Computer Modeling and Simulation

Lectures 25-27

Monte Carlo Simulation

- *Monte Carlo* is the gambling locale in the country of Monaco. It is the home of the famous *Le Grand Casino* as well as many other gambling resorts.
- Monte Carlo simulation is about a concept that underlies gambling, that is, **probability**, hence, its association and designation with the well - known gambling region.
- Gambling casinos rely on probability to ensure, over the long run, that they are profitable.
    - For this to happen, the odds or chance of the casino winning has to be in its favor.
    - This is where probability comes into play because *the theory of probability provides a mathematical way to set the rules for each one of its games to make sure the odds are in its favor* .
- As a simulation technique, Monte Carlo simulation relies very heavily on probability

Monte Carlo Simulation – Coin Tossing example

- Consider you have a coin which you toss four times and you want to find the probability of having 3 heads and 1 tail.
- Using combinatorics, we’ll find the probability in the following way
- P(3 heads) =
-

Monte Carlo Simulation – Coin Tossing example

- Simulation is repeatedly performing an experiment.
- In the example, being discussed, one experiment will consist of four coin tosses and then you repeatedly perform this experiment to see how many times of the total experiments, you get three heads and one tail.
- The more the number of experiments is, the closer will be this probability, determined by repeated experiments (simulation), to the actual probability (determined using combinatorics).
-

Monte Carlo Simulation – Python program for Coin tossing example

```
from random import randint
successes = 0
attempts = 10000
for i in range(attempts):
        if randint(0,1)+randint(0,1)+randint(0,1)+randint(0,1) == 3:
                successes += 1
print(“No. Of attempts is”, = attempts)
print(“No. Of successes is”., = successes)
```

Monte Carlo Simulation – More difficult problems

- Consider the game of war.
- Description of war game. https://www.youtube.com/watch?v=yX-jOVer758
- How long will the game go i.e. on average, how many rounds will a game of war contain?
- The solution to this problem using combinatorics will be very difficult as we have to consider all the combinations of a cards in each deck as the entire game depends on that combination.
- Using monte carlo

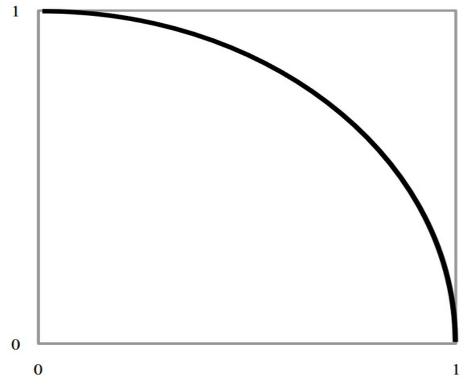
History of Monte Carlo Simulation

- The Monte Carlo simulation was invented in the way we just saw in the example of cards.
- The technique was first developed by Stanislaw Ulam, a mathematician who worked on the Manhattan Project.
- After the war, while recovering from brain surgery, Ulam entertained himself by playing countless games of solitaire.
- He became interested in plotting the outcome of each of these games in order to observe their distribution and determine the probability of winning.
- After he shared his idea with John Von Neumann (the brain behind the stored program computer), the two collaborated to develop the Monte Carlo simulation.
- While developing the nuclear weapons, scientists knew the behavior of one neutron, but they did not have a formula for how a system of neutrons would behave.
- Although they needed to understand such behavior to construct dampers and shields for the atomic bomb, experimentation was too time consuming and dangerous.
- John von Neumann and Stanislaus Ulam developed the technique of Monte Carlo simulation to solve the problem.

￼

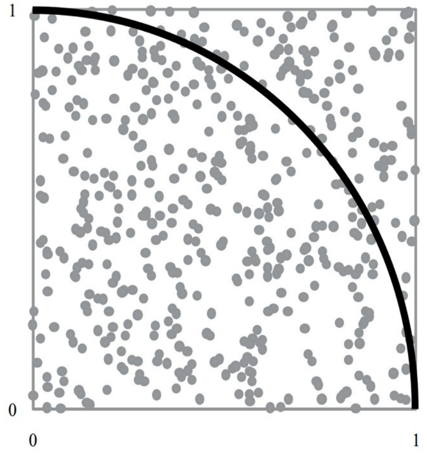
Calculating value of pi using Monte Carlo simulation

- Recall that the value of π is the ratio of a circle ’ s circumference to its diameter.
- To calculate this value, we can set up a Monte Carlo simulation that employs a geometric representation of the circle.
- Step 1: To start, draw a unit circle arc, that is, an arc of radius one circumscribed by a square as shown in the figure.



Calculating value of pi using Monte Carlo simulation

- Step 2: Then, randomly choose an x and y coordinate inside the square, and place a dot at that location.



Calculating value of pi using Monte Carlo simulation

- **Step 3**: Repeat step 2 at a given number of times.
- **Step 4:** Count the total number of dots inside the square and the number of dots inside the quarter circle.
- **Step 5**: With a large number of dots generated, these values will approximate the area of the circle and the area of the square.

