Numerical Mathematics & Computing, 7 Ed.
§10 Monte Carlo Methods and Simulation

Ward Cheney & David Kincaid

Cengage§c

http://www.cengagebrain.com
http://www.ma.utexas.edu/CNA/NMC7

§10.1 Monte Carlo Methods

- **Monte Carlo methods** (or **Monte Carlo experiments**) are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results.
- Their essential idea is using randomness to solve problems that might be deterministic in principle.
- They are often used in physical and mathematical problems and are most useful when it is difficult or impossible to use other approaches.

A brief History

- An early variant of the Monte Carlo method can be seen in the Buffon's needle experiment, in which π can be estimated by dropping needles on a floor made of parallel and equidistant strips.

- In the 1930s, Enrico Fermi first experimented with the Monte Carlo method while studying neutron diffusion, but did not publish anything on it.

Enrico Fermi, PhD (1901–1954)	
Notable awards	•Matteucci Medal (1926) •**Nobel Prize** (1938) •Hughes Medal (1942) •Medal for Merit (1946) •Franklin Medal (1947) •ForMemRS (1950) •Barnard Medal for Meritorious Service to Science (1950) •Rumford Prize (1953) •Max Planck Medal (1954)

A brief History (cont.)

- The modern version of the Monte Carlo method was invented in the late 1940s by Stanislaw Ulam, while he was working on nuclear weapons projects at the Los Alamos National Laboratory.

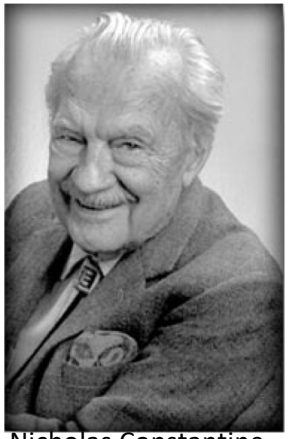
- Immediately after Ulam's breakthrough, John von Neumann understood its importance and programmed the ENIAC computer to carry out Monte Carlo calculations.

Stanisław Ulam (1909-1984)

John von Neumann, PhD (1903-1957)

A brief History (cont.)

- Being secret, the work of von Neumann and Ulam required a code name.
- A colleague of von Neumann and Ulam, Nicholas Metropolis, suggested using the name *Monte Carlo*, which refers to the Monte Carlo Casino in Monaco where Ulam's uncle would borrow money from relatives to gamble.
- Using lists of "truly random" random numbers was extremely slow, but von Neumann developed a way to calculate pseudorandom numbers.



Nicholas Constantine Metropolis, PhD (1915-1999)

A brief History (cont.)

- Monte Carlo methods were central to the simulations required for the Manhattan Project, though severely limited by the computational tools at the time.
- In the 1950s they were used at Los Alamos for early work relating to the development of the hydrogen bomb, and became popularized in the fields of physics, physical chemistry, and operations research.
- The Rand Corporation and the U.S. Air Force were two of the major organizations responsible for funding and disseminating information on Monte Carlo methods during this time, and they began to find a wide application in many different fields.

Applications

- Physical sciences
- Engineering
- Climate change and radiative forcing
- Computational biology
- Computer graphics
- Applied statistics
- Artificial intelligence for games
- Design and visuals
- Search and rescue
- Finance and business
- Law

https://en.wikipedia.org/wiki/Monte_Carlo_method

An Example Application of Monte Carlo Methods

- A highway engineer wishes to **simulate the flow of traffic** for a proposed design of a major freeway intersection.

- The information that is obtained will then be used to determine the capacity of **storage lanes** (in which cars must slow down to yield the right of way).

- By writing and running a **simulation program**, the engineer can study the effect of different speed limits, determine which flows lead to saturation (bottlenecks), and so on.

- Some techniques for constructing such programs are developed.

Overview of This Chapter

- Instead of addressing clear-cut mathematical problems, this chapter attempts to develop methods for **simulating complicated processes or phenomena**.

- Use computer to imitate an experiment or a process, then by repeating the computer simulation with different data, we can draw **statistical conclusions**.

- In such an approach, the conclusions may lack a high degree of mathematical precision but still be sufficiently accurate to enable us to understand the process being simulated.

Overview of This Chapter (cont.)

§10.1 Random Numbers

§10.2 Estimation of Areas and Volumes by Monte Carlo Techniques

§10.3 Simulation

§10.1 Random Numbers

- The computer simulation involves an element of **chance**.

