{aligned}
\mathbb{E}[X(T)] &= \sum_{k\geq 0} kP(X(T) = k) = \sum_{k\geq 0} k\frac{(\nu T)^k e^{-\nu T}}{k!} \\
&= \sum_{k\geq 1} k\frac{(\nu T)^k e^{-\nu T}}{k!} = e^{-\nu T}\sum_{k\geq 1}\frac{(\nu T)^k}{(k-1)!} \\
&= e^{-\nu T}\nu T\sum_{k\geq 1}\frac{(\nu T)^{k-1}}{(k-1)!} = e^{-\nu T}\nu T\sum_{j\geq 0}\frac{(\nu T)^j}{(j)!} \\
&= e^{-\nu T}\nu T e^{\nu T} \\
&= \nu T
\end{aligned} \tag{1}$$

c)

We now consider a population of N individuals following the Wright-Fisher model. Let us find the probability $P(t)$ of two individuals having their "first" (latest chronologically) common ancestor t generations ago.

Let A denote the probabilistic event : *selecting 2 survivors which have the same parent.* Let S denote the event : *selecting two survivors.* Also, we denote by B_i the event: *the two selected individuals have parent i.* Finally, we denote by D: *the two selected individuals have survived.*

Since one child cannot have two different parents, we have that $(B_i)_{1\leq i\leq N}$ are disjoint (i.e: for all $i, j \in \{1, \cdots, N\}$ such that $i \neq j$, we must have that $B_i \cap B_j = \emptyset$. Hence, by the law of total probability, we must have that:

$$P(A) = \sum_i^N P(S \cap B_i) \tag{2}$$

Let's focus on the term in the sum. Considering that D and $\bar{D}$ span the universe, by the law of total probability, we can decompose the previous term as:

$$\begin{aligned}
P(S \mid B_i) &= P([S \cap B_i] \cap D) + P([S \cap B_i]\bar{D}) \\
&= P([S \cap B_i] \cap D) \text{ since the probability of selecting individuals that are dead is 0.} \\
&= P([S \cap B_i] \mid D)P(D) \text{ by Bayes' theorem}
\end{aligned} \tag{3}$$

We have that :

$$\begin{aligned}
P(S \cap B_i \mid D) = P(&\text{selecting two survivors} \cap \text{the two selected individuals have parent } i \\
&\mid \text{the two selected individuals are alive})
\end{aligned}$$

Indeed, within the population of N survivors, there $\binom{N}{2}$ ways of selecting a pair of survivors, but since there is only one pair of survivors that have the same parent, we must have that : $P(S \cap B_i) = 1/\binom{N}{2}$.

Then, we have that $P(D) = P(\text{the two selected individuals have survived})$. We know that the number of ways for $N-2$ individuals within the remaining $2N-2$ individuals to be kept alive is $\binom{2N-2}{N-2}$. We note that this equals the number of ways for N individuals to be killed within the $2N-2$ remaining population : $\binom{2N-2}{N-2} = \binom{2N-2}{N}$. Then, considering that there is a total of $\binom{2N}{N}$ ways of saving and/or killing $N-2$ and N individuals respectively, we have that :

$$P(D) = \frac{\binom{2N-2}{N-2}}{\binom{2N}{N}} = \frac{\binom{2N-2}{N}}{\binom{2N}{N}}$$

Hence, we have that :

$$\begin{aligned} P(1) = P(A) &= \sum_{i}^{N} P(S \cap B_i) \\ &= \sum_{i}^{N} P([S \cap B_i] \mid D) P(D) \\ &= \sum_{i}^{N} \frac{1}{\binom{N}{2}} \frac{\binom{2N-2}{N}}{\binom{2N}{N}} \\ &= \frac{N}{\binom{N}{2}} \frac{\binom{2N-2}{N}}{\binom{2N}{N}} \\ &= \frac{2N(N-2)!}{N!} \cdot \frac{(2N-2)!}{N!(N-2)!} \cdot \frac{N!N!}{(2N)!} \\ &= \frac{1}{2N-1} \end{aligned} \tag{4}$$

Yes ! Now we know the probability of selecting two survivors which have the same parent, However, the probability of selecting two survivors that do not have the same parent is $1 - P(1) = 1 - \frac{1}{2N-1}$. Considering that the probability $P(t)$ of selecting two individuals that have their "first" common ancestor t generations ago is equivalent to not selecting 2 alive siblings for $t-1$ generations, and to select two alive siblings once, it follows that :

$$P(t) = P(1)(1-P(1))^{t-1} = \frac{1}{2N-1}\left(1 - \frac{1}{2N-1}\right)^{t-1} \xrightarrow[N\to\infty]{} 0$$

Indeed, we can check that for a fixed positive integer N, $\sum_{t=1}^{+\infty} P(t) = 1$ as:

$$\sum_{t=1}^{+\infty} P(t) = \frac{1}{2N-1}\left(1 - \frac{1}{2N-1}\right)^{t-1} = \frac{1}{2N-1} \cdot \frac{1}{1-(1-\frac{1}{2N-1})} = 1$$

d)

Now add mutations to the Wright-Fisher model. Assume we sample two individuals that followed two distinct lineages for precisely t generations (i.e., their first common ancestor occurred t generations ago). What is $P(\pi \mid t)$, the probability of π mutations arising during the t generations?

We suppose that the mutations that took place in these two family lines are independent. Let $X(t)$ be the random variable associated to the number of mutations that took place after t generations in an individual A from the first family line. Let $Y(t)$ be the random variable associated to the number of mutations that took place after t generations in an individual B from the second family line. We have that:

$$
\begin{aligned}
P(\pi \mid t) &= P(X(t) + Y(t) = \pi) \\
&= \sum_{k=1}^{\pi} P(X(t) + Y(t) = \pi \cap X(t) = k) \text{ By the law of total probability} \\
&= \sum_{k=1}^{\pi} P(X(t) + Y(t) = \pi) P(X(t) = k) \text{ by independence of X(t) and Y(t)} \\
&= \sum_{k=1}^{\pi} P(Y(t) = \pi - k) P(X(t) = k) \\
&= \sum_{k=1}^{\pi} \frac{(\nu t)^{\pi-k} e^{-\nu t}}{(\pi - k)!} \cdot \frac{(\nu t)^{k} e^{-\nu t}}{k!} \\
&= \sum_{k=1}^{\pi} \frac{(\nu t)^{\pi} e^{-2\nu t}}{(\pi - k)! k!} \\
&= \frac{e^{-2\nu t}}{\pi!} \sum_{k=1}^{\pi} \frac{\pi!}{(\pi - k)! k!} (\nu t)^{\pi} \\
&= \frac{e^{-2\nu t}}{\pi!} \sum_{k=1}^{\pi} \binom{\pi}{k} (\nu t)^{k} (\nu t)^{\pi - k} \\
&= \frac{e^{-2\nu t}}{\pi!} (\nu t + \nu t)^{\pi} \\
&= \frac{e^{-\mu t}}{\pi!} (\mu t)^{\pi} \text{ with } \mu = 2\nu
\end{aligned}
\tag{5}
$$

e)

The probability of two individuals being separated by π mutations after they were born from the same parent is $P(\pi \mid 1) = \frac{e^{-\mu}}{\pi!}(2\nu)^{\pi}$. In the previous question, we showed that $X + Y \sim \text{Poi}(\mu)$. Hence, we must have that the expected value of π is $\mathbb{E}[X(1) + Y(1)] = \mu = 2\nu$. We could also reach the same result by linearity of the expectation : $\mathbb{E}[X(1) + Y(1)] = \mathbb{E}[X(1)] + \mathbb{E}[Y(1)] = \nu + \nu = 2\nu$

Question 3: Sampling from a piecewise linear CDF

In this question, we will use the inverse sampling method to sample from the given cumulative distribution function (cdf) defined as :

$$F(x) = \begin{cases} \frac{x-x_i}{x_{i+1}-x_i}(y_{i+1}-y_i)+y_i & \text{if } x_i < x < x_{i+1} \\ 0 & \text{if } x < x_1 \\ 1 & \text{if } x > x_n \end{cases}$$

In order to find the analytical median, we iterate though the data points x_i, y_i, and find the index j such that $y_j < 0.5 < y_{j+1}$. Then we compute the theoretical median x^*_{th} by linear interpolation:

$$\frac{y_{j+1-0.5}}{y_{j+1}-y_j} = \frac{x_{j+1}-x^*}{x_{j+1}-x_j} \implies x^* = \frac{0.5-y_{j+1}}{y_{j+1}-y_j}(x_{j+1}-x_j)+x_{j+1} = 0.830091$$

Let us now generate our samples. The transformation $x = g(u)$ which is obtained from $x = F^{-1}(u)$ or equivalently $F(x) = u$ is derived by solving $F(x) = u$ with respect to x which yields:

$$x^{(i)} = \frac{(u^{(i)}-y_i)(x_{i+1}-x_i)}{y_{i+1}-y_i} + x_i$$

where $u^{(i)}$, $i, \cdots, n$ are samples drawn from the standard uniform distribution on the segment $[0, 1]$ and where (x_i, y_i) are the samples given in the cdf.csv file.