From mathematics, this result can be represented as

$$\frac{\#\ \text{of dots inside circle}}{\#\ \text{of dots inside square}} = \frac{\frac{1}{4}\pi r^2}{r^2} = \frac{1}{4}\pi.$$

Definition of Monte Carlo Simulation

- Monte Carlo Simulation is this concept of repeated random samples of model input variables over many simulation runs .
- When setting up a Monte Carlo simulation or employing the Monte Carlo Method, one follows a four - step process.
- **Step 1** Define a distribution of possible inputs for each input random variable.
- **Step 2** Generate inputs randomly from those distributions.
- **Step 3** Perform a deterministic computation using that set of inputs.
- **Step 4** Aggregate the results of the individual computations into the final result.

Step 1:

- **Step 1** Define a distribution of possible inputs for each input random variable.
- Random variable is a variable whose values depend on outcomes of a random phenomenon.
- For example: The outcome of a coin toss is a random variable with possible values of Head or Tail.

- The **distribution** of these random numbers is a description of the portion of times each possible outcome or each possible range of outcomes occurs on the average over a great many trials.
- Different distributions:
    - **Uniform distribution** Suppose a specified range is partitioned into intervals of the same length. With a uniform distribution, the generator is just as likely to return a value in any of the intervals.
    - Normal distribution
    - Exponential Distribution
    - Triangular Distribution etc.

Pseudorandom Numbers

- **Pseudorandom** numbers are generated by computers.
- They are not truly random, because when a computer is functioning correctly, nothing it does is random.
- Computers

Applications of Pseudorandom Numbers

- Pseudorandom numbers are essential to many computer applications, such as games and security. In games, random numbers provide unpredictable elements the player can respond to, such as dodging a random bullet or drawing a card from the top of a deck.
- In computer security, pseudorandomness is important in encryption algorithms, which create codes that must not be predicted or guessed.

Pseudorandom number generator

- A **pseudorandom number generator**, or **PRNG**, is any program, or function, which uses math to simulate randomness. It may also be called a **DRNG** (digital random number generator) or **DRBG** (deterministic random bit generator).
- The math can sometimes be complex, but in general, using a PRNG requires only two steps:
  - Provide the PRNG with an arbitrary seed.
  - Ask for the next random number.
- The seed value is a "starting point" for creating random numbers. The seed value is used when computing the numbers.
- If the seed value changes, the generated numbers also change, and a single seed value always produce the same numbers. For this reason, the numbers aren't really random, because true randomness could never be re-created.
- The current time is often used as a unique seed value. For instance, if it's March 5, 2018, at 5:03 P.M. and 7.01324 seconds UTC, that can be expressed as an integer. That precise time never occur again, so a PRNG with that seed should produce a unique set of random numbers.

Linear Congruential Method – A PRNG

- In 1949, D. J. Lehmer presented one of the best techniques for generating uniformly distributed pseudorandom numbers, the **linear congruential method**.
- One simple linear congruential random number generator that generates values between 0 and 10, inclusive, is as follows:

$$r_0 = 10$$

$$r_n = (7r_{n-1} + 1) \bmod 11, \text{ for } n > 0$$

- The initial value in the sequence of random numbers, $r_0 = 10$, is the **seed**. The **mod** function returns the remainder.
- Thus, substituting $r_0 = 10$ on the right-hand side of the second line of the definition, the **generating function**, we calculate $r_1 = (7 \cdot 10 + 1) \bmod 11 = 5$.
- After we calculate one "random number," to evaluate the next, we substitute that value into the expression on the right-hand side. Consequently, the next random number is $r_2 = (7 \cdot \mathbf{5}$

Linear Congruential Method

- The general form for the **linear congruential method** to generate pseudorandom integers from 0 up to, but not including, *modulus* is as follows:

$$r_0 = seed$$

$$r_n = (multiplier\ r_{n-1} + increment) \bmod modulus, \text{ for } n > 0$$

- where *seed*, *modulus*, and *multiplier* are positive integers and *increment* is a nonnegative

Random Floating-point Number generation

- If we desire floating-point numbers between 0 and 1, we divide each number in the sequence by the **modulus.**
- For this computation, the smallest possible pseudorandom floating-point number is 0.0 and the largest is (*modulus* – 1)/*modulus*. Thus, floating-point numbers that we generate by dividing by the modulus are in the interval [0.0, 1.0), or the interval between

Possible values for multiplier and modulus

- Much research has been done to discover choices for *multiplier* and *modulus* that give the largest possible sequence that appears random.
- For built-in random number generators, *modulus* is often the largest integer a computer can store, such as $2^{31} – 1$ = 2,147,483,647 on some machines. For this modulus, a multiplier of 16,807 and an increment of 0 produce a sequence of 2^{31}