- Since chance or randomness is part of the method, we begin with the elusive concept of **random numbers**.

Random Number Generation

- Historical methods of generating random numbers include:
  - Dice throwing.
  - Coin Flipping.
  - Shuffling of playing cards.

• But these are mechanical and generating large amounts of sufficiently random numbers would require a large amount of work and/or time.

• Alternatively results can be collected and passed out as random number tables.

The coin toss at the start of Super Bowl XLIII

Random Number Generation (cont.)

- With the advent of computers there are several techniques to quickly generate ‘random numbers’.
- However, computers are deterministic machines, and can not act in a random manner.
- Yet they can generate numbers in such a complex manner that, to all intents and purposes, the successive numbers are unpredictable with no discernible pattern.

What are Random Numbers?

- Consider a sequence of real numbers $x_1, x_2, \ldots$ all lying in the unit interval $(0, 1)$.
- Expressed informally, the sequence is **random** if the numbers seem to be distributed haphazardly throughout the interval and if there seems to be no pattern in the progression $x_1, x_2, \ldots$

Are the following Random Numbers?

- For example, if all the numbers in decimal form begin with the digit 3, then the numbers are clustered in the subinterval $0.3 \le x < 0.4$
- If the numbers are monotonically increasing
- If each x_i is obtained from its predecessor by a simple continuous function, say, $x_i = f(x_{i-1})$, then the sequence is …

Properties of random numbers

- A sequence of numbers is **uniformly distributed** in the interval (0, 1) if no subset of the interval contains more than its share of the numbers.

- In particular, the probability that an element x drawn from the sequence falls within the subinterval $[a, a + h]$ should be h and hence independent of the number a.

- Similarly, if $p_i = (x_i, y_i)$ are random points in the plane uniformly distributed in some rectangle, then the number of these points that fall inside a small square of area k should depend only on k and not on where the square is located inside the rectangle.

Properties of Random Numbers (cont.)

- Two important statistical properties:
  - **Uniformity**.
  - **Independence**.

- Random number R_i must be independently drawn from a uniform distribution

Pseudo-randomness

- Random numbers produced by a computer code cannot be truly random because the manner in which they are produced is completely **deterministic**; that is, no element of chance is actually present.

- But the sequences that are produced by these routines appear to be random, and they do pass certain tests for randomness.

- Some authors prefer to emphasize this point by calling such computer-generated sequences **pseudo-random** numbers.

Derrick Henry Lehmer, PhD (1905-1991), Professor, UC Berkeley.

"A random sequence is a vague notion . . . in which each term is unpredictable to the uninitiated and whose digits pass a certain number of tests traditional with statisticians . . . " - D. H. Lehmer, 1951

Random-Number Algorithms and Generators

- Most computer systems have **random-number generators**, which are procedures that produce either a single random number or an entire array of random numbers with each call.
- For example, random-number generators are contained in **mathematical software systems** such as MATLAB, Maple, and Mathematica as well as many computer programming languages.
- We call such a procedure **Random**.
- The reader can use a random-number generator available on his or her own computing system, one available within the computer language being used, or one of the generators described below.

Characteristics of a Good Generator

- Maximum Density:
  - Such that the values assumed by R_i, $i = 1, 2, \ldots$ have no large gaps on $[0, 1]$.
  - Problem: Instead of continuous, each R_i is discrete.
  - Solution: a very large integer for modulus m (approximation appears to be of little consequence).
- Maximum Period:
  - To achieve maximum density and avoid cycling.
  - Achieve by: proper choice of a, c, m and X_0.
- Most digital computers use a binary representation of numbers:
  - Speed and efficiency are aided by a modulus m to be (or close to) a power of 2.

Techniques for Generating Random Numbers

- Linear Congruential Method (LCM).
- Combined Linear Congruential Generators (CLGM).
- Random-Number Streams.

Linear Congruential Method

- To produce a sequence of integers $X_1, X_2, \ldots$ between 0 and m−1 by following a recursive relationship:

$$X_{i+1} = (aX_i + c) \bmod m, \; i = 0, 1, 2, \ldots$$

  a is the multiplier, c is the increment, and m is the mudulus.

- The selection of a, c, m and X_0 drastically affects the statistical properties of the cycle length.

