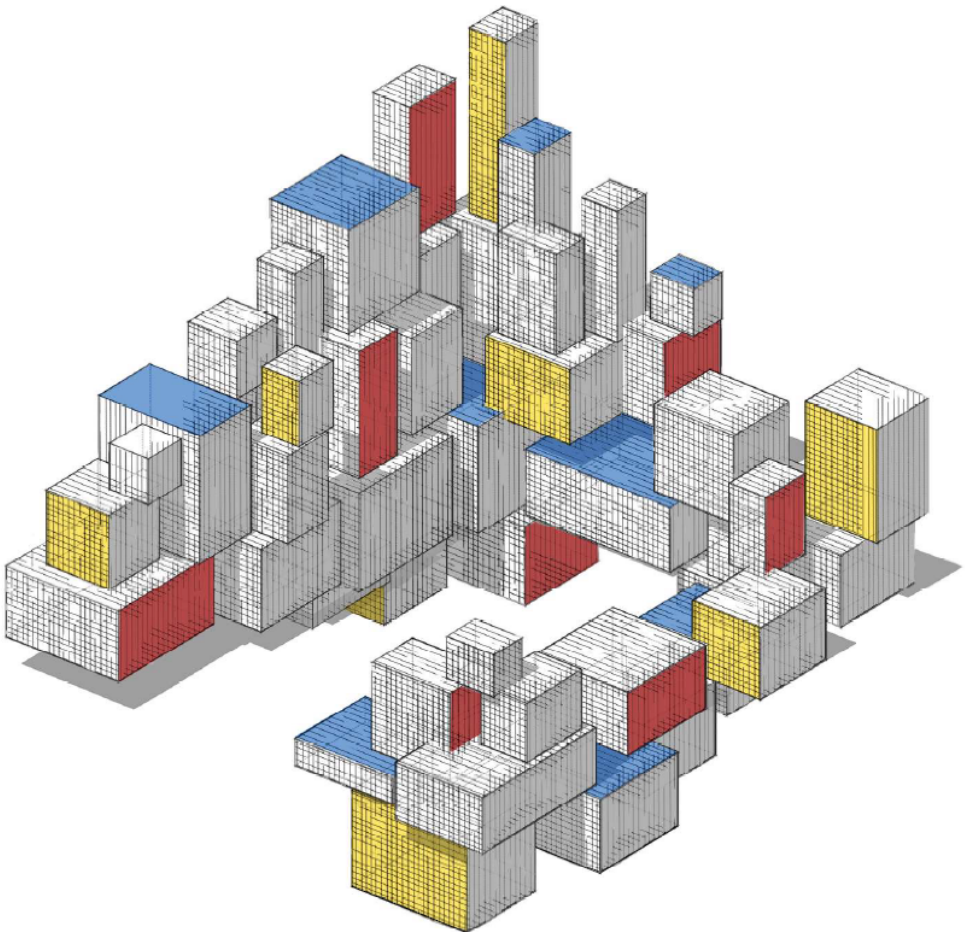
Simulating Random Variables



Purpose

In this lecture we discuss:

- The three main uses of Monte Carlo simulation:
  - Simulation of random objects and processes in order to observe their behavior
  - Estimation of numerical quantities by repeated sampling
  - Solving complicated optimization problems through randomized algorithms.
- How to generate random numbers
- How to simulate random variables from a prescribed distribution via:
  - The inverse-transform method
  - The accceptence–rejection method

Monte Carlo Simulation

Briefly put, Monte Carlo simulation is the generation of random data by means of a computer.

These data could arise from simple models, such as linear models or from very complicated models describing real-life systems, such as the positions of vehicles on a complex road network, or the evolution of security prices in the stock market.

In many cases, Monte Carlo simulation simply involves random sampling from certain probability distributions. The idea is to repeat the random experiment that is described by the model many times to obtain a large quantity of data that can be used to answer questions about the model.

Uses of MC

The three main uses of Monte Carlo simulation are:

- Sampling: Gather information about a random object by observing many realizations of it.
  - A random process that mimics the behavior of a telecommunications network.
  - Simulating from the posterior distribution in Bayesian statistics.
- Estimation: Compute certain numerical quantities via repeated sampling.
  - Multidimensional integrals
  - Expectations of complicated functions of random variables
- Optimization: Maximize/minimize complicated functions.
  - High-dimensional multimodal functions
  - Noisy functions

Generating Random Numbers

At the heart of any Monte Carlo method is a random number generator: a procedure that produces a stream of uniform random numbers on the interval (0,1).