Different Ranges of Random Numbers

- The linear congruential method generated a random integer from 0 up to the modulus, where by up to we mean not including the modulus.
- We can obtain a floating-point counterpart with value from 0.0 up to 1.0 by dividing by the modulus.
- We can obtain uniformly distributed integer or real random numbers in any range.
- Suppose that *rand* is a uniformly distributed random floating point number from 0.0 up to 1.0. Suppose, however, that we need a random floating point number from 0.0 up to 5.0. Because the length of this interval is 5.0, we multiply *rand*

Random numbers within any range

- Mathematically, we have the following:

$$0.0 \le rand < 1.0$$

- Thus, multiplying by 5.0 throughout, we obtain the correct interval, as shown:

$$0.0 \le 5.0\ rand < 5.0$$

- If the lower bound of the range is different from 0, we add that bound. For example, if we need a random floating-point number from 2.0 up to 7.0, we multiply by the length of the interval, 7.0 – 2.0 = 5.0, to expand the range. Then, we add the lower bound, 2.0, to shift, or translate, the result so that the following inequalities hold:

$$2.0 \le (7.0 - 2.0)\ rand + 2.0 < 7.0$$

or

$$2.0 \le 5.0\ rand$$

Specifying Random Floating-Point Numbers in Other Ranges

- If *rand* is a random floating-point number such that 0.0 ≤ *rand* < 1.0, then ***(max – min)rand + min*** is a random floating-point number from *min* up to *max* that satisfies the following inequality:

$$min \le (\boldsymbol{max - min})\ \boldsymbol{rand + min} < max$$

Random Integer numbers upto the modulus

- Frequently, we need a more-restricted range of random integers than from 0 up to *modulus.*
- For example, a simulation might require random integer temperatures between 0 and 99, inclusive.
- One method of restricting the range is to multiply a floating-point random number between 0.0 and 1.0 by 100 (the number of integers from 0 through 99, or 99 + 1) and then return the **integer part** (the number before the decimal point). For example, suppose *rand* is 0.692871. Multiplying by 100, we obtain 100 · 0.692871 = 69.2871. Truncating, we obtain an integer (69) between 0 and

Random Integer numbers with lower bound other than 0

- Sometimes we want the range of random integers to have a lower bound other than 0, for example, from 100 to 500, inclusive. Because we include 100 and 500 as options, the number of integers from 100 to 500 is one more than the difference in these values, (500 – 100 + 1) = 401.
- As with the last example, we multiply this value by *rand* to expand the range. Then, we add the lower bound, 100, to the product to translate the range to start at 100 as follows:

$$100.0 \le 401\,rand$$

Random Integer numbers upto the modulus with lower bound other than 0

- Finally, we take the integer part of the result, which we write here as applying a function INT.

$$100 \leq \text{INT}(401\textit{rand} + 100) < 501$$

or

$$100 \leq \text{INT}(401\textit{rand} + 100) \leq 500$$

- Because the floating-point numbers (401*rand* + 100) are less than 501.0, after truncation, the

Specifying Random Integers in Other Ranges

- If *rand* is a random floating-point number such that $0.0 \leq rand < 1.0$, then $\mathrm{INT}((max - min + 1)rand + min)$ is a random integer from *min* to *max*, inclusive, that satisfies the following inequality:

$$min \leq \mathbf{INT(\ (\mathit{max} - \mathit{min} + 1)\cdot \mathit{rand} + \mathit{min})} \leq max$$

-

Random Numbers from Various Distributions

- A Monte Carlo simulation requires the use of unbiased random numbers.
- The **distribution** of these numbers is a description of the portion of times each possible outcome or each possible range of outcomes occurs on the average over a great many trials.
- However, the distribution that a simulation requires depends on the problem. There are algorithms for generating random numbers from

Uniform Distribution

- Suppose a specified range is partitioned into intervals of the same length. With a **uniform distribution**, the generator is just as likely to return a value in any of the intervals.
- Equivalently, in a list of many such random numbers, on the average each interval contains the same number

Uniform Distribution

- For example, figure on the left presents a histogram with 10 intervals of length 0.1 of a table of 10,000 random floating-point numbers, uniformly distributed from 0.0 up to 1.0.
- As expected, approximately one-tenth of the 10,000, or 1000, numbers appears in each subdivision. Thus, the curve across the tops of the bars is virtually a horizontal line of height 1000.

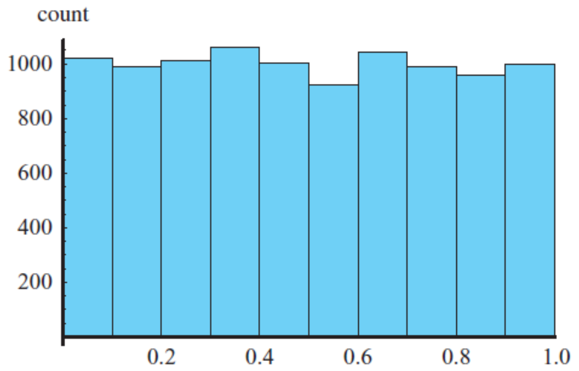


Fig. 3: Histogram of 10,000 random floating-point numbers, uniformly distributed from 0.0 up to 1.0