Then, in order to find the empirical median, we sorted our samples and averaged the two middle abscyssa points -since we have an even number of samples. Finally, we obtain:

$$\text{median}_{\text{theoretical}} = 0.830091 \quad > \quad \text{median}_{\text{empirical}} = 0.830966$$

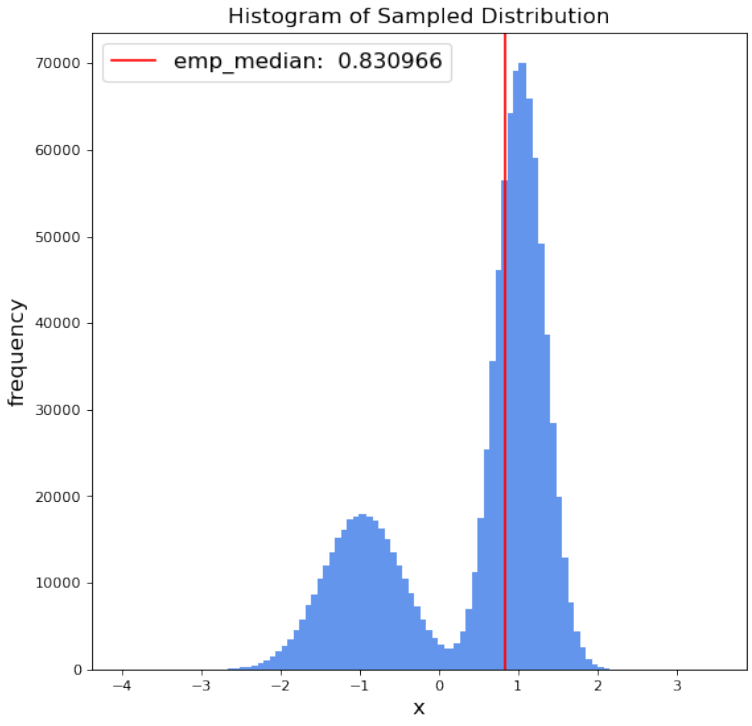


Figure 1: Histogram of Sampled Distribution

Question 4: Monte Carlo Integration

a)

We have that $V = [-1, 1]^d$.

$$
\begin{aligned}
I = \int_V f(x)dx &= \frac{1}{2^d} \int_{[-1,1]^d} \sum_{i=1}^{d} x_i^2 dx_1 \cdots dx_d \\
&= \frac{1}{2^d} \sum_{i=1}^{d} \int_{[-1,1]^{d-1}} \left[\frac{x_i^3}{3}\right]_{-1}^{1} dx_1 \cdots dx_{i-1} dx_{i+1} \cdots dx_d \text{ since integrating is a linear operation} \\
&= \frac{1}{2^d} \sum_{i=1}^{d} \int_{[-1,1]^{d-1}} \frac{2}{3} dx_1 \cdots dx_{i-1} dx_{i+1} \cdots dx_d \\
&= \frac{1}{3 \cdot 2^{d-1}} \sum_{i=1}^{d} \int_{[-1,1]^{d-1}} dx_1 \cdots dx_{i-1} dx_{i+1} \cdots dx_d \\
&= \frac{1}{3 \cdot 2^{d-1}} \sum_{i=1}^{d} \prod_{j=1}^{d-1} \int_{-1}^{1} 1 \cdot dx_j \text{ since the integral is separable } d-1 \text{ times} \\
&= \frac{1}{3 \cdot 2^{d-1}} \sum_{i=1}^{d} \prod_{j=1}^{d-1} 2 \\
&= \frac{1}{3 \cdot 2^{d-1}} \sum_{j=1}^{d} 2^{d-1} \\
&= \frac{1}{3 \cdot 2^{d-1}} d 2^{d-1} \\
&= \frac{d}{3}
\end{aligned}
\tag{6}
$$

e)

For each of the three methods, we computed 100 estimates for $M = 1, 10, 100, 1000, 10000, 100000$ and 1000000 samples for $d = 1, 2, 4, 8, 16$, and estimated our error with the standard deviation of our estimates. The error evolution for each method can be found in figure 2. We can see within each plot that whatever the value of d, the error scales as $1/\sqrt{M}$ i.e the error scales independently of d.

f)

Let us compare the running time of C++ and python for the Rejection Sample algorithm. This is shown in figure 3 and 4. These show that C++ is in average 9.6 times faster than python (on our computer) to estimate the given integral using rejection sampling. We can see that the time taken by python to perform rejection sampling scales exponentially with the dimension of integration and iteration. The command line proof of how we computed the time taken by C++ to perform rejection sampling can be found in figure 4.

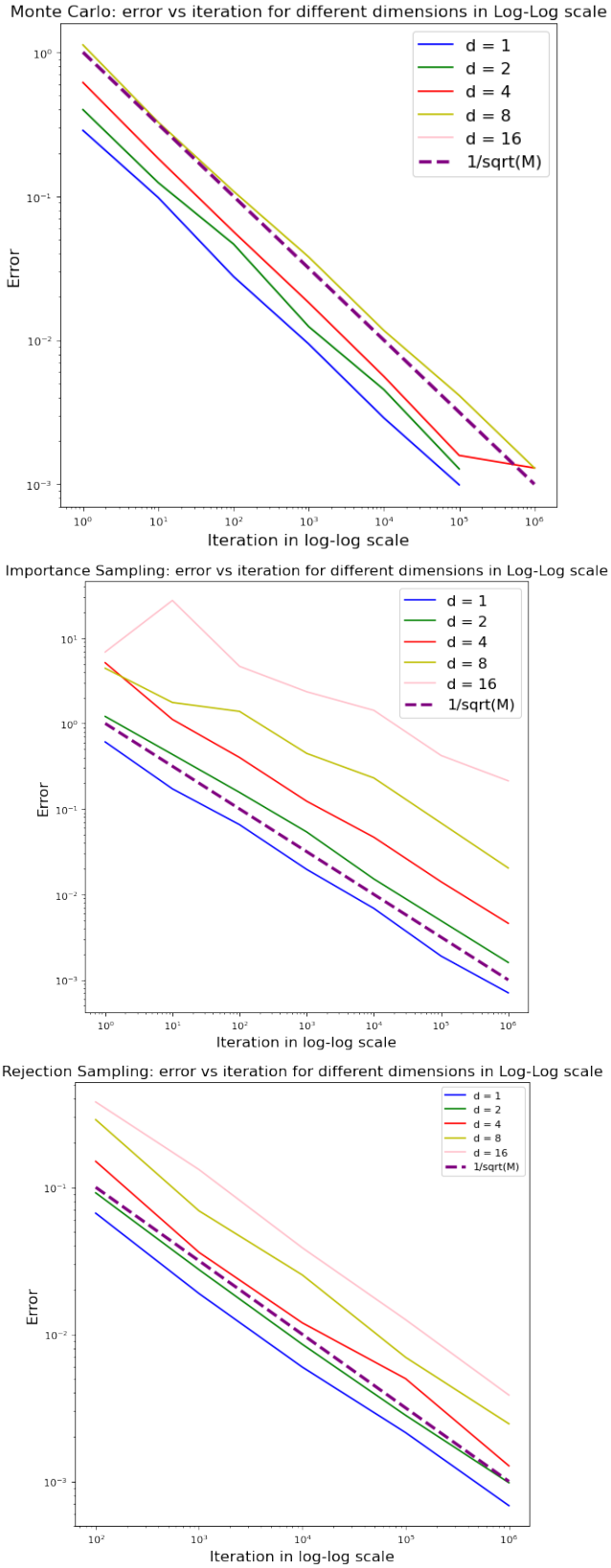


Figure 2: Error (std) of our sampling Monte Carlo (left), Importance (middle) and Rejection (right) Sampling methods as a function of the number of iterations.