- The random integers are being generated in [0, m − 1], so to convert the integers to random numbers:

$$R_i = X_i/m, \; i = 1,2,\ldots$$

Derrick Henry Lehmer, PhD (1905-1991), Professor, UC Berkeley.

Input / output of the generators

- The random-number procedures return one or an array of uniformly distributed pseudo-random numbers in the unit interval (0, 1) depending on whether the argument is a scalar variable or an array.

- A random seed procedure restarts or queries the pseudo-random-number generator.

Example LCM

- If the reader wishes to program a random-number generator, the following one should be satisfactory on a machine that has 32-bit word length.

- This algorithm **generates n random numbers $x_1, x_2, \ldots, x_n$ uniformly distributed in the open interval (0, 1)** by means of the following recursive algorithm:

integer array $(\ell_i)_{0:n}$; **real array** $(x_i)_{1:n}$
$\ell_0 \leftarrow$ any integer such that $1 < \ell_0 < 2^{31} - 1$
for $i = 1$ **to** n
    $\ell_i \leftarrow$ remainder when $7^5 \ell_{i-1}$ is divided by $2^{31} - 1$
    $x_i \leftarrow \ell_i / (2^{31} - 1)$

- Here, all I_i's are integers in the range $1 < I_i < 2^{31} - 1$.

- The initial integer I_0 is called the **seed** for the sequence and is selected as any integer between 1 and the Mersenne prime number $2^{31} - 1 = 2147483647$.

- An external function procedure to generate a new array of pseudo-random numbers per call could be based on the following pseudocode:

real procedure Random $((x_i))$
integer $seed, i, n$; **real array** $(x_i)_{1:n}$
integer $k \leftarrow 16807$, $j \leftarrow 2147483647$
$seed \leftarrow$ select initial value for $seed$
$n \leftarrow size((x_i))$
for $i = 1$ **to** n
    $seed \leftarrow \mod(k \cdot seed, j)$
    $x_i \leftarrow \text{real}(seed)/\text{real}(j)$
end procedure Random

- To allow adequate representation of the numbers involved in procedure *Random*, it must be written by using double or extended precision for use on a 32-bit computer; otherwise, it will produce nonrandom numbers.

- Recall that here and elsewhere, mod (n, m) is the remainder when n is divided by m; that is, it results in $n - \text{integer}(n/m)]m$, where $\text{integer}(n/m)$ is the integer resulting from the truncation of n/m.

- Thus, we have

$$\text{mod}(44, 7) = 2, \qquad \text{mod}(3, 11) = 3, \qquad \text{mod}(n, m) = 0$$

whenever m divides n evenly.

- We also note that

$$x \equiv y \text{ modulo } (z)$$

means that $x - y$ is divisible by z.

Combined Linear Congruential Generators

- Reason: Longer period generator is needed because of the increasing complexity of simulated systems.

- Approach: Combine two or more multiplicative congruential generators.

- Let $X_{i,1}$, $X_{i,2}$, . . . $X_{i,k}$ be the i-th output from k different multiplicative congruential generators:
  - The j-th generator has prime modulus m_j and multiplier a_j and period is $m_j - 1$.

Combined Linear Congruential Generators

- Suggested form:

$$X_i = \left(\sum_{j=1}^{k} (-1)^{j-1} X_{i,j}\right) \bmod (m_1 - 1)$$

- The returned value is:

$$R_i = \begin{cases} X_i/m_1 & \text{if } X_i > 0 \\ (m_1 - 1)/m_1 & \text{if } X_i = 0. \end{cases}$$

- The maximum possible period is:

$$p = \frac{(m_1 - 1)(m_2 - 1) \cdots (m_k - 1)}{2^{k-1}}$$

Mother of All Pseudo-Random-Number Generators

- Outlines of **two other random-number generator algorithms** follow:

Algorithm 1

- *Initialize the four values of* x_0, x_1, x_2, x_3 *and* c *to random values based on a value of the seed.*
- *Let*

$$s = 2111111111x_{n-4} + 1492x_{n-3} + 1776x_{n-2} + 5115x_{n-1} + c$$

- *Compute*

$$x_n = s \bmod (2^{32}) \qquad (n \geq 4)$$

$$c = \lfloor s/2^{32} \rfloor$$

- Invented by George Marsaglia.

http://www.agner.org/random/

Mother of All Pseudo-Random-Number Generators

- Producing uniformly distributed pseudo random 32 bit values with period about 2^{250}.
- **George Marsaglia**: “Because I want to describe a generator, or rather, a class of generators, so promising I am inclined to call it

  The Mother of All Random Number Generators

  and because the generator seems promising enough to justify shortcutting the many months, even years, before new developments are widely known through publication in a journal.”