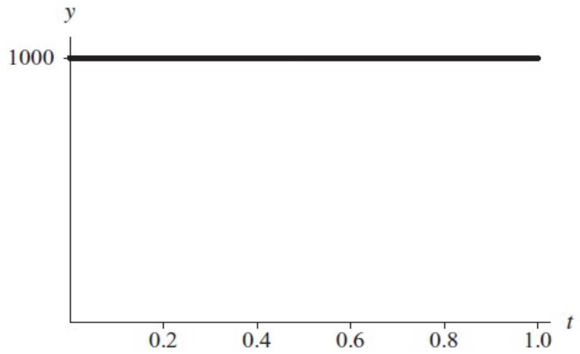


Fig. 4: Horizontal line at height 1000 approximately goes across the top of the histogram

Discrete and Continuous Distribution

- A distribution can be **discrete** or **continuous**.
- A **discrete distribution** is a distribution with discrete values. A **continuous distribution** is a distribution with continuous values.
- Discrete data can only take certain values.
- Continuous data can take any value (within a range).
- To illustrate the difference between the terms discrete and continuous, a digital clock shows time in a discrete manner, from one minute to the next, while a clock with two hands indicates time in a continuous, unbroken

Probability Density Function

- For a discrete distribution, a **probability function** (or **density function**, or **probability density function**) returns the probability of occurrence of a particular argument value.
- For example, *P*(1382) might be the probability that the random number generator returns 1382.
- However, if a distribution is continuous, the probability of occurrence of any particular value is zero. Thus, for a continuous distribution, a probability function (or density function, or probability density function) indicates the probability that a given outcome falls inside a specific range of

Probability Density Function

- The integral of the probability function from the lower to the upper bound of the range, which is the area under that portion of the curve, gives the probability that the outcome is in that range.
- For example, the probability that the random velocity in the *x*-direction of a dust particle is between 3.0 and 4.0 mm/s is the integral of the probability density function from 3.0 to 4.0.
- Figure presents a horizontal line of height 1 that is the graph of the probability density function ($P(x) = 1$) for uniformly generated random numbers with values from 0.0 up to 1.0.
- The probability that a uniform random floating-point number between 0.0 and 1.0 falls between 0.6 and 0.8 is the integral of the function $f(x) = 1$ from 0.6 to 0.8. Thus, the probability is the area of the shaded region between 0.6 and 0.8, which is (0.8 – 0.6) (1.0) = 0.2. Such a random number is between 0.6 and 0.8 for 0.2 = 20% of the time.

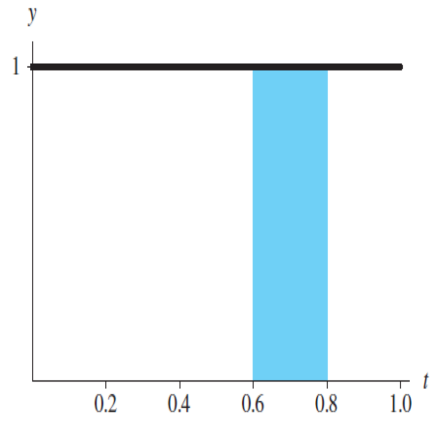


Fig. 5: Probability density function for the distribution with histogram in Fig. 3

Discrete Distributions with equal probability of all events

- If an equal likelihood of each of several discrete events exists, in a simulation we can generate a random integer to indicate the choice.
- For example, in a simulation of a pollen grain moving in a fluid, suppose at the next time step the grain is just as likely to move in any direction—north, east, south, west, up, or down—in a three-dimensional (3D) grid.
- A probability of 1/6 exists for the grain to move in any of the six directions.
- With these equal probabilities, we can generate a uniformly distributed integer between 1 and 6 to indicate the direction of movement.
- Thus to simulate discrete distributions where every event has equal chance to exist, generate a uniform random integer from a sequence of *n* integers, where each integer corresponds to an event.

Discrete Distributions with varying probability of events

- Frequently, however, the discrete choices do not carry equal probabilities.
- For example, in an initial 3D grid, suppose only 15% of the grid sites, or cells, contain pollen grains. Thus, a *probPollen* = 15% = 0.15 chance exists for a cell to contain a grain.
- If the location is to contain a pollen grain, we make the cell's value equal to *POLLEN* = 1; otherwise, the cell's value becomes *EMPTY* = 0.
- To initialize a grid for a simulation, we must designate for each cell if the location contains pollen or not.
- For each cell, we need to generate a uniformly distributed random floating-point number from 0.0 up to 1.0. On the average, 15% of the time this random number is less than 0.15, while 85% of the time the number is greater than or equal to 0.15.
- Thus, to initialize the cell, if the random number is less than 0.15, we make the cell's value *POLLEN*; otherwise, we assign *EMPTY* to the cell's value.