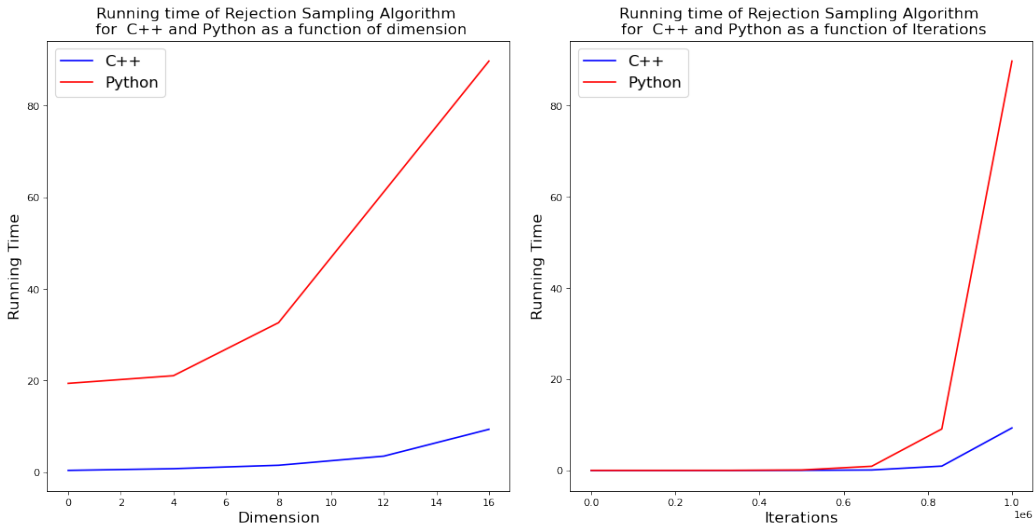


Figure 3: Running times in seconds for Rejection Sampling algorithm with respect to the dimension (with M $= 10^6$ fixed on the left) and the number of iterations (with $d = 16$ fixed on the right)

```
d : 1, M is : 1000000, estimate is : 0.332975
./out3  0.38s user 0.00s system 81% cpu 0.472 total
(base) elieattias@dhcp-10-250-156-68 AM207_cpp % g++ -std=c++14 -o out3  ex4d.cpp
(base) elieattias@dhcp-10-250-156-68 AM207_cpp % time ./out3

d : 2, M is : 1000000, estimate is : 0.667935
./out3  0.75s user 0.00s system 87% cpu 0.861 total
(base) elieattias@dhcp-10-250-156-68 AM207_cpp % g++ -std=c++14 -o out3  ex4d.cpp
(base) elieattias@dhcp-10-250-156-68 AM207_cpp % time ./out3

d : 4, M is : 1000000, estimate is : 1.33464
./out3  1.50s user 0.00s system 90% cpu 1.659 total
(base) elieattias@dhcp-10-250-156-68 AM207_cpp % g++ -std=c++14 -o out3  ex4d.cpp
(base) elieattias@dhcp-10-250-156-68 AM207_cpp % time ./out3

d : 8, M is : 1000000, estimate is : 2.66583
./out3  3.49s user 0.01s system 96% cpu 3.642 total
(base) elieattias@dhcp-10-250-156-68 AM207_cpp % g++ -std=c++14 -o out3  ex4d.cpp
(base) elieattias@dhcp-10-250-156-68 AM207_cpp % time ./out3

d : 16, M is : 1000000, estimate is : 5.33632
./out3  9.35s user 0.02s system 98% cpu 9.497 total
(base) elieattias@dhcp-10-250-156-68 AM207_cpp %
```

```
d : 16, M is : 10, estimate is : nan
./out3  0.00s user 0.00s system 0% cpu 0.154 total
(base) elieattias@dhcp-10-250-156-68 AM207_cpp % g++ -std=c++14 -o out3  ex4d.cpp
(base) elieattias@dhcp-10-250-156-68 AM207_cpp % time ./out3

d : 16, M is : 100, estimate is : 4.76978
./out3  0.00s user 0.00s system 2% cpu 0.115 total
(base) elieattias@dhcp-10-250-156-68 AM207_cpp % g++ -std=c++14 -o out3  ex4d.cpp
(base) elieattias@dhcp-10-250-156-68 AM207_cpp % time ./out3

d : 16, M is : 1000, estimate is : 5.30327
./out3  0.01s user 0.00s system 4% cpu 0.264 total
(base) elieattias@dhcp-10-250-156-68 AM207_cpp % g++ -std=c++14 -o out3  ex4d.cpp
(base) elieattias@dhcp-10-250-156-68 AM207_cpp % time ./out3

d : 16, M is : 10000, estimate is : 5.28787
./out3  0.09s user 0.00s system 42% cpu 0.226 total
(base) elieattias@dhcp-10-250-156-68 AM207_cpp % g++ -std=c++14 -o out3  ex4d.cpp
(base) elieattias@dhcp-10-250-156-68 AM207_cpp % time ./out3

d : 16, M is : 100000, estimate is : 5.32533
./out3  0.94s user 0.00s system 86% cpu 1.088 total
(base) elieattias@dhcp-10-250-156-68 AM207_cpp % g++ -std=c++14 -o out3  ex4d.cpp
(base) elieattias@dhcp-10-250-156-68 AM207_cpp % time ./out3

d : 16, M is : 1000000, estimate is : 5.33632
./out3  9.31s user 0.05s system 98% cpu 9.492 total
(base) elieattias@dhcp-10-250-156-68 AM207_cpp %
```