George Marsaglia, PhD (1924-2011), Professor Emeritus of Washington State University, Florida State University.

Algorithm 2

`rand()` *in Unix*

- *Initialize the* x_0 *to a random value based on a value of the seed.*
- *Compute*

$$x_{n+1} = (1103515245x_n + 12345) \mod (2^{31}) \qquad (n \geq 1)$$

- On the Internet, one can find **new and improved pseudo-random-number generators**, which are designed for the fast generations of high-quality random numbers with colossal periods and with special distributions.

http://www.gnu.org/software/gsl

- A few **words of caution** about random-number generators in computing systems are needed.

- The fact that the sequences produced by these programs are **not truly random** has already been noted.

- In some simulations, the failure of randomness can lead to erroneous conclusions.

Here are several specific points and examples to remember:

Properties 1

- *The algorithms of the type illustrated here by* **Random** *and those above produce* **periodic sequences**; *that is, the sequences eventually repeat themselves.*
- *If a random-number generator is used to produce* **random points in** *n-* **dimensional space**, *these points lie on a relatively small number of planes or hyperplanes.*
- As Marsaglia [1968] reports, points obtained in **3-space** lie on a set of only 119086 planes for computers with integer storage of 48 bits. In **10-space** they lie on a set of 126 planes, which is quite small.

George Marsaglia, PhD (1924-2011), Professor Emeritus of Washington State University, Florida State University.

- The individual digits that make up random numbers generated by routines such as **Random** *are* **not**, *in general,* **independent random digits**.

- For example, in the **individual digits** that make up random numbers generated by routines, it might happen that the digit 3 follows the digit 5 more (or less) often than would be expected.

Random number shuffling

- A **random-number generator** produces a sequence of numbers that are random in the sense that they are uniformly distributed over a certain interval such as (0, 1) and it is not possible to predict the next number in the sequence from knowing the previous ones.

- One can increase the randomness of such a sequence by a suitable **shuffle** of them.

- The idea is to fill an array with the consecutive numbers from the random-number generator and then to use the generator again to choose at random which of the numbers in the array is to be selected as the next number in a new sequence.

- The hope is that the new sequence is more random than the original one.
- For example, a shuffle can remove any correlation between near successors of a number in a sequence.
- See Flowers [1995] for a shuffling procedure that can be used with a random-number generator based on a linear congruence.
- It is particularly useful on computers with a small word length.

Test for Randomness

- There are **statistical tests that can be performed on a sequence of random numbers**.

- While such tests do not certify the randomness of a sequence, they are particularly important in applications.

- For example, they are useful in choosing between different random-number generators, and it is comforting to know that the random-number generator being used has passed such tests.

- Situations exist when random-number generators are useful even though they do not pass rigid tests for true randomness.

- Thus, if one is producing random matrices for testing a linear algebra code, then strict randomness may not be important.

- On the other hand, strict randomness is **essential** in Monte Carlo integration and other applications.

- In these cases in which strict randomness is important, it is recommended that one use a machine with a large word size and a random-number generator with known statistical characteristics.

Prime number

- A **prime number** is an integer greater than 1 whose only factors (divisors) are itself and 1.
- Prime numbers are some of the fundamental building blocks in mathematics.
- The search for large primes has a long and interesting history.
- In 1644, **Mersenne** (a French friar) conjectured that $2^n - 1$ was a prime number for $n = 17, 19, 31, 67, 127, 257$ and for no other n in the range $1 \leq n \leq 257$.
- In 1876, Lucas proved that $2^{127} - 1$ was prime.
- In 1937, however, Lehmer showed that $2^{257} - 1$ was **not** prime. Until 1952, $2^{127} - 1$ was the largest known prime.
- Then it was shown that $2^{521} - 1$ was prime.

