Stochastic Simulation
Professor: Prof. Fabio Nobile
Assistant: Eliott Van Dieren

Week of 2025-09-15 to 2025-09-19

Lab 2 of Thursday 18th September 2025

Exercice 1.

Consider the random variable X with cumulative distribution function (CDF) $F\colon [-1,3] \to [0,1]$ given by

$$F(x) = \begin{cases} 0\ , & -1 \leqslant x < 0\ , \\ 1 - \frac{2}{3}e^{-x/2}\ , & 0 \leqslant x < 2\ , \\ 1\ , & 2 \leqslant x \leqslant 3\ . \end{cases}$$

Implement the inverse-transform method to generate n independent copies of the random variable X. Assess the quality of the realizations by comparing the empirical CDF with the theoretical CDF F for various values of n.

Solution

By direct computation of the generalized inverse

$$F^{-1}(z) = \arg\inf\{x \in [-1,3] \mid F(x) \geqslant z\} = \begin{cases} -1, & z = 0 \\ 0, & z \in (0, 1/3) \\ -2\log(3(1-z)/2) & z \in [1/3, 1-2e^{-1}/3) \\ 2, & z \in [1-2e^{-1}/3, 1]. \end{cases}$$

Figure 1 illustrates the CDF and generalized inverse CDF of X.

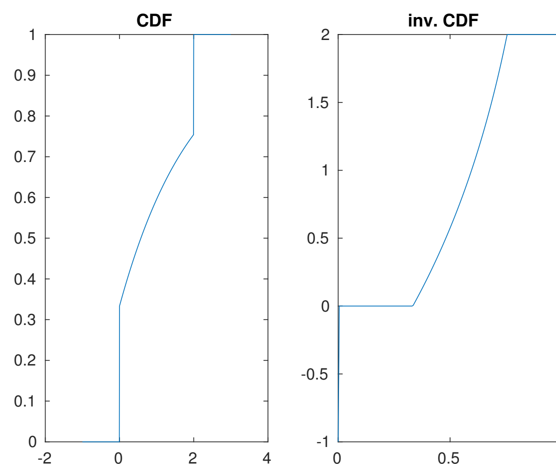


Figure 1: CDF and its inverse

Samples of X may be generated by the inverse transform by $X_i = F^{-1}(U_i)$, where $U_i \sim U([0,1])$. In Figure 2 we compare the resulting empirical CDF F_n for $n = 10, 100, 1000$ with the exact CDF of X. Convergence is observed with the naked eye.

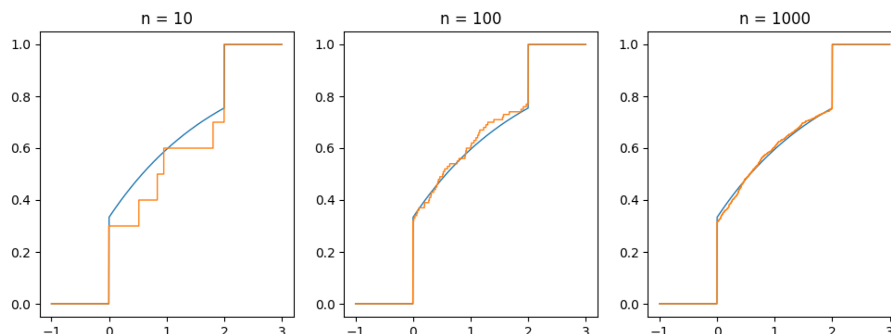


Figure 2: True and empirical CDF comparison

Python code:

```
import numpy as np
import scipy.stats as st
import matplotlib.pyplot as plt
from tools import cdf

# defines the cdf F and pdf f
F = lambda u: (u < 2) * (u >= 0) * (1 - 2 * np.exp(-u / 2.) / 3.) + (u >= 2) * (u <= 3)
f = lambda u: (u >= 0) * (u <= 2) * np.exp(- u / 2.) / 3.

#creates a vector of x and evaluates the cdf F
x = np.linspace(-1, 3, 1000)
y = F(x)

#defines the inverse cdf
intervalPoint = 1-2/3*np.exp(-1)
invF = lambda u: (u < intervalPoint) * (u > 1./3.) * ( - 2 * np.log(3*(1 - u)/2.)) + 2 * (u >
↪ intervalPoint)

# generates plots
n = [10, 100, 1000]
fig = plt.figure(figsize = (12, 4))

for i in range(3):
        U = st.uniform.rvs(size = (n[i],))
        X = invF(U)

        [xx, Nxx] = cdf(X, [-1, 3])
        ax1 = fig.add_subplot(1,3,i+1)
        ax1.plot(x,y, linewidth = 1.)
        ax1.plot(xx, Nxx, linewidth = 1.)
        ax1.set_title('n = ' + str(n[i]))

plt.show()
```

Exercice 2.

Consider the random variable X with probability density function (PDF) f, which is only known up a multiplicative constant. Specifically, let $f(x) := k\tilde{f}(x)$ with

$$\tilde{f}(x) := (\sin^2(6x) + 3\cos^2(x)\sin^2(4x) + 1)e^{-x^2/2} \,,$$

where the normalization constant $k = \frac{1}{\int_{\mathbb{R}} \tilde{f}(x)\,dx}$ is unknown.

1) Argue that $\tilde{f}(x)$ can be bounded by $C\phi(x)$, where C is an appropriately chosen constant and ϕ denotes the PDF of the standard normal distribution, that is $\phi(x) = e^{-x^2/2}/\sqrt{2\pi}$. Find an acceptable value for C.

2) Generate $n = 10^4$ random variables according to the PDF f using the Acceptance–Rejection Method.

   Hint. use `scipy.stats.norm.rvs()` to sample normally distributed random variables in `Python`.

3) Derive an estimate of the normalization constant k, using your procedure's acceptance probability. Compare it to the exact value $k = 0.1696542774$. Furthermore, compare the empirical CDF to the theorized, normalized CDF $F(x) = \int_{-\infty}^{x} f(u)\,du$.

   Hint. you can use the file `trueF_RVG_2.py` from the course's website to plot the theorized CDF.

Solution

1) A rough theoretical estimate for C can be derived through observing that

$$\sup_{x\in\mathbb{R}} \sin(6x)^2 + 3\cos(x)^2\sin(4x)^2 + 1 \leqslant 5,$$

   which implies that

$$\sup_{x\in\mathbb{R}} \frac{\tilde{f}(x)}{\phi(x)} \leqslant 5\sqrt{2\pi} =: \bar{C}.$$

   An improved estimate of C can for instance be derived observing that

$$\frac{\tilde{f}(x)}{\phi(x)} = \frac{\sin(6x)^2 + 3\cos(x)^2\sin(4x)^2 + 1}{\sqrt{2\pi}},$$

   which is a π-periodic function, whose maximimum, by inspection of the plotted graph in `Python`, is contained in the interval $x \in [0, 0.5]$. The maximum can be well approximated by using the built-in `minimize` function within the module `scipy.optimize`. This approach leads to $C = 10.94031 < \bar{C}$, improving the efficiency of the method sightly. Fig. 3 shows that $\tilde{f}$ is bounded by $C\phi$ and that $\tilde{f}/\phi$ is π-periodic whereas Fig. 4 shows the plots of $\tilde{f}$, Cg, f using the estimated k and samples of f using the accept-reject method.