Figure 4: TTime taken by C++ to perform Rejection Sampling as a function of dimension evolution (left) and iteration (right)

```cpp
#include <fstream>
#include <iostream>
#include <random>
#include <sstream>
#include <string>
#include <vector>
#include <algorithm>


static void readCsv(const std::string& filename,
                    std::vector<double>& x,
                    std::vector<double>& y)
{
    std::ifstream fin(filename);

    // skip header
    {
        std::string s;
        std::getline(fin, s);
    }

    double a, b;
    char delim;

    while(fin >> a >> delim >> b)
    {
        x.push_back(a);
        y.push_back(b);
    }
}

class PieceWiseLinearCDF
{
public:
    PieceWiseLinearCDF(std::vector<double> x, std::vector<double> y)
        : x_(std::move(x))
        , y_(std::move(y))
    {}

    PieceWiseLinearCDF(const std::string& filename)
    {
        readCsv(filename, x_, y_);

    }

    double generateSample(std::mt19937& gen, int c) const
    {
        std::uniform_real_distribution<double> distrib(0, 1);
        double u = distrib(gen);
        int i = 0;
        while (y_[i]<u){
            i += 1;
        }
        int j=0;
        while (0.5 > y_[j+1]){
            j++;
        }
        double the_median;

        if (c ==0){
            //the_median =(x_[j] +x_[j + 1] )/2;
            //the_median =x_[j];
            the_median = (0.5 -y_[j + 1])*(x_[j+1] - x_[j])/(y_[j+1] - y_[j]) + x_[j+1];
            std :: cout << "the theoretical median is "<< the_median << '\n';
            c+=1;
        }
        return (u - y_[i])*(x_[i + 1] - x_[i])/(y_[i + 1] - y_[i])+ x_[i];
    }

    double getMinX() const {return x_.front();}
    double getMaxX() const {return x_.back();}

private:
    std::vector<double> x_;
    std::vector<double> y_;
};


int main(int argc, char **argv)
{
    // Load the data
    PieceWiseLinearCDF cdf("cdf.csv");

    // histogram quantities
    const int nbins = 100;

    std::vector<double> histogramLoc(nbins); // center of each bin
    std::vector<int> histogramCounts(nbins); // number of samples per bin

    const double a = cdf.getMinX(); // lowest bound of the histogram
    const double b = cdf.getMaxX(); // highest bound of the histogram
    const double h = (b-a) / nbins; // bin size

    for (int i = 0; i < nbins; ++i){
        histogramLoc[i] = a + (i+0.5) * h;
    }

    // collect samples
    const int nsamples = 1'000'000;
    std::vector<double> samples(nsamples);
    std::mt19937 gen(3456789);
    int c = 0;
    for (auto& x : samples)
    {
        x = cdf.generateSample(gen, c);
        int i=0;
        while (histogramLoc[i] < x){
            i++;
        }
        histogramCounts[i] += 1;
        c = 1;
        // TODO fill in the histogram counts
    }

    // TODO compute and printout the empirical median and the analytical median
    double mid = 0.5;
    std::sort(std::begin(samples), std::end(samples));
    double emp_median = (samples[499999] + samples[500000])/2;

    std :: cout << "the empirical median is "<<emp_median;


    // write the histogram to a csv file

    std :: ofstream cout("histogram.csv");
    std :: cout << "x,count" << std::endl;

    for (int i = 0; i < nbins; ++i){
        std :: cout << histogramLoc[i] << ',' << histogramCounts[i]<< '\n';
    }
    return 0;
}
```

```cpp
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <vector>
#include <random>
#include <fstream>
#include <sstream>
#include <string>
#include <algorithm>
#include <format>

double F(double x, int d);
double monteCarloEstimate(int M, int d, std :: mt19937 gen);
double Importance_Sampling(int M, int d,std ::  mt19937 gen);
double Rejection_Sampling( int M, int d, std :: mt19937 gen);
double stad(std::vector<double> x);
double mean(std::vector<double> x);

int main()
{
    int itera[] = {1, 10, 100, 1'000, 10'000, 100'000, 1'000'000};
    float dims[] = {1, 2, 4, 8, 16};

    int M_max = 7;
    int d_max = 5;
    int nrows = M_max; //M
    int ncols = d_max;//d

    double err_MC[ncols][nrows];
    double err_IS[ncols][nrows];
    double err_RS[ncols][nrows];

    std :: vector <double> estimates_MC (100);
    std :: vector <double> estimates_IS (100);
    std :: vector <double> estimates_RS (100);

    for (int d = 3; d < d_max ; d++){
        for (int M = 0; M < M_max ; M++){
            std:: cout << "d: "<< dims[d] << " M is : "<<itera[M]<< "\n";
            std:: cout << "\n";

            for (int i= 0; i < 100; i++){
                int seed = rand();
                std :: mt19937 gen(seed);
                estimates_MC[i] = monteCarloEstimate(itera[M], dims[d], gen);
                estimates_IS[i] = Importance_Sampling(itera[M], dims[d], gen);
                estimates_RS[i] =  Rejection_Sampling(itera[M], dims[d], gen);

            }
            err_MC[M][d] = stad(estimates_MC);
            err_IS[M][d] = stad(estimates_IS);
            err_RS[M][d] = stad(estimates_RS);
        }
    }
    std:: cout << "Matrix of errors for MC is : "<<  "\n";
    for (int i = 0; i < nrows; i++)
        for (int j = 0; j < ncols; j++)
            std :: cout << err_MC[i][j] << ",\n"[j == ncols-1];

    std :: cout << '\n';
    std :: cout << '\n';

    std:: cout << "Matrix of errors for IS is : "<< "\n";
    for (int i = 0; i < nrows; i++)
        for (int j = 0; j < ncols; j++)
            std :: cout << err_IS[i][j] << ",\n"[j == ncols-1];

    std :: cout << '\n';
    std :: cout << '\n';

    std:: cout << "Matrix of errors for RS is : "<< "\n";
    for (int i = 0; i < nrows; i++)
        for (int j = 0; j < ncols; j++)
            std :: cout << err_RS[i][j] << ",\n"[j == ncols-1];
        return 0;
}


double mean(std::vector<double> x){
    double sum;
    for (int i = 0; i< x.size(); i++){
        sum += x[i];
    }
    return sum/x.size();
}

double stad(std::vector<double> x){
    double average ;
    average = mean(x);
    double sum;
    for (int i = 0; i< x.size(); i++){
        sum += pow(fabs(average - x[i]), 2)/x.size();
    }
    return sqrt(sum);
}

double F(std :: vector <double> x , int d)
{
    double sum = 0;
    for (int i = 0; i < d ; i++){
        sum += pow(x[i], 2);
    }
    return pow(2, -d)*sum;
}

float g(std :: vector <double> x, int d )
{
    double s = 1;
    for (int i = 0; i < d ; i++){
        if (x[i] < -1 ){
            return 0;
        }
        else if (x[i] >1  ){
            return 0;
        }
    }
    s = F(x, d);
    return s;
};

float normal_pdf(float x, float s)
{
    static const float inv_sqrt_2pi = 0.3989422804014327;
    float a = x/ s;
    return inv_sqrt_2pi / s * std::exp(-0.5f * a * a);
};

float multiply(std :: vector <double> x, int d )
{
    double multi = 1;
    for (int i = 0; i < d ; i++){
        multi *= normal_pdf(x[i], 0.5);
    }
    return multi;
};


double monteCarloEstimate(int M, int d, std :: mt19937 gen)
//Function to execute Monte Carlo integration on predefined function
{

        double totalSum = 0;
        double randNum, functionVal;
        int iter = 0;
        std :: vector <double> x_ (d);
    std :: uniform_real_distribution< double> distrib(-1,1);
        while (iter<M)
        {
                // generate random vector
                for (int i = 0; i < d ; i++){
            x_[i] = distrib(gen);
                }
                totalSum += F(x_, d);
                iter++;
        }

    double estimate = totalSum * pow(2, d)/(M);
        return estimate;
}

double Importance_Sampling(int M, int d,std ::  mt19937 gen)
//Function to execute Monte Carlo integration on predefined function
{
    double totalSum = 0;
    double randNum, functionVal;
    int iter = 0;
    std :: vector <double> x_ (d);
    std :: normal_distribution<double> distrib(0.,0.5);
    while (iter<M)
    {
        // generate random vector
        for (int i = 0; i < d ; i++){
            x_[i] = distrib(gen);
        }
        totalSum += g(x_, d) / multiply (x_, d);
        iter++;
    }

    double estimate = totalSum / M;
    return estimate;
}

double Rejection_Sampling(int M, int d, std :: mt19937 gen){
    std::random_device rd;
    std::uniform_real_distribution<> dis_x(-1, 1);
    double estimate;

    std::uniform_real_distribution<> dis_y(0, 1);
    double gamma = 1.5;
    double SUM1 =0. ; double SUM2 = 0.;
    int c = 0;


    for(int m = 0; m < M; ++m){
        std::vector<double> x(d);
        while (true){
            for(int i = 0; i < d; ++i) {
                x[i] = dis_x(gen);
            }
            double y; double q; double w;
            y = dis_y(gen);
            if (y < gamma * (0.1 + F(x, d))){
                q = 0.1 + F(x, d);
                w = 1/q;
                SUM1 += w * g(x, d);
                SUM2 += w;
                c += 1;
                break;
            }
        }
    }
    estimate = pow(2,d) * SUM1/SUM2;
    return estimate;
}
```

In [1]:
```
import pandas as pd
import numpy as np
import os
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure
import time
```