Since such numbers are usually produced via deterministic algorithms, they are not truly random. However, for most applications all that is required is that such pseudo-random numbers are statistically indistinguishable from genuine random numbers $U_1, U_2, \ldots$ that are uniformly distributed on the interval (0,1) and are independent of each other; we write

$$U_1, U_2, \ldots \sim_{\text{iid}} \mathcal{U}(0,1).$$

For example, in Python the `rand` method of the `numpy.random` module is widely used for this purpose.

Multiple-recursive Generator

Most random number generators at present are based on linear recurrence relations. One of the most important random number generators is the multiple-recursive generator (MRG) of *order* k, which generates a sequence of integers $X_k, X_{k+1}, \ldots$ via the linear recurrence

$$X_t = (a_1 X_{t-1} + \cdots + a_k X_{t-k}) \bmod m, \quad t = k, k+1, \ldots$$

for some modulus m and multipliers $\{a_i, i = 1, \ldots, k\}$. Here "mod" refers to the modulo operation: $n \bmod m$ is the remainder when n is divided by m. The recurrence is initialized by specifying k "seeds", $X_0, \ldots, X_{k-1}$. To yield fast algorithms, all but a few of the multipliers should be 0. When m is a large integer, one can obtain a stream of pseudo-random numbers $U_k, U_{k+1}, \ldots$ between 0 and 1 from the sequence $X_k, X_{k+1}, \ldots$, simply by setting $U_t = X_t/m$.

Other Generators

It is also possible to set a small modulus, in particular $m = 2$. The output function for such modulo 2 generators is then typically of the form

$$U_t = \sum_{i=1}^{w} X_{tw+i-1} 2^{-i}$$

for some $w \leqslant k$, e.g., $w = 32$ or 64.

Examples are the feedback shift register generators, the most popular of which are the Mersenne twisters.

MRGs with excellent statistical properties can be implemented efficiently by combining several simpler MRGs and carefully choosing their respective moduli and multipliers. One of the most successful is L'Ecuyer's `MRG32k3a` generator.

Simulating Random Variables

Simulating a random variable X from an arbitrary (that is, not necessarily uniform) distribution invariably involves the following two steps:

1. Simulate uniform random numbers $U_1, \ldots, U_k$ on $(0, 1)$ for some $k = 1, 2, \ldots$.
2. Return $X = g(U_1, \ldots, U_k)$, where g is some real-valued function.

The construction of suitable functions g is as much of an art as a science.

Many simulation methods may be found, for example, in *The Handbook of Monte Carlo Methods* and the accompanying website `www.montecarlohandbook.org`.

Before we discuss two main general techniques for simulating random variables, we show one possible way to simulate standard normal random variables.

Simulating Standard Normal Random Variables

In Python we can generate standard normal random variables via the **`randn`** method of the **`numpy.random`** module.

If X and Y are independent standard normally distributed random variables (that is, $X, Y \sim_{\text{iid}} \mathcal{N}(0,1)$), then their joint pdf is

$$f(x,y) = \frac{1}{2\pi} \mathrm{e}^{-\frac{1}{2}(x^2+y^2)}, \quad (x,y) \in \mathbb{R}^2,$$

which is a radially symmetric function.

If we write, in polar coordinates,

$$X = R\cos(\Theta), \quad Y = R\cos(\Theta).$$

then (exercise!) the angle Θ is $\mathcal{U}(0, 2\pi)$ distributed (by symmetry) and the radius R has pdf $f_R(r) = r\,\mathrm{e}^{-r^2/2}, r > 0$. Moreover, R and Θ are independent.

Box-Muller Method

Note that R has the same distribution as $\sqrt{-2\ln U}$ with $U \sim \mathcal{U}(0,1)$.

So, to simulate $X, Y \sim_{\text{iid}} \mathcal{N}(0,1)$, the idea is to first simulate R and Θ independently and then return $X = R\cos(\Theta)$ and $Y = R\sin(\Theta)$ as a pair of independent standard normal random variables.

This leads to the Box–Muller approach for generating standard normal random variables.