2)-3) From the lecture notes, we know that for $Y \sim N(0,1)$ and $U \sim U([0,1])$

$$\mathbb{P}(U \leqslant \frac{\tilde{f}(Y)}{C\phi(Y)}) = \frac{1}{kC}. \tag{2.1}$$

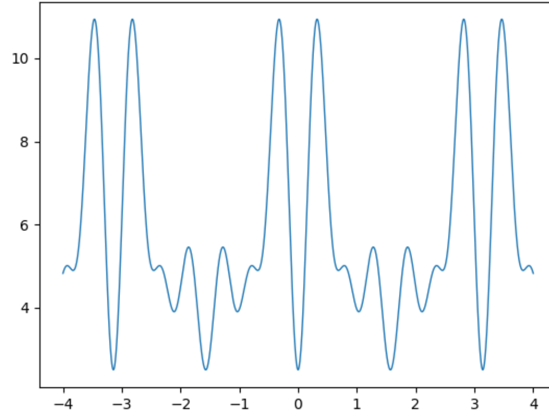


Figure 3: $\tilde{f}/\phi$ plotted over the interval $[0, 2\pi]$.

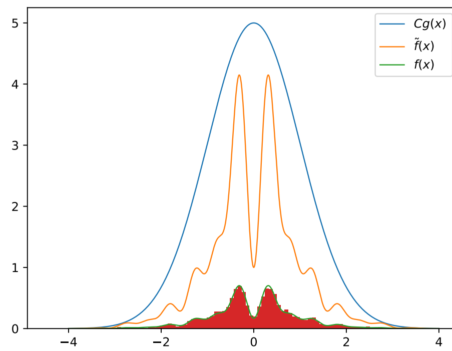


Figure 4: Plots of $\tilde{f}$, Cg, f and samples of f (red).

Letting $U_1, U_2, \ldots, U_N \sim U([0,1])$ and $Y_1, Y_2, \ldots, Y_N \sim N(0,1)$ represent the i.i.d. sequences random variables used for generating an $n = 10^4$ sequence of $X \sim f$ by the AR method (so that $N \in \mathbb{N}$ is a random variable which necessarily is bounded from below by $N \geqslant n$). Equation (2.1) motivates the following approximation of k from the acceptance probability:

$$k = \frac{1}{C\mathbb{P}(U \leqslant \frac{\tilde{f}(Y)}{C\phi(Y)})} \approx \frac{1}{C\frac{1}{N}\sum_{i=1}^{N} \mathbb{1}_{U_i \leqslant \frac{\tilde{f}(Y_i)}{C\phi(Y_i)}}} =: \bar{k}$$

Using this approximation in the `Python` implementation of the AR method that is given below, we obtain $\bar{k} = 0.1694$. Figure 5 compares the AR method's empirical CDF based on $n = 10^4$ samples to that of the true CDF F that can be obtained from the `Python` script that is provided.

Python code:

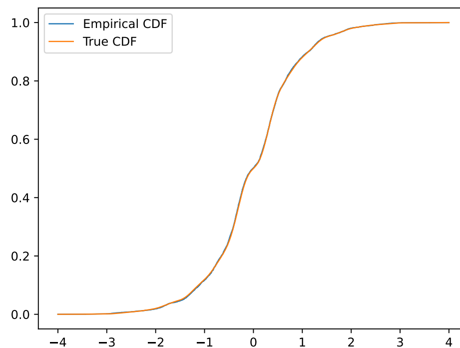


Figure 5: Comparison of the empirical and true CDF's.

```
import numpy as np
import scipy.stats as st
import matplotlib.pyplot as plt
from tools import cdf
from scipy import io

np.random.seed(42)

# creates the un-normalized density
f = lambda u: (np.sin(6*u)**2 + 3 * np.cos(u)**2 * np.sin(4*u)**2 + 1.) * np.exp(- u**2 / 2.)
x = np.linspace(-4, 4, 1001)

#plots
plt.plot(x, f(x)/st.norm.pdf(x), linewidth = 1.)
plt.show()

#defines some hyperparameters
N = 10000
X = np.zeros(N)
i = 0
j = 0
C = 5 * np.sqrt(2 * np.pi)

# Does the AR part
while i < N:
        x_prop = st.norm.rvs()
        U = st.uniform.rvs()
        if U <= f(x_prop) / (C* st.norm.pdf(x_prop)):
                X[i] = x_prop.copy()
                i = i + 1
        j = j + 1

#computes the acceptance rate
acc = float(i) / j
k = 1. / (acc * C)
print('Acceptance rate: ' + str(acc))
print('Normalization constant: ' + str(k))

#plots results
plt.plot(x, C*st.norm.pdf(x), linewidth = 1., label = r'$Cg(x)$')
plt.plot(x, f(x), linewidth = 1., label = r'$\tilde{f}(x)$')
plt.plot(x, k * f(x), linewidth = 1., label = r'$f(x)$')
```

```
plt.hist(X, bins = 100, density = True)
plt.legend()
plt.show()

truecdf = io.loadmat('truecdf.mat')['truecdf'][0,:]
truex = np.linspace(-4, 4, 1601)
xx, Nxx = cdf(X, [-4, 4])
plt.plot(xx, Nxx, linewidth = 1., label = 'Empirical CDF')
plt.plot(truex, truecdf, linewidth = 1., label = 'True CDF')
plt.legend()

plt.show()
```

Exercice 3.

An element $(x, y) \in \mathbb{R}^2$ may be represented by its polar coordinates $(\rho, \Theta) \in [0, \infty) \times [0, 2\pi)$ defined by

$$\rho(x, y) = \sqrt{x^2 + y^2},$$

and

$$\Theta(x, y) = \begin{cases} \tan^{-1}\left(\frac{y}{x}\right) & \text{if } x > 0 \text{ and } y \geqslant 0, \\ \tan^{-1}\left(\frac{y}{x}\right) + \pi & \text{if } x < 0, \\ \tan^{-1}\left(\frac{y}{x}\right) + 2\pi & \text{if } x > 0 \text{ and } y \leqslant 0, \\ 0 & \text{if } x = y = 0, \end{cases}$$

where $\tan^{-1} : \mathbb{R} \to (-\pi/2, \pi/2)$.