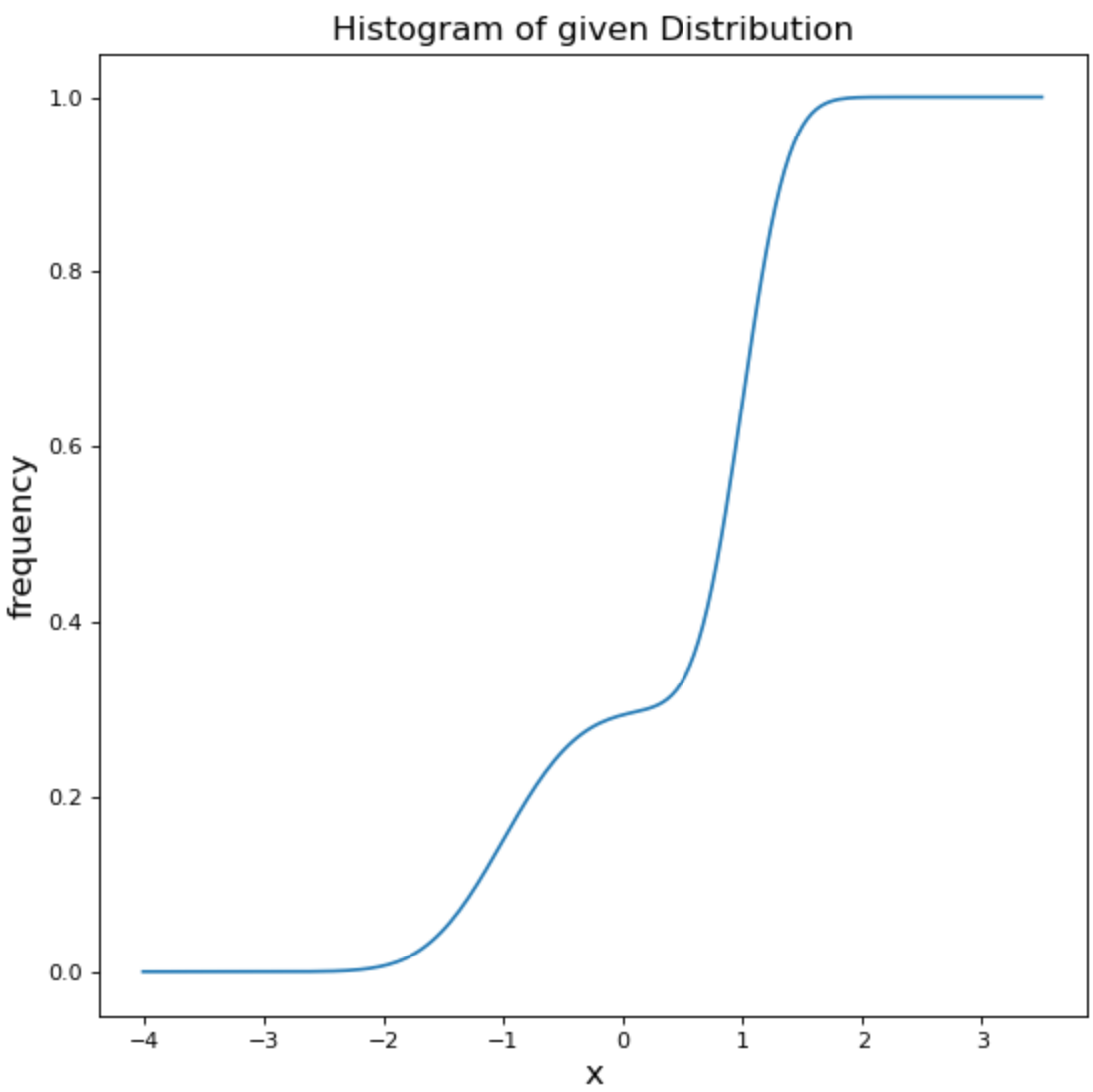
AM207 Homework 1

Elie Attias

In [2]:
```
#os.getcwd()
file = "./cdf.csv"
df = pd.read_csv(file)
df.head()
x = df[["x"]].values
y = df[["cdf"]].values
```

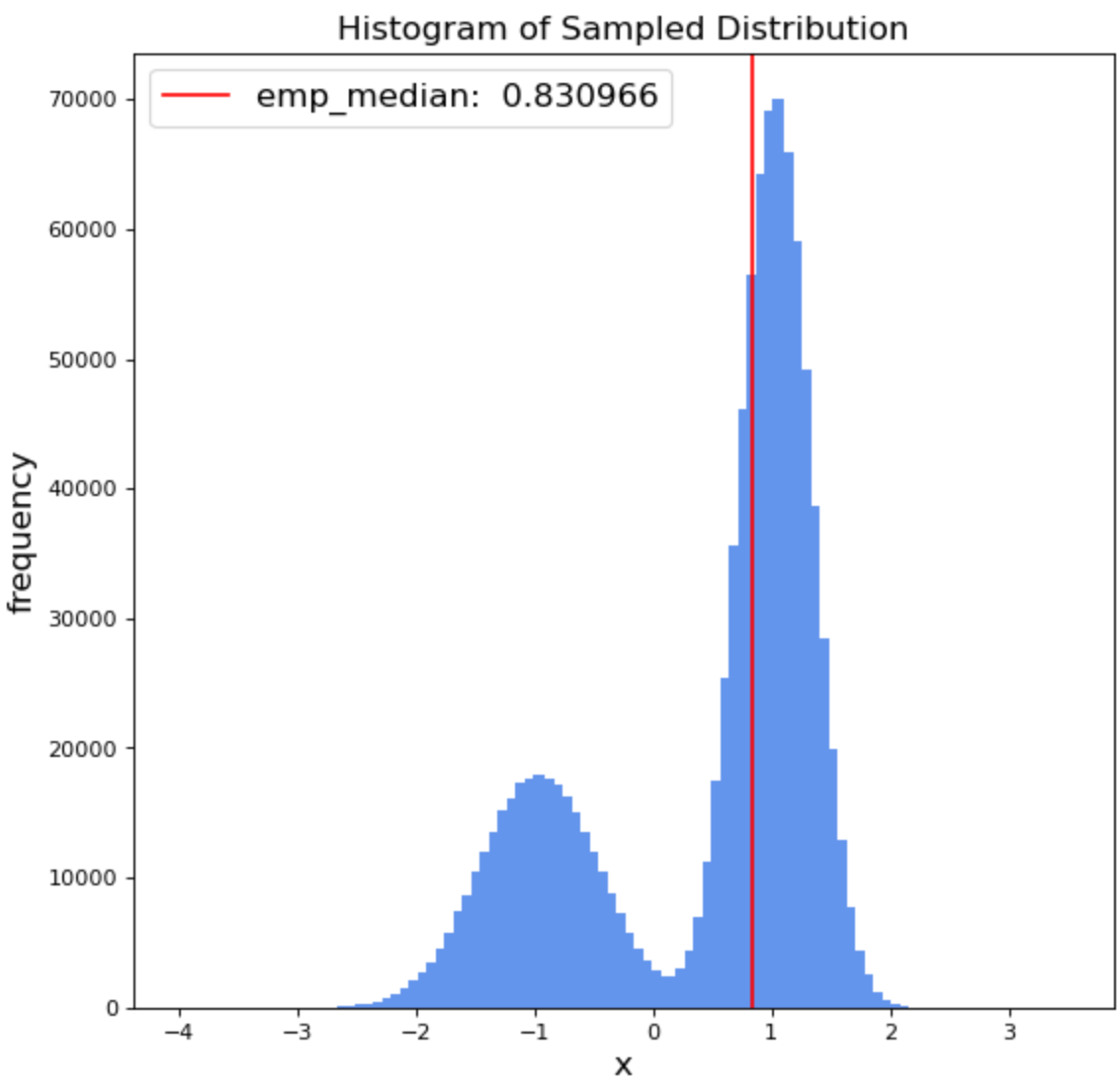
In [3]:
```
median = 0.5
for i in range(len(y)):
    if y[i]>0.5:
        break
the_median = x[i]
figure(figsize=(8, 8), dpi=80)
plt.plot(x, y)
plt.title('Histogram of given Distribution', fontsize = 15)
plt.xlabel('x', fontsize = 15)
plt.ylabel('frequency', fontsize = 15)
```