Algorithm 1 Box–Muller

Output : $X, Y \sim_{\text{iid}} \mathcal{N}(0,1)$

1 Simulate $U_1, U_2 \overset{\text{iid}}{\sim} \mathcal{U}(0,1)$.

2 $X \leftarrow (-2\ln U_1)^{1/2}\cos(2\pi U_2)$

3 $Y \leftarrow (-2\ln U_1)^{1/2}\sin(2\pi U_2)$

4 **return** X, Y

Simulating Multivariate Normal Random Vectors

Once a $\mathcal{N}(0,1)$ generator is available, simulation of $\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ is straightforward.

First, find an $n \times n$ matrix $\mathbf{B}$ that decomposes $\boldsymbol{\Sigma}$ into the matrix product $\mathbf{B}\mathbf{B}^\top$, e.g., the Cholesky decomposition. In Python, the function `cholesky` of `numpy.linalg` can be used.

Next, use the fact that $\boldsymbol{\mu} + \mathbf{B}\boldsymbol{Z}$ where $\boldsymbol{Z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}_n)$.

Algorithm 2: Normal Random Vector Simulation

input: $\boldsymbol{\mu}, \boldsymbol{\Sigma}$

output : $\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$

1 Determine the Cholesky factorization $\boldsymbol{\Sigma} = \mathbf{B}\mathbf{B}^\top$.

2 Simulate $\boldsymbol{Z} = [Z_1, \ldots, Z_n]^\top$ by drawing $Z_1, \ldots, Z_n \sim_{\text{iid}} \mathcal{N}(0,1)$.

3 $\boldsymbol{X} \leftarrow \boldsymbol{\mu} + \mathbf{B}\boldsymbol{Z}$

4 **return** $\boldsymbol{X}$

Simulating from a Bivariate Normal Distribution

The Python code below draws $N = 1000$ iid samples from two bivariate normal pdfs. The resulting point clouds are given in the resulting Figure.

bvnormal.py

```
import numpy as np
from numpy.random import randn
import matplotlib.pyplot as plt

N = 1000
r = 0.0   #change to 0.8 for other plot
Sigma = np.array([[1, r], [r, 1]])
B = np.linalg.cholesky(Sigma)
x = B @ randn(2,N)
plt.scatter([x[0,:]],[x[1,:]], alpha =0.4, s = 4)
```