1) Show by calculation that if the random variables $X, Y \sim N(0, 1)$ are independent, then the polar coordinate representation of (X, Y) satisfies

$$\rho^2 \sim \exp(1/2) \quad \text{and} \quad \Theta \sim U([0, 2\pi)).$$

Show further that ρ and Θ are independent.

2) In the opposite direction, show that if $\rho^2 \sim \exp(1/2)$ and $\Theta \sim U([0, 2\pi))$ and ρ and Θ are independent, then the Cartesian representation of the polar coordinates (ρ, Θ),

$$X = \rho \cos(2\pi\Theta) \quad \text{and} \quad Y = \rho \sin(2\pi\Theta),$$

satisfies $X, Y \sim N(0, 1)$ with X and Y being independent.

3) In order to construct an Acceptance–Rejection (AR) method for generating standard normal random variables consider the auxiliary PDF $g(x) = e^{-|x|}/2$. For your auxiliary PDF, determine a $C \geqslant 1$ such that

$$\frac{e^{-x^2/2}}{\sqrt{2\pi}} \leqslant Cg(x), \qquad \forall x \in \mathbb{R}\,.$$

Hint. See lecture notes for how to sample from the PDF g.

4) Implement the above AR method and the Box–Muller method in `Python` and use the built-in timer function `time()` within the `time` module, to compare the performance of the respective methods in terms of runtime per sample.

Solution

1) For $X, Y \sim N(0,1)$ and independent, the joint PDF satisfies

$$f_{X,Y}(x,y) = f_X(x) f_Y(y) = \frac{e^{-(x^2+y^2)/2}}{2\pi},$$

due to the independence of X and Y. Consequently,

$$\begin{aligned}
F_{\rho^2}(r) &= \mathbb{P}(\rho^2 \leqslant r) \\
&= \mathbb{P}(X^2 + Y^2 \leqslant r) \\
&= \int_{\mathbb{R}^2} \mathbb{1}_{x^2+y^2 \leqslant r} f_{X,Y}(x,y) dxdy \\
&= \int_{\mathbb{R}^2} \mathbb{1}_{x^2+y^2 \leqslant r} \frac{e^{-(x^2+y^2)/2}}{2\pi} dxdy \\
&= \int_0^{\sqrt{r}} \int_0^{2\pi} \frac{e^{-s^2/2}}{2\pi} s d\theta ds \\
&= 1 - e^{-r/2} \\
&= F_{\exp(1/2)}(r).
\end{aligned} \tag{3.1}$$

Next, for any $\hat{\theta} \in [0, 2\pi)$

$$\begin{aligned}
F_\Theta(\hat{\theta}) &= \mathbb{P}(\Theta(X,Y) \leqslant \hat{\theta}) \\
&= \int_{\mathbb{R}^2} \mathbb{1}_{\Theta(x,y) \leqslant \hat{\theta}} f_{XY}(x,y) dxdy \\
&= \int_0^{\hat{\theta}} \int_0^{\infty} \frac{e^{-s^2/2}}{2\pi} s ds d\theta \\
&= \frac{\hat{\theta}}{2\pi} \\
&= F_{U([0,2\pi))}(\hat{\theta}).
\end{aligned}$$

To verify that ρ^2 and Θ are independent, note first that (3.1) implies that $F_\rho(r) = 1 - e^{-r^2/2}$. Furthermore,

$$\begin{aligned}
F_{\rho,\Theta}(r,\hat{\theta}) &= \mathbb{P}(\rho \leqslant r, \Theta \leqslant \hat{\theta}) \\
&= \mathbb{P}(\rho \leqslant r, \Theta \leqslant \hat{\theta}) \\
&= \int_{\mathbb{R}^2} \mathbb{1}_{\sqrt{x^2+y^2} \leqslant r} \mathbb{1}_{\Theta(x,y) \leqslant \hat{\theta}} f_{X,Y}(x,y) dxdy \\
&= \int_0^r \int_0^{\hat{\theta}} \frac{e^{-s^2/2}}{2\pi} s d\theta ds \\
&= \int_0^r e^{-s^2/2} s ds \int_0^{\hat{\theta}} \frac{1}{2\pi} d\theta \\
&= F_\rho(r) F_\Theta(\hat{\theta}).
\end{aligned}$$

2) Since ρ and Θ are independent, the following holds for all $r \geqslant 0$ and $\theta \in [0, 2\pi)$,

$$f_{\rho,\Theta}(r,\theta) = f_\rho(r) f_\Theta(\theta) = \frac{d}{dr} F_\rho(r) \frac{d}{d\theta} F_\Theta(\theta) = \frac{re^{-r^2/2}}{2\pi}.$$

To show that X and Y are independent standard normals, we have for any $(\hat{x}, \hat{y}) \in \mathbb{R}^2$,

$$\begin{aligned}
F_{X,Y}(\hat{x},\hat{y}) &= \mathbb{P}(X \leqslant \hat{x}, Y \leqslant \hat{y}) \\
&= \mathbb{P}(\rho\cos(2\pi\Theta) \leqslant \hat{x}, \rho\sin(2\pi\Theta) \leqslant \hat{y}) \\
&= \int_0^\infty \int_0^{2\pi} \mathbb{1}_{r\cos(2\pi\theta)\leqslant\hat{x}} \mathbb{1}_{r\sin(2\pi\theta)\leqslant\hat{y}} f_{\rho,\Theta}(r,\theta) d\theta dr \\
&= \int_0^\infty \int_0^{2\pi} \mathbb{1}_{r\cos(2\pi\theta)\leqslant\hat{x}} \mathbb{1}_{r\sin(2\pi\theta)\leqslant\hat{y}} \frac{re^{-r^2/2}}{2\pi} d\theta dr \\
&= \int_{\mathbb{R}^2} \mathbb{1}_{x\leqslant\hat{x}} \mathbb{1}_{y\leqslant\hat{y}} \frac{e^{-(x^2+y^2)/2}}{2\pi} dy dx \\
&= \int_{-\infty}^{\hat{x}} \frac{e^{-x^2/2}}{\sqrt{2\pi}} dx \int_{-\infty}^{\hat{y}} \frac{e^{-y^2/2}}{\sqrt{2\pi}} dy \\
&= F_{N(0,1)}(\hat{x}) F_{N(0,1)}(\hat{y}).
\end{aligned}$$

Here, the fifth equality involved a change of variables $(r, \theta) \mapsto (x, y)$ from polar to Cartesian coordinates.