Out[3]:
```
Text(0, 0.5, 'frequency')
```



In [4]:
```
file = "./histogram_Elie.csv"
data = pd.read_csv(file)
data.head()
x = data["X"]
y = data["Y"]

figure(figsize=(8, 8), dpi=80)
plt.bar(x, y, width = 0.1, color = 'cornflowerblue')
emp_median =  0.830966
plt.axvline(x = emp_median, color = 'r', label = 'emp_median:  0.830966')
plt.title('Histogram of Sampled Distribution', fontsize = 15)
plt.xlabel('x', fontsize = 15)
plt.ylabel('frequency', fontsize = 15)
plt.legend(loc = 'upper left', fontsize = 15)
```

Out[4]:
```
<matplotlib.legend.Legend at 0x7febb00cc518>
```



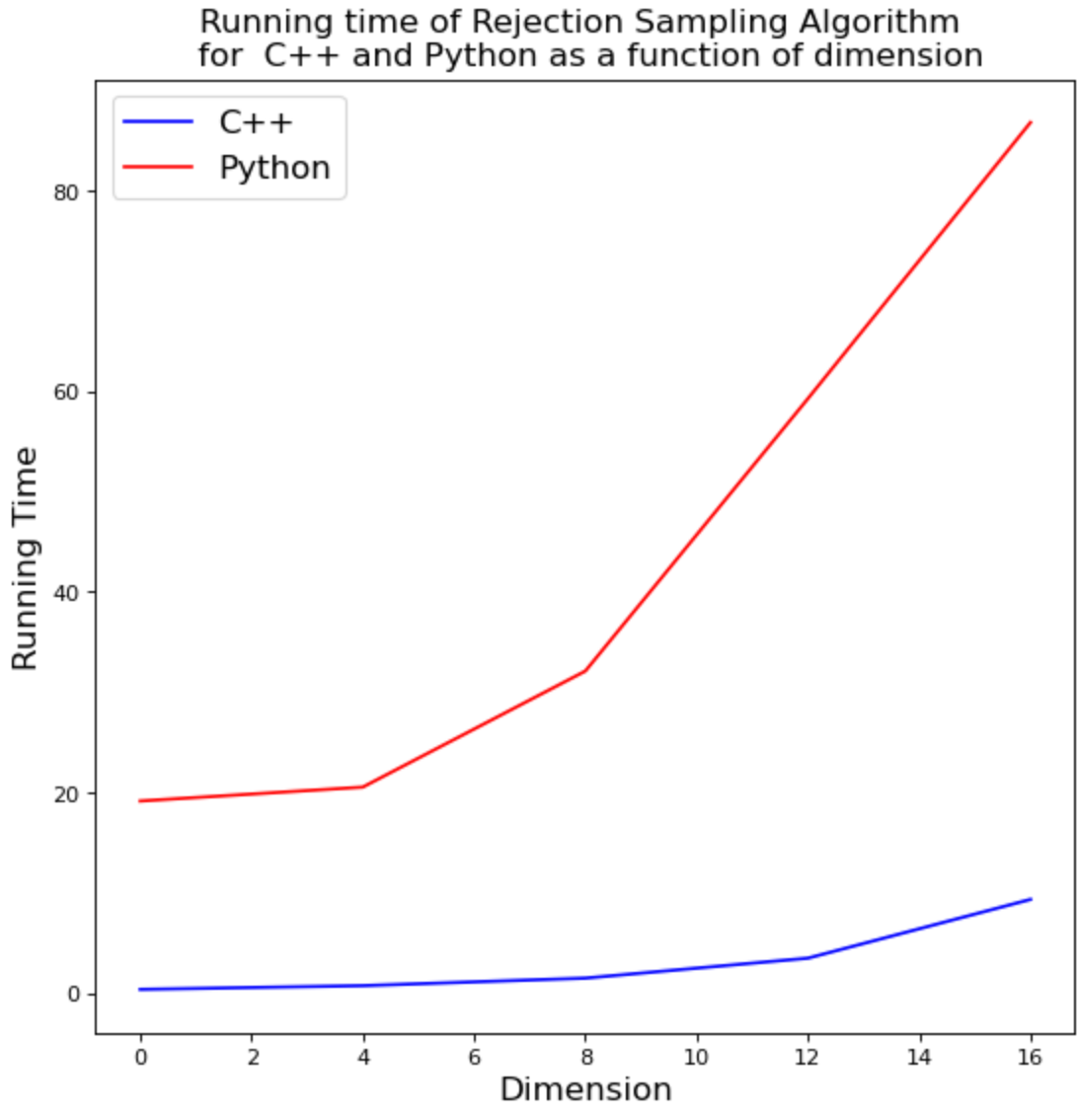
In [5]:
```
def g(x, d):
    for i in x:
        if abs(i) > 1:
            return 0
    return F(x, d)

def F(x, d):
    return sum([x_i**2 for x_i in x])/(2**(d))

np.random.seed(1)
```

In [6]:
```
def Rejection_Sampling(d, M):
    start = time.time()
    total_sum1 = 0
    total_sum2 = 0
    q = 0
    m = 0
    crit = 0
    gamma = 1.5
    for i in range(int(M)):
        q = 0
        while crit ==0:
            x = np.random.uniform(low= -1., high=1.0, size = d)
            y = np.random.uniform(low= 0., high=1.0)
            if y < gamma*(0.1 + F(x, d)):
                q = 0.1 + F(x, d)
                w = 1/q
                total_sum1 += w * g(x, d)
                total_sum2 += w
```

```
            break

    estimate = pow(2, d) * total_sum1 / total_sum2
    end = time.time()
    return estimate, end - start
```

In [7]:
```
Dimensions = np.linspace(0, 16, 5)
Iter = np.linspace(1, 1e6, 7)
M = 1000000

## varying dimension
python_dims = [Rejection_Sampling(a, M)[1] for a in [1,2,4,8,16]]
python_M = [Rejection_Sampling(16, M)[1] for M in [2, 10, 1e2, 1e3, 1e4, 1e5, 1e6]]

#evaluated by hand from terminal : screenshots of command line present in the latex
cpp_dim = [0.38, 0.75, 1.5, 3.49, 9.35]
cpp_iter = [0,0,0, 0.01,0.09, 0.94, 9.31]


figure(figsize=(8, 8), dpi=80)
plt.plot(Dimensions, cpp_dim, c = 'b', label = 'C++')
plt.plot(Dimensions, python_dims, c = 'r', label = 'Python')
plt.legend(loc = "upper left", fontsize=15)
plt.xlabel('Dimension', fontsize=15)
plt.ylabel('Running Time', fontsize=15)
plt.title('Running time of Rejection Sampling Algorithm \n for  C++ and Python as a func
```