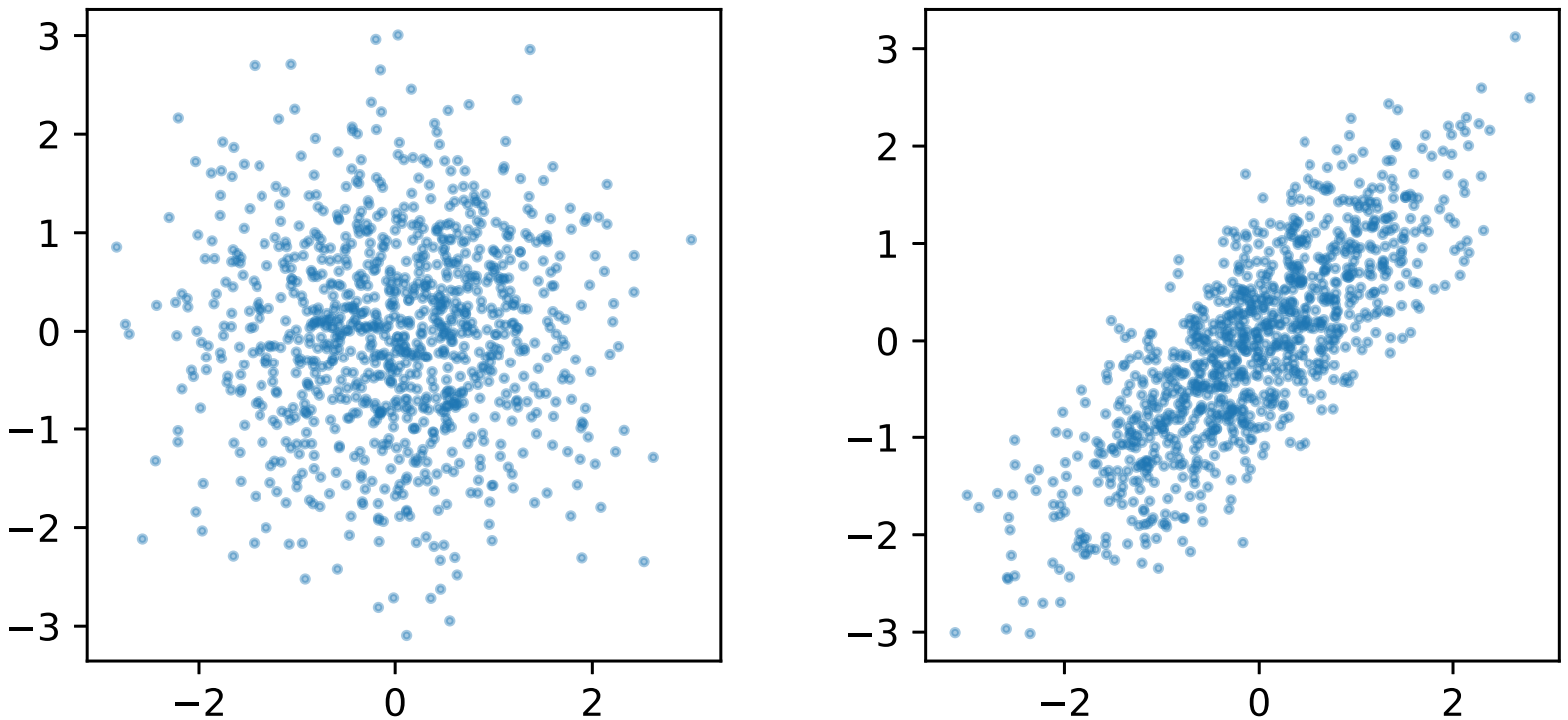


Figure: 1000 realizations of bivariate normal distributions with means zero, variances 1, and correlation coefficients 0 (left) and 0.8 (right).

Inverse-Transform Method

Let X be a random variable with cumulative distribution function (cdf) F and its inverse F^{-1}. Let $U \sim \mathcal{U}(0,1)$. Then,

$$\mathbb{P}[F^{-1}(U) \leqslant x] = \mathbb{P}[U \leqslant F(x)] = F(x).$$

This leads to the following method to simulate a random variable X with cdf F:

Algorithm 3: Inverse-Transform Method

input: Cumulative distribution function F.

output : Random variable X distributed according to F.