3) The smallest possible constant C is given by

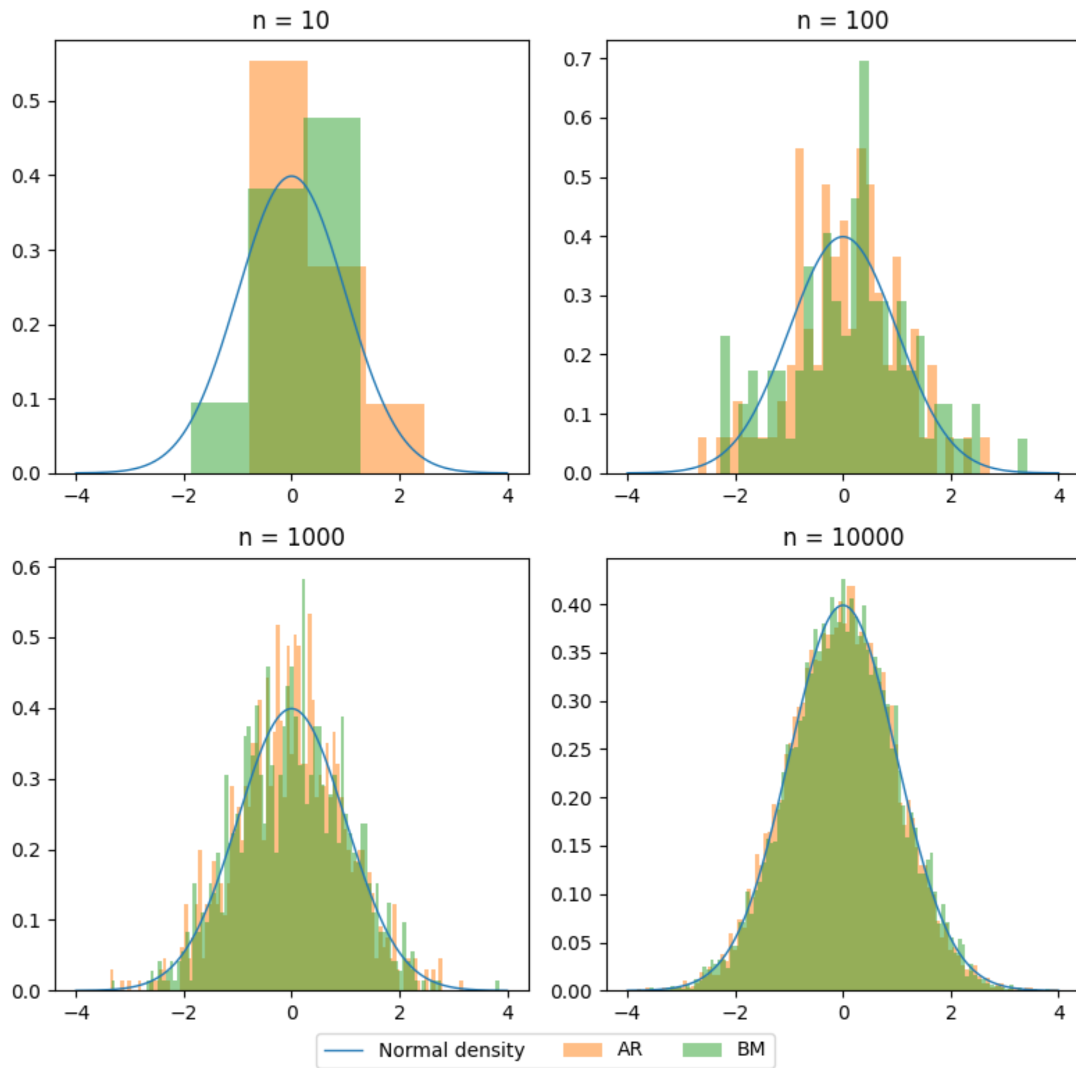
$$C = \sup_{x\in\mathbb{R}} \frac{e^{-x^2/2}}{\sqrt{2\pi} g(x)} = \sup_{x\in\mathbb{R}} \sqrt{\frac{2}{\pi}} e^{\sup_{x\in\mathbb{R}} -x^2/2+|x|} = e^{1/2}\sqrt{\frac{2}{\pi}}.$$

4) For the AR method, we sample random variables $X \sim g$ from the Laplace distribution

$$g(x) = \frac{1}{2} e^{-x} \mathbb{1}_{x\geqslant 0} + \frac{1}{2} e^{x} \mathbb{1}_{x<0},$$

by

$$X = (2B-1)Y, \qquad B \sim \text{Bernoulli}(1/2) \text{ and } Y \sim \text{Exp}(1/2).$$



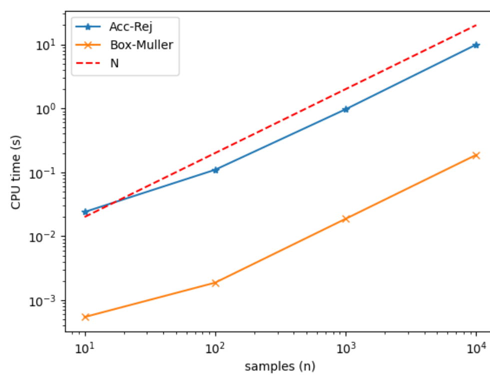


Figure 6

`Python` code for the implementation of the AR and the Box–Muller methods is given below. Figure 6 shows histograms of the samples obtained with the two methods, as well as a runtime per sample comparison. We observe that for both methods the cost of generating n samples is asymptotically $O(n)$ – as one should expect – and that asymptotically the Box–Muller method is roughly 10 times faster than the proposed AR method (in this implementation at least).