- As a means of testing new computer systems, **the search for ever-larger Mersenne primes continues**.
- In fact, the search for ever **larger primes** has grown in importance for use in cryptology.
- In 1992, a Cray 2 supercomputer using the Lucas-Lehmer test determined after a 19-hour computation that the number $2^{756,839} - 1$ was a prime.
- This number has 227,832 digits!
- In 2006, the largest known prime $2^{32,582,657} - 1$, with 9.8 million digits, was discovered using the Internet facility **GIMPS** (Great Internet Mersenne Prime Search).
- In 2016, the largest known prime $2^{74,207,281} - 1$, with 22 million digits.

- Thousands of individuals have used the GIMPS database to facilitate their search for large primes, and interaction with the database can be done automatically without human intervention.
- For more **information on large primes** and to find out the current record for the largest known prime, consult

http://www.mersenne.org/prime.html

http://www.utm.edu/research/primes

Plan for Today

- Computer simulation of the tossing of a coin
- Generating random points in geometric configuration
- Estimation of Areas and Volumes by Monte Carlo Techniques

- Implement the following pseudocode to generate a new array of pseudo-random numbers per call

real procedure Random $((x_i))$
integer $seed, i, n$; **real array** $(x_i)_{1:n}$
integer $k \leftarrow 16807, j \leftarrow 2147483647$
$seed \leftarrow$ select initial value for $seed$
$n \leftarrow size((x_i))$
for $i = 1$ **to** n
  $seed \leftarrow \mod(k \cdot seed, j)$
  $x_i \leftarrow \text{real}(seed)/\text{real}(j)$
end procedure Random

Define function in this way:

```
function [X]=random(n, seed)
.....
endfunction
```

Computer simulation of the tossing of a coin

- A single toss corresponds to the selection of a random number x in the interval (0, 1).
- We arbitrarily associate heads with event $0 < x \leq \frac{1}{2}$ and tails with event $\frac{1}{2} < x < 1$.
- One thousand tosses of the coin corresponds to 1000 choices of random numbers.
- The results show the proportion of heads that result from repeated tossing of the coin.

- In the pseudocode to be shown next, a sequence of 10000 random numbers is generated.

- Along the way, the current proportion of numbers less than 1/2 is computed at the 1000th step and then at multiples of 1000.

- Some of the computer results of the experiment are 49.5, 50.2, 51.0, and 50.625.

Here is the pseudocode to carry out this experiment:

```
program Coarse_Check
integer i, m;   real per;   real array (r_i)_{1:n}
integer n ← 10000
m ← 0
call Random ((r_i))
for i = 1 to n
|    if r_i ≤ 1/2 then m ← m + 1
|    if mod(i, 1000) = 0 then
|    |    per ← 100 real(m)/real(i)
|    |    output i, per
end program Coarse_Check
```

- Observe that (at least in this experiment) reasonable precision is attained with only a moderate number of random numbers (4000).

- Repeating the experiment 10000 times has only a marginal influence on the precision.

- Of course, theoretically, if the random numbers were truly random, the limiting value as the number of random numbers used increases without bound would be exactly 50%.

- In this pseudocode and others in the chapter, all of the random numbers are generated initially, stored in an array, and used later in the program as needed.

Generating random points in geometric configuration

- Now we consider some basic questions about generating random points in various geometric configurations.
- Assume that procedure **Random** is used to obtain a random number r in the interval $(0, 1)$.

First, if uniformly distributed random points are needed on some **interval** (a, b), this statement accomplishes this.

$$x \leftarrow (b - a)r + a$$

Second, this pseudocode produces random integers in the **set** $\{0, 1, \ldots, n\}$.

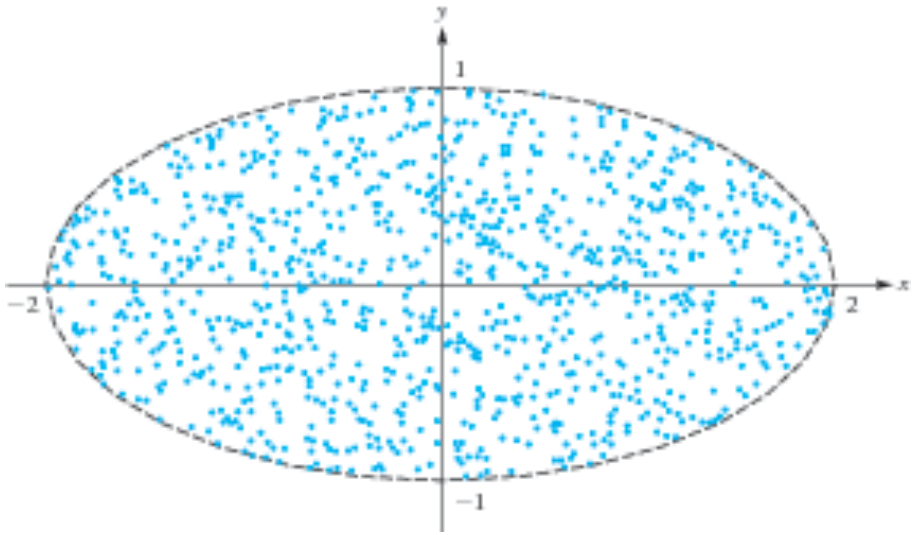
$$i \leftarrow \text{integer}\,((n + 1)r)$$