Discrete Distributions with varying probability of events

- Using the probabilities and cell values, employ the following logic is employed to initialize each cell in the grid:

```
if a random number is less than probPollen (i.e., pollen grain at site)
        set the cell’s value to POLLEN
else (i.e., no pollen grain at site)
        set the cell’s value to EMPTY
```

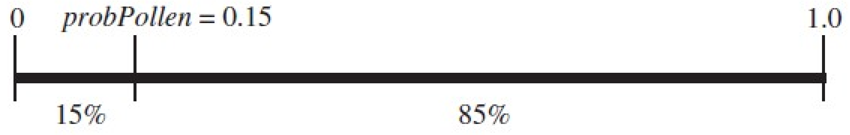
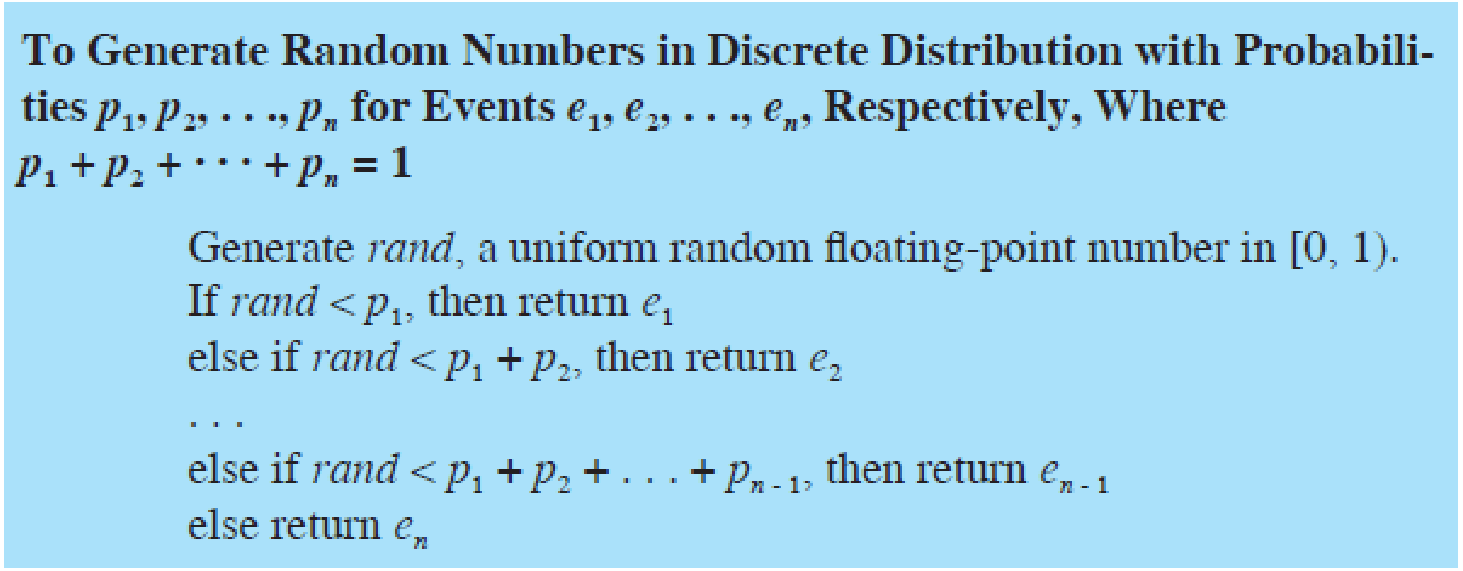


Fig: 15% of floating-point values between 0 and 1 are less than *probPollen* = 0.15

Discrete Simulations with varying probabilities



To Generate Random Numbers in Discrete Distribution with Probabilities $p_1, p_2, \ldots, p_n$ for Events $e_1, e_2, \ldots, e_n$, Respectively, Where $p_1 + p_2 + \cdots + p_n = 1$

```
Generate $rand$, a uniform random floating-point number in $[0, 1)$.
If $rand < p_1$, then return $e_1$
else if $rand < p_1 + p_2$, then return $e_2$
. . .
else if $rand < p_1 + p_2 + \ldots + p_{n-1}$, then return $e_{n-1}$
else return $e_n$
```

Normal Distribution

- A **normal,** or **Gaussian, distribution**, which statistics frequently employs, has a probability density function where *μ* is the mean and *σ* is the standard deviation of the form .

$$\frac{1}{\sqrt{2\pi\sigma}}e^{-(x-\mu)^2/(2\sigma)}$$

Normal Distribution

- Below figure displays a histogram of a set of 1000 random numbers in the Gaussian distribution with mean 0 and standard deviation 1.
- 68.3% of the values in a normal distribution are within $\pm\sigma$ of the mean, μ; 95.5% are within $\pm 2\sigma$ of μ; and 99.7% are within $\pm 3\sigma$ of μ.