1 Generate U from $\mathcal{U}(0,1)$.

2 $X \leftarrow F^{-1}(U)$

3 **return** X

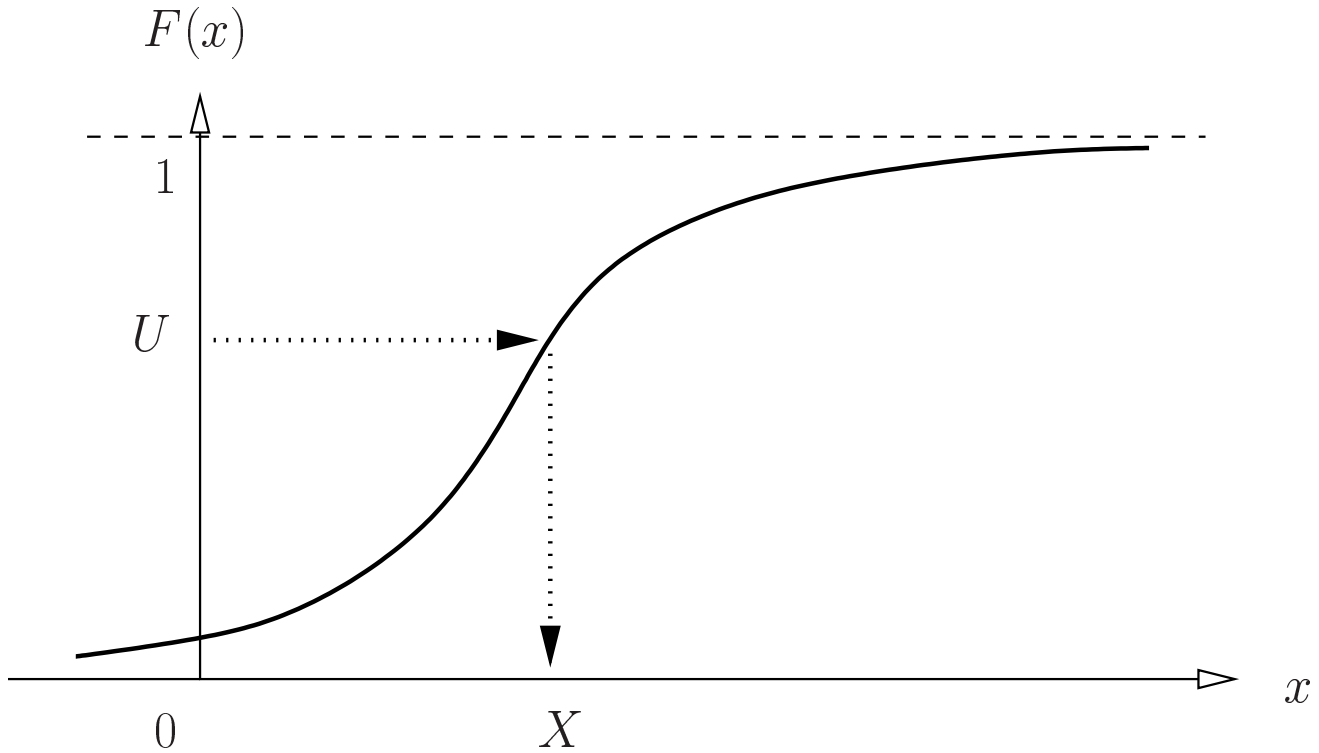


Figure: The inverse-transform method.

The inverse-transform method works both for continuous and discrete distributions.

Example

One remaining issue in our discussion of the Box-Muller method was how to simulate the radius R when we only know its density $f_R(r) = r\,\mathrm{e}^{-r^2/2}, r > 0$. We can use the inverse-transform method for this, but first we need to determine its cdf. The cdf of R is, by integration of the pdf,

$$F_R(r) = 1 - \mathrm{e}^{-\frac{1}{2}r^2}, \quad r > 0,$$

and its inverse is found by solving $u = F_R(r)$ in terms of r, giving

$$F_R^{-1}(u) = \sqrt{-2\ln(1-u)}, \quad u \in (0,1).$$

Thus R has the same distribution as $\sqrt{-2\ln(1-U)}$, with $U \sim \mathcal{U}(0,1)$. Since $1-U$ also has a $\mathcal{U}(0,1)$ distribution, R has also the same distribution as $\sqrt{-2\ln U}$.

Acceptance–Rejection Method

The acceptance–rejection method is used to sample from a "difficult" probability density function (pdf) $f(x)$ by generating instead from an "easy" pdf $g(x)$ satisfying $f(x) \leqslant C\, g(x)$ for some constant $C \geqslant 1$ (for example, via the inverse-transform method), and then accepting or rejecting the drawn sample with a certain probability.

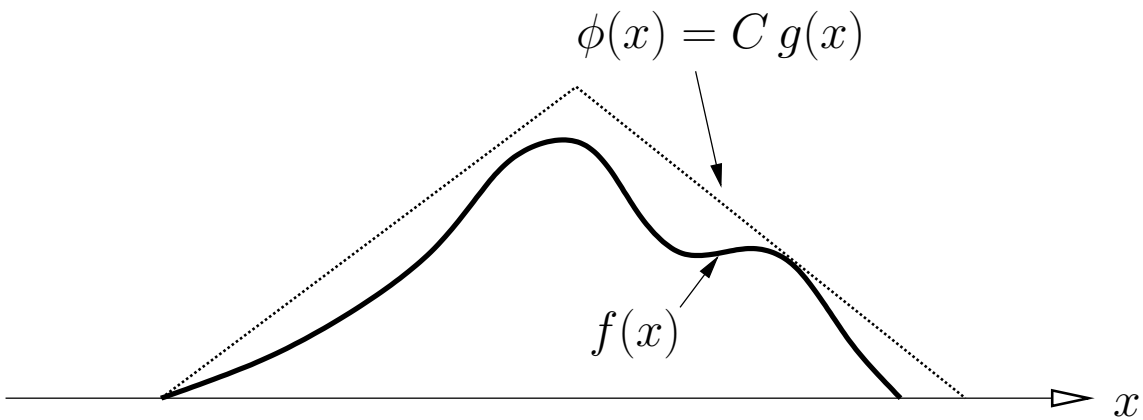


Figure: The acceptance-rejection method with a majorizing function.