Third, for **random integers from** j **to** k $(j \leq k)$, use the assignment statement

$$i \leftarrow \text{integer}\,((k - j + 1)r + j)$$

Uses of Pseudocode *Random*

Consider the problem of generating 1000 random points uniformly distributed inside the ellipse $x^2 + 4y^2 = 4$.



Uniformly distributed random points in ellipse $x^2 + 4y^2 = 4$

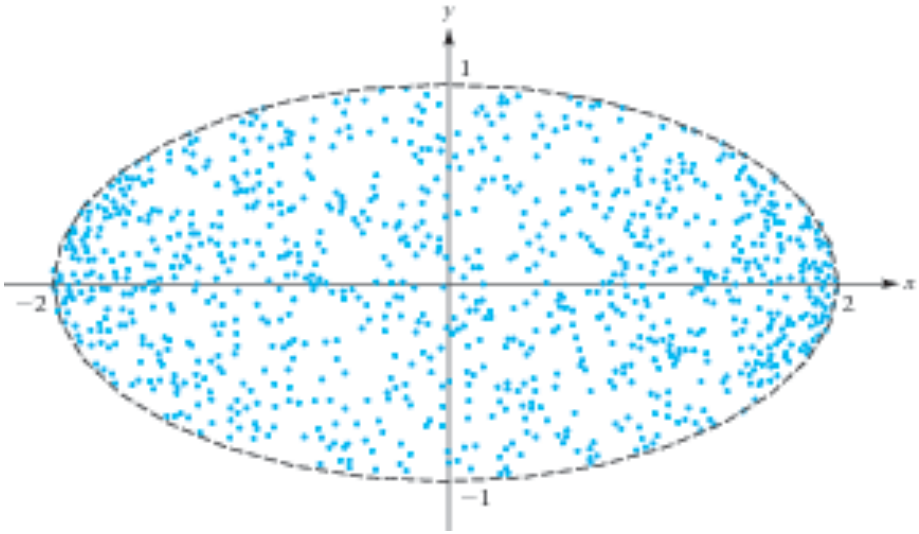
One way to do so is to generate random points in the rectangle $-2 \leq x \leq 2, -1 \leq y \leq 1$, and discard those that do not lie in the ellipse.

```
program Ellipse
integer i, j;   real u, v;   real array (x_i)_{1:n}, (y_i)_{1:n}, (r_ij)_{1:npts×1:2}
integer n ← 1000, npts ← 2000
call Random ((r_ij))
j ← 1
for i = 1 to npts
    u ← 4r_{i,1} − 2
    v ← 2r_{i,2} − 1
    if u^2 + 4v^2 ≤ 4 then
        x_j ← u
        y_j ← v
        j ← j + 1
        if j = n then exit loop i
end program Ellipse
```