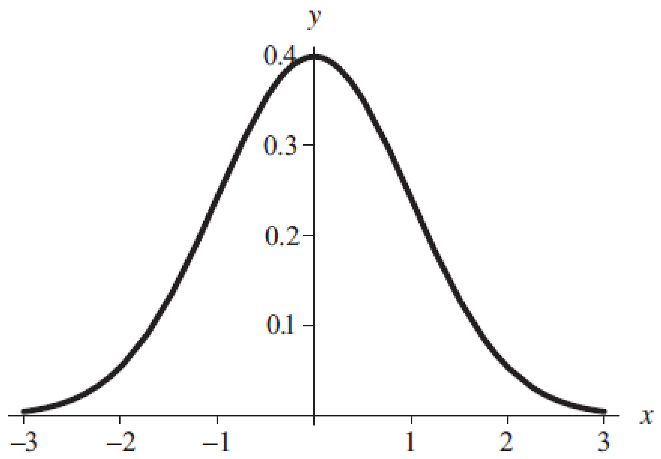


Fig: Probability density function for normal distribution with mean 0 and standard deviation 1

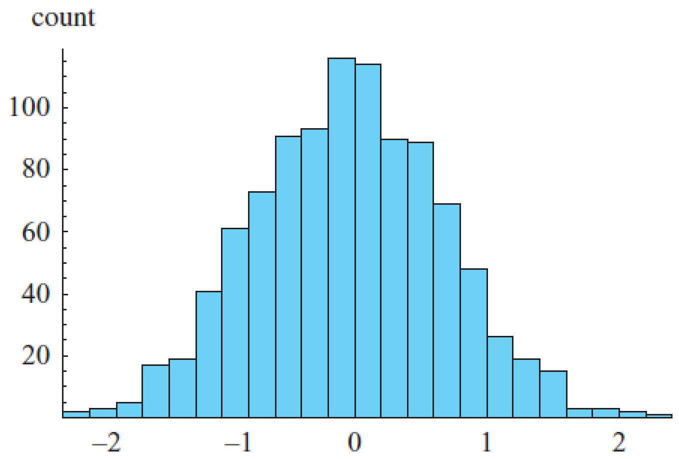


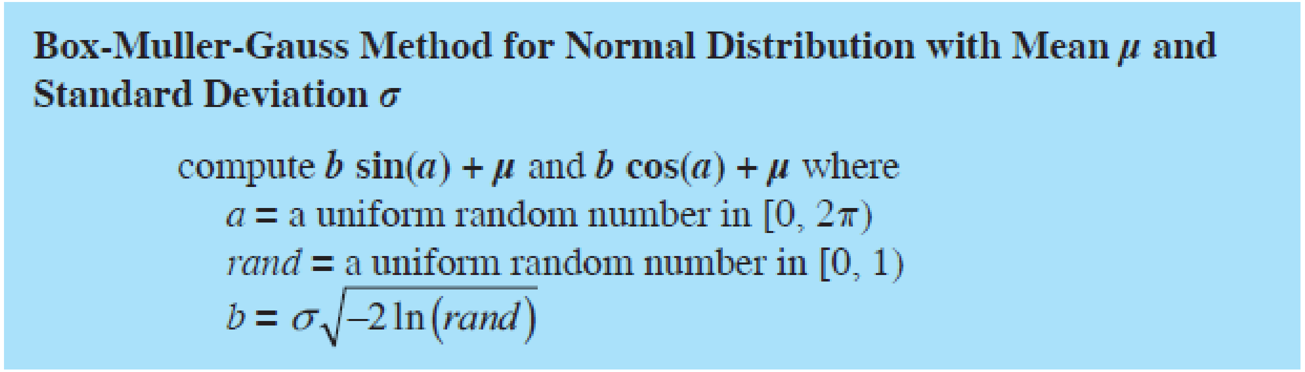
Fig: Histogram of a normal distribution with mean 0 and standard deviation 1

Generating Random Numbers in Normal Distribution

- The **Box-Muller-Gauss method** can be employed to generate numbers in a normal distribution.
- The method first generates a uniformly distributed random number, *a*, between 0 and 2π.
- Then, the technique computes *b*, the product of the standard deviation (*σ*) and the square root of the negative natural logarithm of a uniformly distributed random number between 0.0 and 1.0.

Box-Muller-Gauss method

- The two values $b \cdot \sin(a) + \mu$ and $b \cdot \cos(a) + \mu$ are normally distributed with mean μ and standard deviation σ.



Box-Muller-Gauss Method for Normal Distribution with Mean μ and Standard Deviation σ

compute $b \sin(a) + \mu$ and $b \cos(a) + \mu$ where

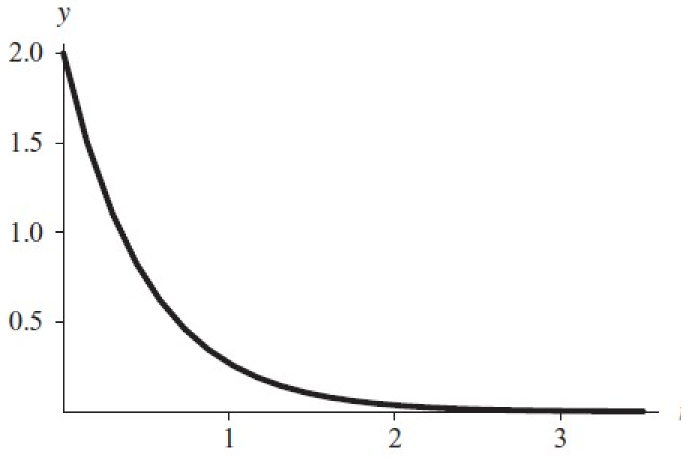
a = a uniform random number in $[0, 2\pi)$