Python code:

```
import numpy as np
import scipy.stats as st
import matplotlib.pyplot as plt
import time

np.random.seed(42)

#defines number of samples
N = 10000

#generates samples X and Y
X = st.norm.rvs(size = (N,))
Y = st.norm.rvs(size = (N,))

#computes R and theta
R = np.sqrt(X**2 + Y**2)
theta = np.zeros(X.shape) #preallocates
#Does the actual computation
for i in range(X.shape[0]):
    angle = np.arctan(Y[i]/X[i])
    theta[i] = (X[i] > 0) * (Y[i] >= 0) * (angle) + (X[i] < 0) * (angle + np.pi)\
             + (X[i] > 0) * (Y[i] <= 0) * (angle + 2 * np.pi)

x = np.linspace(0, 10, 1001)

#fig = plt.figure(figsize = (9,4))
#ax1 = fig.add_subplot(121)
#ax1.plot(x, st.expon.pdf(x, scale = 2.), linewidth = 1,)
#ax1.hist(R**2, bins = 100, density = True)
#ax2 = fig.add_subplot(122)
#ax2.hist(theta, bins = 100, density = True)

#plt.show()

N = [10, 100, 1000, 10000]

times_AR = np.zeros(4)
times_BM = np.zeros(4)
fig = plt.figure(figsize = (8,8))
for n in N:
    # ---- Acc-Rej method

    i = 0
    j = 0
    C = np.exp(1./2) * np.sqrt(2./np.pi)
    X_acc = np.zeros(n)
    start = time.time()
    while i < n:
        B = st.bernoulli.rvs(0.5)
        Y = st.expon(scale = 2).rvs()
        X_prop = (2 * B - 1.) * Y
        U = st.uniform.rvs()
        if U < st.norm.pdf(X_prop) / (C * np.exp(-np.abs(X_prop) / 2.)):
```

```
            X_acc[i] = X_prop.copy()
            i = i + 1
            j = j + 1
    end_acc = time.time()

        # ---- Box-Muller method
    # this is a very inefficient implementation of the BM algorithm
    #as it not vectoized. However, we chose to do so in order to have a
    # more fair comparison between A.R and B.M For a more efficient
    # implementation, please uncomment bellow.
    #        U = st.uniform.rvs(size = (n,))
    #        V = st.uniform.rvs(size = (n,))
    #        rho = np.sqrt(- 2 * np.log(U))
    #        theta = 2 * np.pi * V
    #        X_bm = rho * np.cos(theta)
    #        Y_bm = rho * np.sin(theta)
    #inefficient implementation
    U=np.zeros(n)
    V=np.zeros(n)
    X_bm=np.zeros(n)
    Y_bm=np.zeros(n)

    for i in range(n):
        U[i] = st.uniform.rvs(size = (1,))
        V[i] = st.uniform.rvs(size = (1,))
        rho=np.sqrt(-2*np.log(U[i]))
        theta=2*np.pi*V[i]
        X_bm[i] = rho * np.cos(theta)
        Y_bm[i] = rho * np.sin(theta)

    end_bm = time.time()

    time_AR = end_acc - start
    time_BM = end_bm - end_acc

    # print end_acc, end_bm
    # print(time_AR, time_BM)

    x = np.linspace(-4, 4, 1001)
    if n == 10:
        i0 = 1
    elif n == 100:
        i0 = 2
    elif n == 1000:
        i0 = 3
    else:
        i0 = 4

    #plots results
    times_AR[i0-1] = time_AR
    times_BM[i0-1] = time_BM
    ax = fig.add_subplot(2,2,i0)
    ax.plot(x, st.norm.pdf(x), linewidth = 1.)
    ax.hist(X_acc, bins = int(np.min([n**1/3,100])), density = True, alpha = 0.5)
    ax.hist(X_bm, bins = int(np.min([n**1/3,100])), density = True, alpha = 0.5)
    ax.set_title('n = ' + str(n))
    # plt.legend(['Normal density','AR','BM'])

handles, labels = ax.get_legend_handles_labels()
fig.legend(['Normal density', 'AR', 'BM'],
           loc='lower center',
           ncol=3,
           bbox_to_anchor=(0.5, 0.02))
```

```python
plt.tight_layout(rect=[0, 0.05, 1, 1])
plt.show()

plt.loglog(N, times_AR, '-*', label = 'Acc-Rej')
plt.loglog(N, times_BM, '-x', label = 'Box-Muller')
plt.loglog(N,0.002* np.array(N),color='r',linestyle= '--', label = 'N')
plt.legend()
plt.ylabel('CPU time (s)')
plt.xlabel('samples (n)')
plt.show()
```

Exercice 4.

Let $X = (X_1, X_2, \dots, X_n)^T \sim \mathcal{U}((0,1)^n)$ and denote by $X_{(1)} \leqslant X_{(2)} \leqslant \dots \leqslant X_{(n)}$ the ordered sample (i.e. the order statistic).

1) Implement a procedure to generate the order statistic $X_{(1)} \leqslant X_{(2)} \leqslant \dots \leqslant X_{(n)}$, $n \in \mathbb{N}$, based on *sorting* a collection X of i.i.d. uniform random variables.

2) Prove the following properties:

   a) Show that
   $$\mathbb{P}(X_{(j)} \leqslant x) = \sum_{i=j}^{n} \binom{n}{i} x^i (1-x)^{n-i} ,$$
   for any $x \in (0,1)$. Furthermore, use this fact to infer the distribution of the random variable $\max\{X_1, X_2, \dots, X_n\}$.

   b) Then show that
   $$\mathbb{P}(X_{(j)} \leqslant z \big| X_{(k)} = x_k \ , \ \forall \, k > j) = \left(\frac{z}{x_{j+1}}\right)^j ,$$
   for all $z \leqslant x_{j+1}$ and any $j < n$.

3) Use the facts above to implement a procedure that enables generating copies from the order statistic $X_{(1)} \leqslant X_{(2)} \leqslant \dots \leqslant X_{(n)}$ *without sorting*. Compare this procedures and the procedure based on sorting with respect to time for various values of n. What do you observe? Explain.

   Hint. To measure time in Python, you can use

   ```python
   import time

   start = time.time()
   #your code goes here
   end = time.time()
   print end - start
   ```

4) Implement a procedure that generates uniform random vectors in the unit simplex
   $$\mathcal{S} = \left\{ (x_1, x_2, \dots, x_n)^T \in \mathbb{R}^n : x_i \geqslant 0 \; \forall \, i \ , \ \sum_{i=1}^{n} x_i \leqslant 1 \right\} .$$
   Assess your sampling procedure by visualizing $N = 1000$ sampling points for $n = 3$

Hint. You can use the following template to do a 3D scatter plot in Python.

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt



fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')


#Defines the values. CHANGE THESE LINES
x =[1,2,3,4,5,6,7,8,9,10]
y =[5,6,2,3,13,4,1,2,4,8]
z =[2,3,3,3,5,7,9,11,9,10]



ax.scatter(x, y, z, c='r', marker='o')

ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')