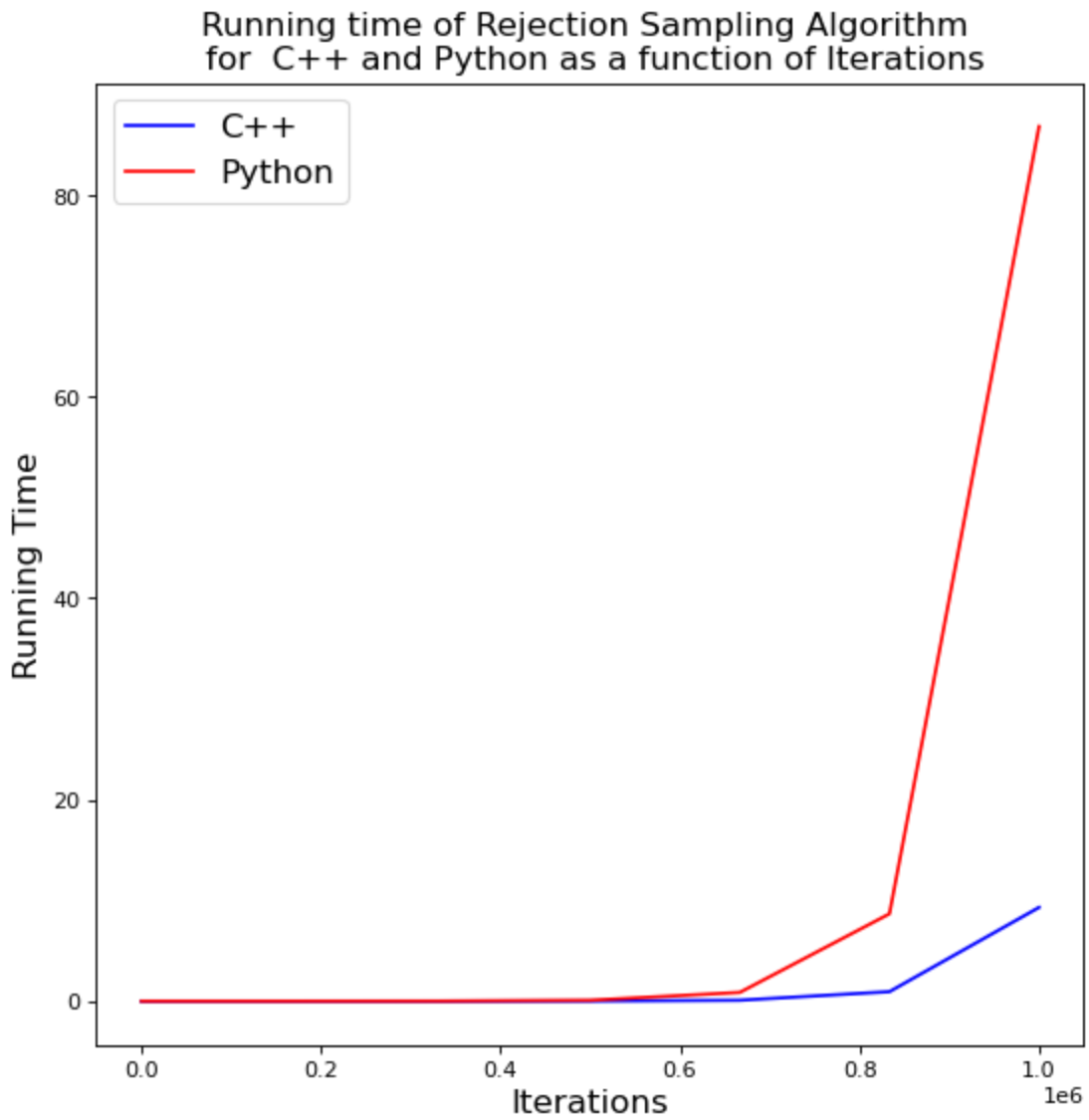
Out[7]:
```
Text(0.5, 1.0, 'Running time of Rejection Sampling Algorithm \n for  C++ and Python as a
function of dimension')
```



In [8]:
```
### compute average relative speed factor difference
a = 0
c = 0
for i in range(7):
    if cpp_iter[i] != 0:
        c += 1
        a += (python_M[i]/cpp_iter[i])
print(f"Python is in average {a/c} times slower than c++ based on our observations")
```

```
Python is in average 9.23361199875685 times slower than c++ based on our observations
```

In [9]:
```
figure(figsize=(8, 8), dpi=80)
plt.plot(Iter, cpp_iter, c = 'b', label = 'C++')
plt.plot(Iter, python_M, c = 'r', label = 'Python')
plt.legend(loc = "upper left", fontsize=15)
plt.xlabel('Iterations', fontsize=15)
plt.ylabel('Running Time', fontsize=15)
plt.title('Running time of Rejection Sampling Algorithm \n for  C++ and Python as a func
plt.savefig('cpp_vs_python.png')
plt.show()
```

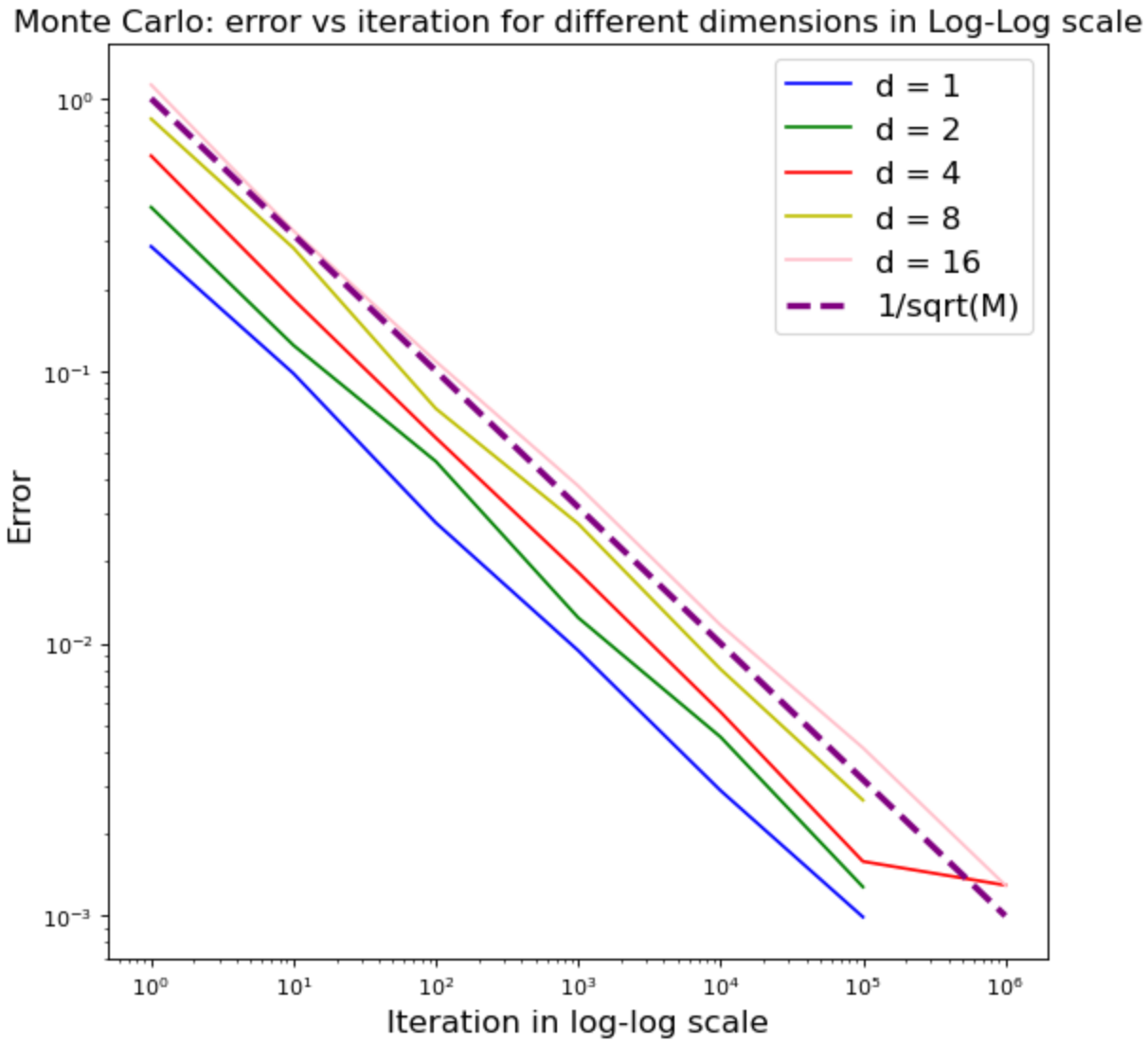


In [10]:
```
df3 = pd.read_csv("./MC_err_matrix.csv")
df4 = pd.read_csv("./IS_error_matrix.csv")
df5 = pd.read_csv("./RS_error_matrix.csv")
Iterations = np.array([1, 10, 100, 1000, 10000, 100000, 1000000])
sqrt_M = [np.sqrt(1/i) for i in Iterations]
```