$rand$ = a uniform random number in $[0, 1)$

$$b = \sigma\sqrt{-2\ln(rand)}$$

Exponential Distribution

- Functions of the form $\boldsymbol{f(t) = |r|e^{rt}}$ **with** $\boldsymbol{r < 0}$ **and** $\boldsymbol{t > 0}$**,** or $\boldsymbol{f(t) = |r|e^{rt}}$ **with** $\boldsymbol{r > 0}$ **and** $\boldsymbol{t < 0}$ are probability density functions in which the area under each curve is 1.
- Figure on the left contains the graph of a function in this category, $f(t) = 2e–2t$. Figure on the right displays a histogram of 1000 such exponentially distributed random



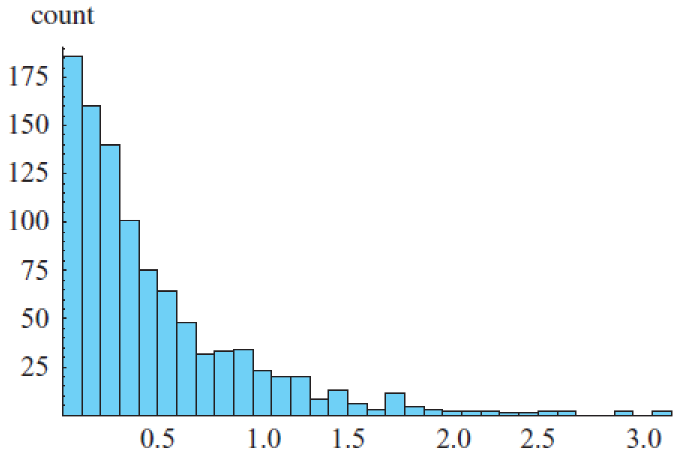


Fig. Histogram of 1000 random numbers ln(*rand*)/(–2), where *rand* is a uniformly generated random number in [0.0, 1.0)

Fig. Probability density function $f(t) = 2e^{-2t}$ for $t > 0$

Generating Random Numbers in Normal Distribution

- To obtain a number in such a distribution, the **exponential method** divides the natural logarithm of a uniformly distributed random number from 0.0 to 1.0 by the rate constant (r), that is, $\ln(\mathit{rand})/r$, where *rand* is random between 0 and 1.
- For example, to generate numbers in the distribution $f(t) = 2e^{-2t}$, we calculate $\ln(\mathit{rand})/(-2)$.

Generating Random Numbers in Exponential Distribution

- The same algorithm is employed to generate random numbers from minus infinity to 0 for probability density function $f(t) = |r|e^{rt} = re^{rt}$ with $r > 0$.
- Figure on the left shows the graph of one such function, $f(t) = 2e^{2t}$; and Figure on the right displays a histogram of 1000 pseudorandom numbers that the algorithm ln(*rand*)/2 generates.