plt.show()
```

Hint. Notice that a vector, whose coordinates are distributed according to the order statistic of a collection of i.i.d. $\mathcal{U}(0,1)$, takes values in the "wedge" $\mathcal{W} = \{(u_1, u_2, \dots, u_n)^T \in \mathbb{R}^n : 0 \leqslant u_i \leqslant 1 \ \forall\, i\ ,\ u_1 \leqslant u_2 \leqslant \dots \leqslant u_n\}$. The unit simplex $\mathcal{S}$ is then simply the image of the "wedge" $\mathcal{W}$ under the linear transformation $x = Au$ where

$$A = \begin{pmatrix} 1 & 0 & \dots & 0 \\ -1 & 1 & \dots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & -1 & 1 \end{pmatrix}.$$

Solution

1) See code.

2) a) Let $j \in \{1, \dots, n\}$ and $x \in (0,1)$, then

$$\mathbb{P}(X_{(j)} \leqslant x) = \sum_{i=j}^{n} \mathbb{P}(X_{(1)}, \dots, X_{(i)} \in (0,x), X_{(i+1)}, \dots, X_{(n)} \in (x,1)) = \sum_{i=j}^{n} \binom{n}{i} x^i (1-x)^{n-i},$$

where the second equality follows from the independence of the X_j's and the fact that there are $\binom{n}{i}$ ways to split n objects in two groups of size i and $n-i$, respectively. This, in particular, implies

$$\mathbb{P}(\max_j X_j \leqslant x) = \mathbb{P}(X_{(n)} \leqslant x) = x^n.$$

b) Assume without loss of generality $0 < x_1 < x_2 < \dots < x_n < 1$. Then, the joint density of $(X_{(1)}, \dots, X_{(n)})$ is given by

$$f_{X_{(1)},\dots,X_{(n)}}(x_1, \dots, x_n) = n!$$

Now, fix $j < n$, and condition on $X_{(k)} = x_k$ for all $k > j$. By Bayes' theorem, the conditional density of $(X_{(1)}, \dots, X_{(j)})$ is given by

$$f_{X_{(1)},\dots,X_{(j)}|X_{(k)}=x_k,k>j}(x_1, \dots, x_j) = \frac{f_{X_{(1)},\dots,X_{(n)}}(x_1, \dots, x_n)}{f_{X_{(j+1)},\dots,X_{(n)}}(x_{j+1}, \dots, x_n)}$$

However, the denominator is given by integrating the joint density over the variables $x_1, \dots, x_j$, i.e.

$$f_{X_{(j+1)},\dots,X_{(n)}}(x_{j+1}, \dots, x_n) = \int_{0<x_1<\dots<x_j<x_{j+1}} n!\, dx_1 \dots dx_j = \frac{n!}{j!} x_{j+1}^j,$$

which implies

$$f_{X_{(1)},\dots,X_{(j)}|X_{(k)}=x_k,k>j}(x_1, \dots, x_j) = \frac{j!}{x_{j+1}^j}.$$

In particular, this implies that the conditional CDF $X_{(j)}$ is given by

$$\mathbb{P}(X_{(j)} \leqslant z | X_{(k)} = x_k, k > j) = \int_{0<x_1<\dots<x_j\leqslant z} \frac{j!}{x_{j+1}^j}\, dx_1 \dots dx_j = \frac{z^j}{x_{j+1}^j},$$

by integrating over the simplex $\{(x_1, \dots, x_j) \in (0, 1)^j : 0 < x_1 < \dots < x_j \leqslant z\}$. This concludes the proof.

Alternatively, one could intuitively argue that conditional on $X_{(j+1)} = x_{j+1}$, the random variables $X_{(1)}, \dots, X_{(j)}$ are distributed as the order statistics of j i.i.d. $\mathcal{U}(0, x_{j+1})$ random variables. This gives the result by scaling. Yet another alternative argument is to use induction on j, starting from the base case $j = n - 1$.

3) The above computations enable us to sample from $X_{(j)} \leqslant z | X_{(k)} = x_k\ ,\ \forall\, k > j$ by, for example, the inversion method. See the code for the details on the implementation. This procedure, in particular, features a complexity which is $O(n)$, instead of the $O(n \log(n))$ featured by the sorting-based sampling algorithm. The comparison of running times is shown in Figure 7.

4) Using the hint, we can generate samples in the simplex by first sampling the order statistics $X = (X_{(1)}, \dots, X_{(n)})^T$, and then computing AX. The results are shown in Figure 8

Python code:

```
import numpy as np
import scipy.stats as st
import matplotlib.pyplot as plt
import time
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt

np.random.seed(42)

### Point 1 ###
```

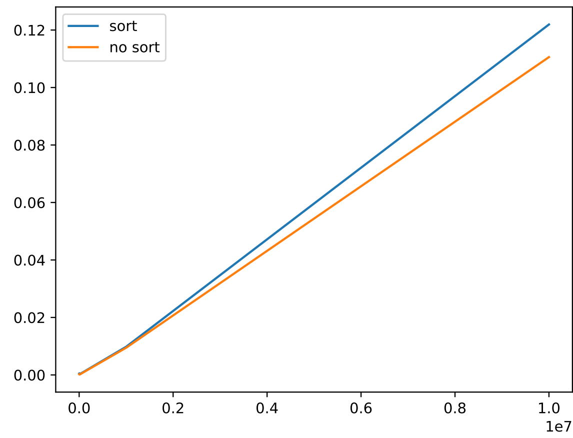


Figure 7: Time comparison of sorting-based and inversion-based sampling.

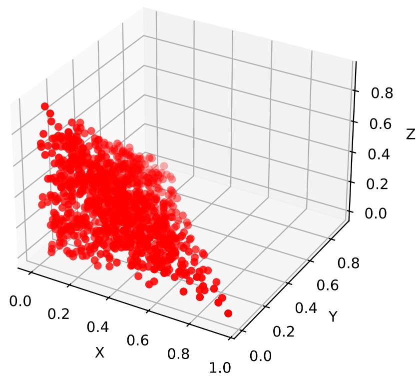


Figure 8: Visualization of a sample of size $N = 1000$ in the 3D simplex.

```
n = 10


# Generate order statistic
def generate_ord_stat_sort(n):
    X = st.uniform.rvs(size=n)
    return np.sort(X)


### Point 3 ###
def generate_ord_stat_nosort(n):
    X = np.empty(n)
```

```python
    x = 1
    for i in range(0, n):
        # Sample from X_{(j)}|X_{(j+1)}=x by inversion
        U = st.uniform.rvs()
        X[n - i - 1] = x * U ** (1 / (n - i))
        x = X[n - i - 1]

    return X


# Vectorize previous function
def generate_ord_stat_nosort_vec(n):
    U = st.uniform.rvs(size=n)
    X = U ** (1 / (np.arange(1, n + 1)))
    X = np.cumprod(X[::-1])[::-1]

    return X


# Compare procedures

n_grid = 10 ** np.arange(3, 8)
time_sort = []
time_nosort = []

for n in n_grid:
    # Compute time for sorting-based sampling
    start = time.time()
    X_ord = generate_ord_stat_sort(n)
    end = time.time()
    time_sort.append(end - start)

    # Compute time for inversion-based sampling
    start = time.time()
    X_ord = generate_ord_stat_nosort_vec(n)
    end = time.time()
    time_nosort.append(end - start)

plt.plot(n_grid, time_sort, label="sort")
plt.plot(n_grid, time_nosort, label="no sort")
# plt.xscale('log')
# plt.yscale('log')
plt.legend()
plt.show()


### Point 4 ###
def generate_uniform_on_unit_simplex(num_samps, n):
    X = np.empty((num_samps, n))

    for i in range(num_samps):
        # Generate order statistics
        X_ord = generate_ord_stat_nosort_vec(n)
        X_old = X_ord.copy()
        X_ord[1:] = X_ord[1:] - X_ord[: n - 1]

        X[i, :] = X_ord

    return X


n = 3
num_samps = 1000
```

```
X = generate_uniform_on_unit_simplex(num_samps, n)

fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")

# Defines the values
x = X[:, 0]
y = X[:, 1]
z = X[:, 2]

ax.scatter(x, y, z, c="r", marker="o")

ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("Z")

plt.show()
```

(Optional) Exercice 5.

The PDF for the Cauchy distribution centered at $x_0 \in \mathbb{R}$ and with scale parameter $\gamma \in \mathbb{R}$ is given by

$$f(x; x_0, \gamma) = \frac{1}{\pi\gamma\left(1 + \left(\frac{x - x_0}{\gamma}\right)^2\right)}.$$

1) Show by integration of the PDF that the CDF of the Cauchy distribution with $x_0 = 0$ and $\gamma = 1$ is given by $F(x; 0, 1) = \tan^{-1}(x)/\pi + 1/2$, for $\tan^{-1} : \mathbb{R} \to (-\pi/2, \pi/2)$.

2) Show that if $X_1, X_2 \sim N(0, 1)$ are independent, then $X = X_1/X_2$ is Cauchy distributed with $x_0 = 0$ and $\gamma = 1$.

3) Based on the information from the 5.1 and 5.2, describe and implement two algorithms for sampling $X \sim F(\cdot; 0, 1)$ in `Python`. Compare the performance of the respective algorithms in terms of runtime per sample.

4) It is possible to extend the preceding methods to sample from $X \sim F(\cdot; x_0, \gamma)$ for any $x_0 \in \mathbb{R}$ and $\gamma > 0$. How?

Solution

1) Recalling that

$$\frac{d}{dx} \tan^{-1}(x) = \frac{1}{1 + x^2},$$

we have for $X \sim \text{Cauchy}(0, 1)$ that

$$F_X(x) = \mathbb{P}(X \leqslant x) = \int_{-\infty}^{x} \frac{1}{\pi(1 + y^2)} dy = \left[\frac{1}{\pi} \tan^{-1}(x)\right]_{-\infty}^{x} = \frac{\tan^{-1}(x)}{\pi} + \frac{1}{2}.$$

2) For $X = X_1/X_2$ where $X_1, X_2 \sim N(0,1)$ are independent, we make use of a Cartesian to polar coordinates transformation $(x,y) \mapsto (r,\theta)$ to show that

$$\begin{aligned}
F_X(\hat{x}) = \mathbb{P}(X \leqslant \hat{x}) = \mathbb{P}(\frac{X_1}{X_2} \leqslant \hat{x}) \\
&= \int_{\mathbb{R}^2} \mathbb{1}_{y/x \leqslant \hat{x}} \frac{e^{-(x^2+y^2)/2}}{2\pi} dxdy \\
&= \int_{-\pi/2}^{\pi/2} \frac{\mathbb{1}_{\tan(\theta) \leqslant \hat{x}}}{\pi} d\theta \int_0^\infty r e^{-r^2/2} dr \\
&= \int_{-\pi/2}^{\tan^{-1}(\hat{x})} \frac{1}{\pi} d\theta = \frac{\tan^{-1}(\hat{x})}{\pi} + \frac{1}{2}.
\end{aligned}$$

3) The pair of sampling methods to compare is one following **5.1**, sampling $X \sim \text{Cauchy}(0,1)$ by the inversion method with $U \sim U([0,1])$ and

$$X = F^{-1}(U) = \tan\Big(\pi\Big(U - \frac{1}{2}\Big)\Big),$$

and one following the ratio of Gaussians approach of 5.2. A vectorized `Python` implementation is given below. Figure 9 compares the two approaches in terms of runtime per sample. The inversion method is slightly faster than the ratio of Gaussians based method. The reason might be that while the fomer method only needs to generate one random variable for every ouput, the latter needs to generate two.

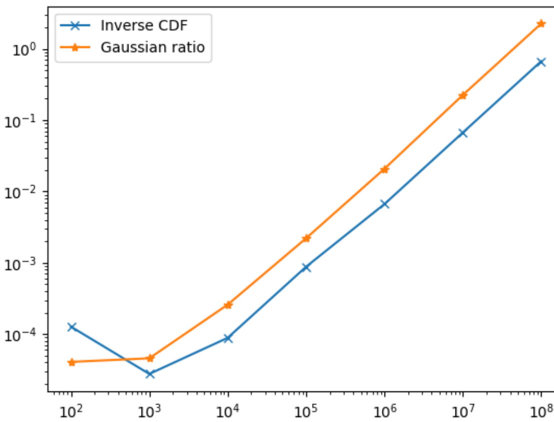


Figure 9

Python code:

```
import numpy as np
import scipy.stats as st
import matplotlib.pyplot as plt
import time
```

```python
np.random.seed(42)

N = [100, 1000, 10000, int(1e+5), int(1e+6), int(1e+7), int(1e+8)]
times = np.zeros((2, 7))
for i in range(len(N)):
        start = time.time()
        # 1st approach - Inverse CDF
        U = st.uniform.rvs(size = (N[i],))
        X1 = np.tan(np.pi*(U - 1./2))
        end1 = time.time()
        # 2nd approach
        Y1 = st.norm.rvs(size = (N[i],))
        Y2 = st.norm.rvs(size = (N[i],))
        X2 = Y1/Y2
        end2 = time.time()

        #plt.hist(X1, bins = 100, density = True, alpha = 0.5)
        #plt.hist(X2, bins = 100, density = True, alpha = 0.5)

        #plt.show()
        times[0,i] = end1 - start
        times[1,i] = end2 - end1
        print( '-'*5 + str(N[i]) + ' samples ' + '-'*5)
        print( 'Inverse CDF method: ' + str(end1 - start))
        print( 'Gaussian ratio method: ' + str(end2 - end1))


plt.plot(N, times[0,:], '-x', label = 'Inverse CDF')
plt.plot(N, times[1,:], '-*', label = 'Gaussian ratio')
plt.xscale('log')
plt.yscale('log')
plt.legend()