Algorithm 4: Acceptance–Rejection Method

input: Pdf g and constant C such that $Cg(x) \geqslant f(x)$ for all x.
output : Random variable X distributed according to pdf f.

```
found ← false
while not found do
    Generate X from g.
    Generate U from 𝒰(0, 1) independently of X.
    Y ← UCg(X)
    if Y ⩽ f(X) then found ← true
return X
```

- Generate uniformly a point (X, Y) under the graph of the function Cg, by first drawing $X \sim g$ and then $Y \sim \mathcal{U}(0, Cg(X))$.
- If this point lies under the graph of f, then we accept X as a sample from f; otherwise, we try again.

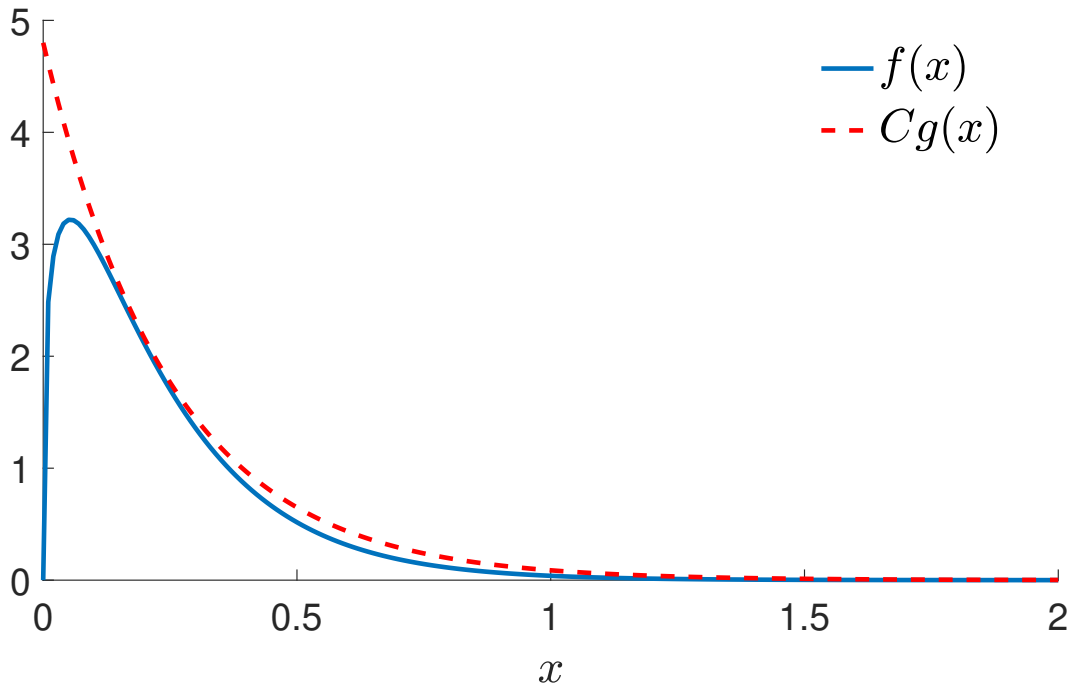
The efficiency of the acceptance–rejection method is usually expressed in terms of the probability of acceptance, which is $1/C$.

Simulating Gamma Random Variables

Consider, the Gamma($\alpha = 1.3, \lambda = 5.6$) distribution. Its pdf,

$$f(x) = \frac{\lambda^\alpha x^{\alpha-1} \mathrm{e}^{-\lambda x}}{\Gamma(\alpha)}, \quad x \geqslant 0,$$

where Γ is the gamma function $\Gamma(\alpha) := \int_0^\infty \mathrm{e}^{-x} x^{\alpha-1}\, \mathrm{d}x$, $\alpha > 0$, is dominated by the pdf $g(x) = 4\mathrm{e}^{-4x}, x \geqslant 0$ of the Exp(4) distribution multiplied by $C = 1.2$.



Hence, we can simulate from this particular Gamma distribution by accepting or rejecting a sample from the Exp(4) distribution.

accrejgamma.py

```
from math import exp, gamma, log
from numpy.random import rand

alpha = 1.3
lam = 5.6
f = lambda x: lam**alpha * x**(alpha-1) * exp(-lam*x)/gamma(alpha)
g = lambda x: lam*exp(-lam*x)
C = 1.2

found = False
while not found:
    x = - log(rand())/lam
    if C*g(x)*rand() <= f(x):
        found = True

print(x)
```