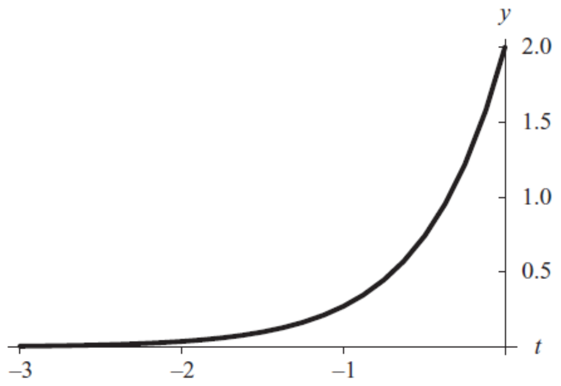


Fig: Probability density function $f(t) = 2e^{2t}$ for $t < 0$

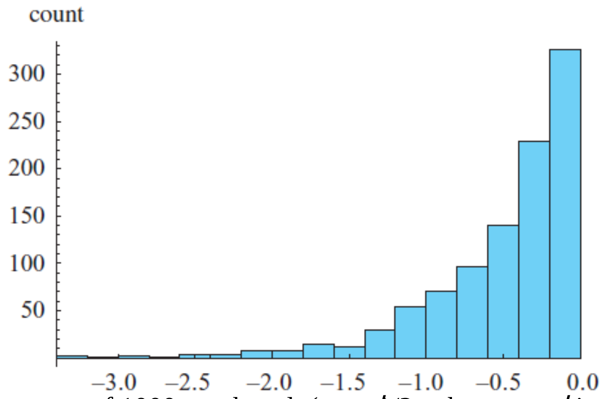
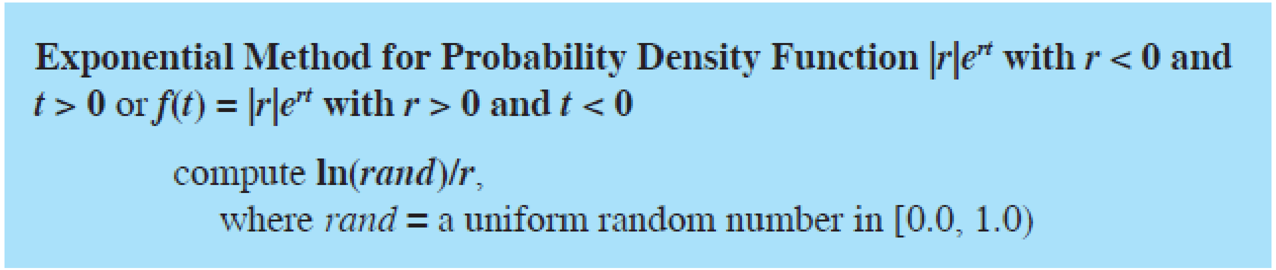


Fig: Histogram of 1000 numbers ln(*rand*)/2, where *rand* is a uniformly generat
random number in [0.0, 1.0)

Probability density function for Exponential Distribution



Exponential Method for Probability Density Function $|r|e^{rt}$ with $r < 0$ and $t > 0$ or $f(t) = |r|e^{rt}$ with $r > 0$ and $t < 0$

```
compute $\mathbf{ln}(rand)/r$,
    where $rand$ = a uniform random number in [0.0, 1.0)
```

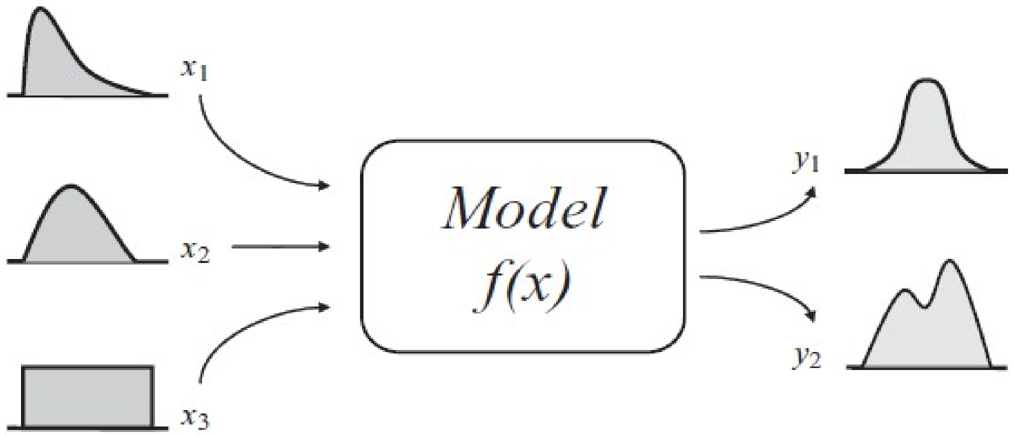
Step 1:

- If this distribution is known or sufficient data exist to derive it, then this step is straightforward.
- However, if the behavior of an input variable is not well understood, then the modeler might have to estimate this distribution based on empirical observation or subject matter expertise.
- The modeler may also use a uniform distribution if he or she is lacking any specific knowledge of the variable ’ s characteristics.
- When additional information is gathered to define the variable, then the uniform distribution can be replaced.

Step 2:

- **Step 2** Generate inputs randomly from those distributions.
- Step 2 requires randomly sampling each input variable ’ s distribution many times to develop a vector of inputs for each variable.
- Suppose we have two input random variables X and Z.
- After sampling n times, we have $X = (x_1, x_2, \ldots, x_n)$ and $Z = (z_1, z_2, \ldots, z_n)$.
- Elements from these vectors are then sequentially chosen as inputs to the function defining the model.
- The question of how large n should be is an important one because the number of samples determines the power of the output test statistic.
- As the number of samples increases, the standard deviation of the test statistic decreases.
- In other words, there is less variance in the output with larger sample sizes.
- However, the increase in power is not linear with the number of samples so there is a point when more sampling provides little improvement.

Basic Monte Carlo Model



Steps 3 and 4

- Step 3 is straightforward. It involves sequentially choosing elements from the randomly generated input vectors and computing the value of the output variable or variables until all n outputs are generated for each output variable.
- Step 4 involves aggregating all these outputs. Suppose we have one output variable Y. Then we would have as a result of step 4 an output vector Y = (y1,y2, … , yn).
- We can then perform a variety of statistical tests on Y to analyze this output.

Components in Monte Carlo Simulation

This four - step method requires having the necessary components in place to achieve the final result. These components may include:

(1) probability distribution functions (pdfs) for each random variable.

(2) a random number generator

(3) a sampling rule— a prescription for sampling from the pdfs

(4) scoring— a method for combining the results of each run into the final result

(5) error estimation— an estimate of the statistical error of the simulation

output as a function of the number of simulation runs and other

parameters.