plt.show()
```

4) First generate an $X \sim \text{Cauchy}(0, 1)$ with either of the above AR methods and thereafter set $Y = \gamma(X + x_0)$. Then

$$F_Y(y) = \mathbb{P}(Y \leqslant y) = \mathbb{P}(X \leqslant \frac{y - x_0}{\gamma}) = \frac{\tan^{-1}\left(\frac{y-x_0}{\gamma}\right)}{\pi} + \frac{1}{2},$$

and

$$f_Y(y) = \frac{d}{dy} F_Y(y) = \frac{1}{\pi\gamma\left(1 + \left(\frac{y-x_0}{\gamma}\right)^2\right)} = f(y; x_0, \gamma).$$

(Optional) Exercice 6.

The density of a random variable X, given a sample $X_1, X_2, \dots, X_n \overset{\text{iid}}{\sim} X$, may be approximated by a mixed distribution generated by so called kernel density estimation. In its simplest form, kernel density estimation consists of the following steps:

(i) Choose a so called kernel function $K \in \{f : \mathbb{R} \to \mathbb{R}_+ \mid \|f\|_{L^1(\mathbb{R})} = 1\}$. (So the kernel is itself a PDF.)

(ii) For some $n \in \mathbb{N}$, generate a sequence of i.i.d. random variable $X_1, X_2, \dots, X_n$ from the distribution of X.

(iii) Define the kernel density estimator by

$$f(x) = \frac{1}{n}\sum_{i=1}^{n} K_\delta(x - X_i),$$

where

$$K_\delta(x - X_i) := \frac{1}{\delta} K\Big(\frac{x - X_i}{\delta}\Big), \quad \delta > 0,$$

and δ is an appropriately chosen scaling parameter relating to the width of the kernel.

The Burr type XII distribution has the CDF

$$F(x; \alpha, c, k) = \begin{cases} 0 & x \leqslant 0 \\ 1 - \frac{1}{\left(1+\left(\frac{x}{\alpha}\right)^c\right)^k}, \quad x \in (0, \infty), & x > 0, \end{cases}$$

with parameters $\alpha, c, k > 0$.

1) Consider the Gaussian density kernel function

$$K(x) = \frac{e^{-x^2/2}}{\sqrt{2\pi}},$$

and implement a kernel density estimator for

$$X \sim \mathrm{BurrXII}(\alpha = 1, c = 2, k = 4)$$

in `Python`. Samples of X can be obtained using the `scipy.stats` class `burr12`. In your code, for varying $n = [100, 10^5]$ and $\delta(n) = n^{-1/5}$, compute kernel density estimators

$$f_n(x) = \frac{1}{n}\sum_{i=1}^{n} K_{\delta(n)}(x - X_i),$$

and plot f_n and the PDF of BurrXII(1,2,4). Furthermore, for each value of n sample $N = 200000$ i.i.d. random variables $Y_i^n \sim f_n(x)$ by means of the composition method.

Hint. The `numpy.random` built-in function `randint` and might come handy.

2) Study how well the empirical CDF of $Y_1^n, Y_2^n, \ldots, Y_N^n$, which we denote by $F_n^N(x)$, converges towards $F(x; 1, 3, 1)$. That is, investigate how fast

$$D_n^N = \sup_{x \in [-2,5]} |F_n^N(x) - F(x; 1, 2, 4)|$$

decreases as n increases.

Solution

A `Python` code implementation is given below. From the runs plotted in Figure 10, we observe in eye-measure that both the output PDFs and CDFs converges to their respective counterpars of BurrXII(1,2,4) as n grows. Figure 11 shows that the Kolmogorov–Smirnov test error decreases roughly decays like $O(n^{-1/4})$.

Python code:

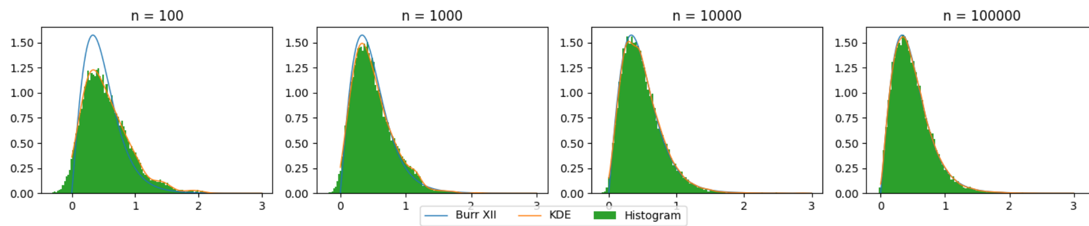


Figure 10

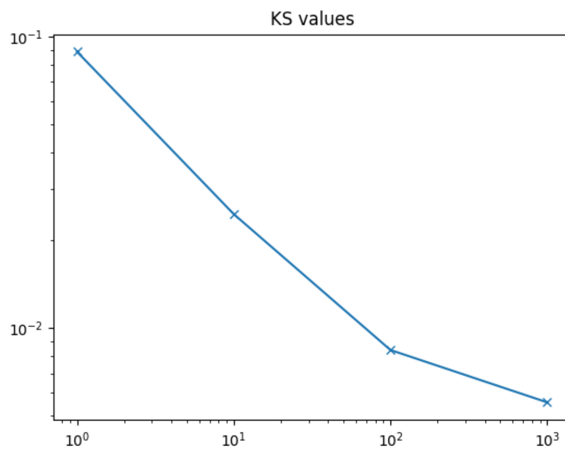


Figure 11

```
import numpy as np
import scipy.stats as st
import matplotlib.pyplot as plt
import math
from tools import *

np.random.seed(42)

#Defines the KDE function
def kde(X, data, h = None):
        N = data.shape[0]
        if h is None:
                h = 1.06 * np.std(data) * N ** (-1. / 5)
        pdf = np.zeros(X.shape[0])
        for i in range(X.shape[0]):
                pdf[i] = np.sum(np.exp( - ((X[i] - data) / h)** 2 / 2.) / np.sqrt(2*math.pi)) / (N
                 ↪  * h)
        return pdf
# Determines some parameters
c, d = 2, 4
N = 1000
#generates samples form the Bur XII distr
X = st.burr12(c, d).rvs(size = (N,))

#plots density and KDE
```

```python
x = np.linspace(0, 3, 1001)
plt.plot(x, st.burr12(c,d).pdf(x), linewidth = 1.)
plt.plot(x, kde(x, X), linewidth = 1.)
plt.hist(X, bins = 100, density = True)
plt.legend(['Burr XII','KDE','Histogram'])

plt.show()

#Prepares to plot
n = [100 * 10**i for i in range(4)]
delta = np.array(n) ** (-1. / 5.)
print(n)
#plots
test_vals = np.zeros(len(n))
fig = plt.figure(figsize = (14,3))
for i in range(len(n)):
        X = st.burr12(c,d).rvs(size = (n[i],))
        h = np.std(X) * delta[i]
        w = np.random.randint(0, high = n[i], size = 20000)
        X_kde = np.random.normal(loc = X[w], scale = h)
        ax = fig.add_subplot(1,4,i+1)
        ax.plot(x, st.burr12(c,d).pdf(x), linewidth = 1.)
        ax.plot(x, kde(x, X, h = h), linewidth = 1.)
        ax.hist(X_kde, bins = 100, density = True)
        ax.set_title('n = ' + str(n[i]))
        mess, stat = KSTest(X_kde, alpha = 0.1)
        test_vals[i] = stat['Statistic']
        print(mess)


handles, labels = ax.get_legend_handles_labels()
fig.legend(['Burr XII', 'KDE', 'Histogram'],
           loc='lower center',
           ncol=3,
           bbox_to_anchor=(0.5, 0.02))
plt.tight_layout(rect=[0, 0.05, 1, 1])
plt.show()


plt.loglog(10.0**np.arange(4),test_vals, '-x')
plt.title('KS values')
plt.show()
```