In [11]:
```
## Monte Carlo Plot
figure(figsize=(8, 8), dpi=80)
```

```
MC_d1 = df3['1'].values
MC_d2 = df3['2'].values
MC_d4 = df3['4'].values
MC_d8 = df3['8'].values
MC_d16 = df3['16'].values

plt.plot(Iterations[:-1], MC_d1[:-1], c = 'b', label = 'd = 1')
plt.plot(Iterations[:-1], MC_d2[:-1], c = 'g', label = 'd = 2')
plt.plot(Iterations, MC_d4, c = 'r', label = 'd = 4')
plt.plot(Iterations[:-1], MC_d8[:-1], c = 'y', label = 'd = 8')
plt.plot(Iterations, MC_d16, c = 'pink', label = 'd = 16')

plt.plot(Iterations, sqrt_M,  '--', c = 'purple', label = "1/sqrt(M)", linewidth=3.0)
plt.xscale('log')
plt.yscale('log')

plt.xlabel("Iteration in log-log scale", fontsize=15)
plt.ylabel("Error", fontsize=15)
plt.legend(loc = 'upper right', fontsize=15)
plt.title("Monte Carlo: error vs iteration for different dimensions in Log-Log scale", f
```

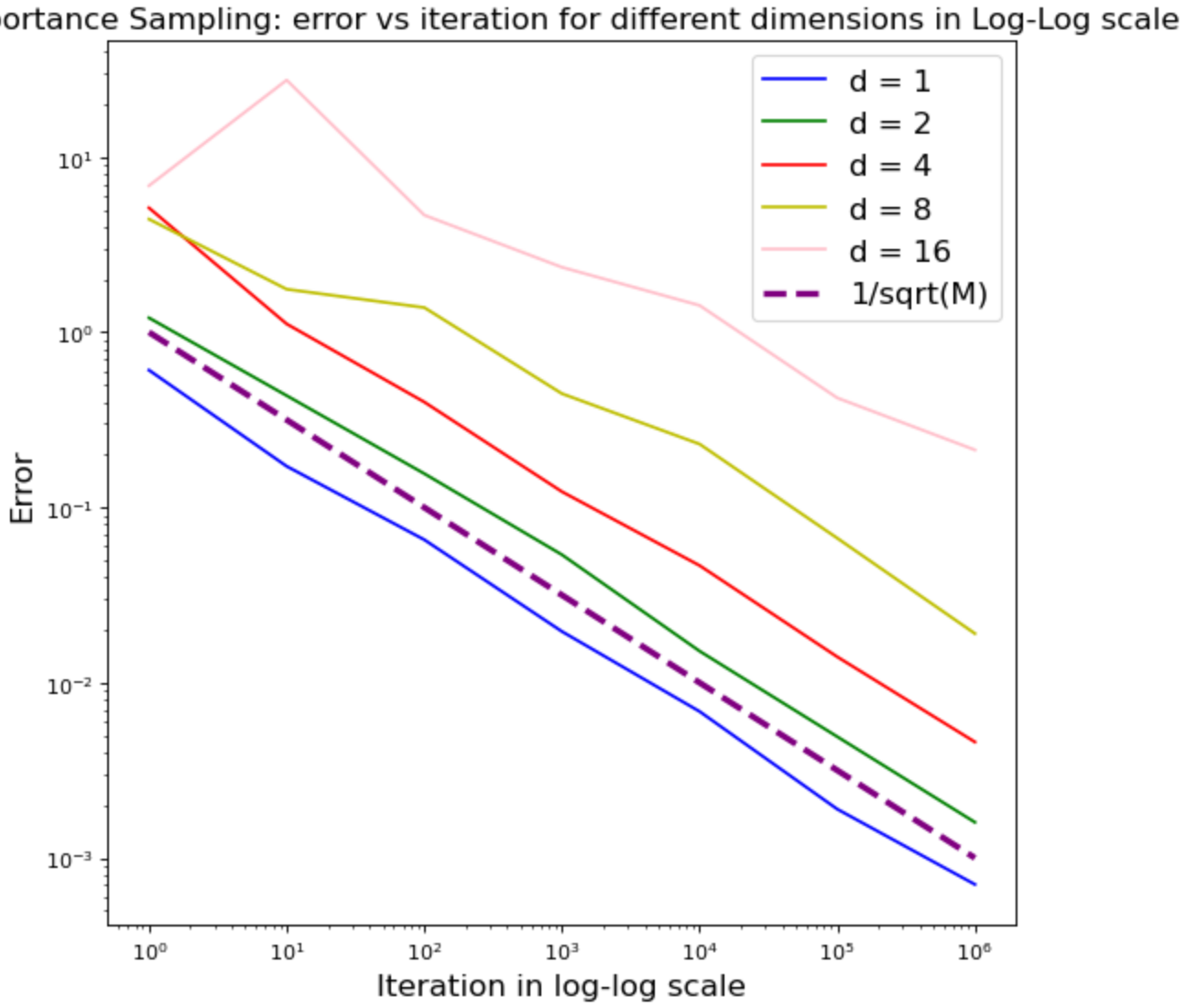
Out[11]:
```
Text(0.5, 1.0, 'Monte Carlo: error vs iteration for different dimensions in Log-Log scal
e')
```



In [12]:
```
## Importance Sampling Plot
figure(figsize=(8, 8), dpi=80)

IS_d1 = df4['1'].values
IS_d2 = df4['2'].values
IS_d4 = df4['4'].values
IS_d8 = df4['8'].values
```

```
IS_d16 = df4['16'].values

plt.plot(Iterations, IS_d1, c = 'b', label = 'd = 1')
plt.plot(Iterations, IS_d2, c = 'g', label = 'd = 2')
plt.plot(Iterations, IS_d4, c = 'r', label = 'd = 4')
plt.plot(Iterations, IS_d8, c = 'y', label = 'd = 8')
plt.plot(Iterations, IS_d16, c = 'pink', label = 'd = 16')
plt.plot(Iterations, sqrt_M,  '--', c = 'purple', label = "1/sqrt(M)", linewidth=3.0)

plt.xscale('log')
plt.yscale('log')
plt.xlabel("Iteration in log-log scale", fontsize=15)
plt.ylabel("Error", fontsize=15)
plt.legend(loc = 'upper right', fontsize=15)
plt.title("Importance Sampling: error vs iteration for different dimensions in Log-Log s
```

Out[12]:
```
Text(0.5, 1.0, 'Importance Sampling: error vs iteration for different dimensions in Log-
Log scale')
```



In [13]:
```
## Rejection Sampling Plot
figure(figsize=(8, 8), dpi=80)
RJ_d1 = df5['1'].values
RJ_d2 = df5['2'].values
RJ_d4 = df5['4'].values
RJ_d8 = df5['8'].values
RJ_d16 = df5['16'].values

plt.plot(Iterations, RJ_d1, c = 'b', label = 'd = 1')
plt.plot(Iterations, RJ_d2, c = 'g', label = 'd = 2')
plt.plot(Iterations, RJ_d4, c = 'r', label = 'd = 4')
plt.plot(Iterations, RJ_d8, c = 'y', label = 'd = 8')
```

```
plt.plot(Iterations, RJ_d16, c = 'pink', label = 'd = 16')
plt.plot(Iterations[2:], sqrt_M[2:],  '--', c = 'purple', label = "1/sqrt(M)", linewidth
plt.legend(loc = 'upper right')
plt.xscale('log')
plt.yscale('log')
plt.xlabel("Iteration in log-log scale", fontsize=15)
plt.ylabel("Error", fontsize=15)
plt.title("Rejection Sampling: error vs iteration for different dimensions in Log-Log sc
```

Out[13]: Text(0.5, 1.0, 'Rejection Sampling: error vs iteration for different dimensions in Log-Log scale')