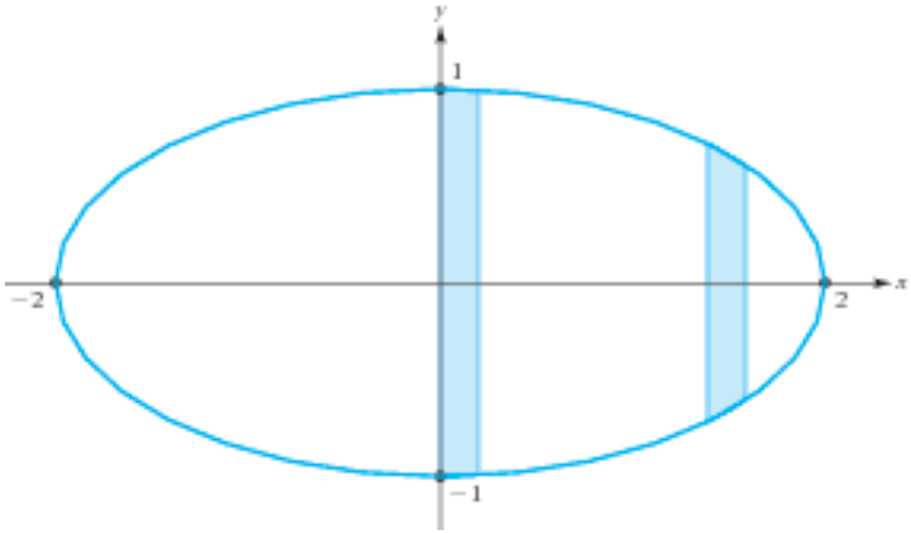
To be less wasteful, we can **force** the $|y|$ value to be less than $\frac{1}{2}\sqrt{4 - x^2}$, as in the following pseudocode

program **Ellipse**
integer i; **real array** $(x_i)_{1:n}, (y_i)_{1:n}, (r_{ij})_{1:n \times 1:2}$
integer $n \leftarrow 1000$
call **Random** $((r_{ij}))$
for $i = 1$ **to** n
    $x_i \leftarrow 4r_{i,1} - 2$
    $y_i \leftarrow [(2r_{i,2} - 1)/2]\sqrt{4 - x_i^2}$
end program **Ellipse**

To be less wasteful, we can **force** the $|y|$ value to be less than $\frac{1}{2}\sqrt{4 - x^2}$, as in the previous pseudocode, **which produces erroneous results**



Nonuniformly distributed random points in the ellipse $x^2 + 4y^2 = 4$



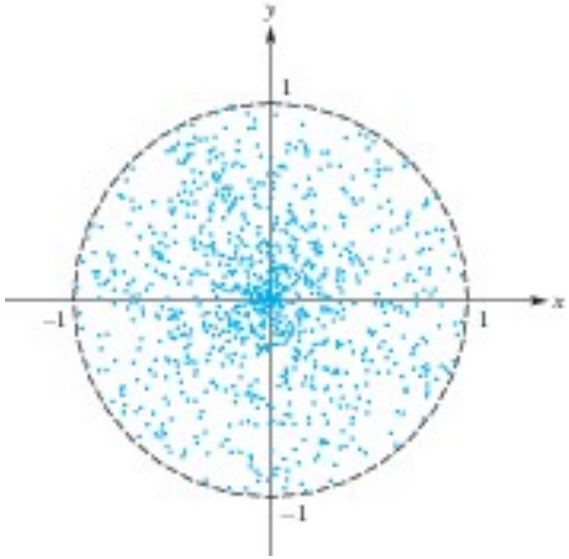
Vertical strips containing nonuniformly distributed points

- This pseudocode does **not** produce uniformly distributed points inside the ellipse.
- To be convinced of this, consider two vertical strips taken inside the ellipse
- If each strip is of width h, then approximately $1000(h/4)$ of the random points lie in each strip because the random variable x is uniformly distributed in $(-2, 2)$, and with each x, a corresponding y is generated by the program so that (x, y) is inside the ellipse.
- But the two strips shown should **not** contain approximately the same number of points because they do not have the same area.
- The points generated by the second program tend to be clustered at the left and right extremities of the ellipse in the figure.

Does following code produce uniformly distributed random points in the circle $x^2 + y^2 = 1$?

program Circle
integer i; **real array** $(x_i)_{1:n}, (y_i)_{1:n}, (r_{ij})_{1:n\times 1:2}$
integer $n \leftarrow 1000$
call Random $((r_{ij}))$
for $i = 1$ **to** n
    $x_i \leftarrow r_{i,1}\cos(2\pi r_{i,2})$
    $y_i \leftarrow r_{i,1}\sin(2\pi r_{i,2})$
end program Circle

For the same reasons, the previous pseudocode does **not** produce uniformly distributed random points in the circle $x^2 + y^2 = 1$



Nonuniformly distributed random points in the circle $x^2 + y^2 = 1$

- In this pseudocode, $2\pi r_{i,2}$ is uniformly distributed in $(0, 2\pi)$, and $r_{i,1}$ is uniformly distributed in $(0, 1)$.

- However, in the transfer from polar to rectangular coordinates by the equations

$$x = r_{i,1}\cos(2\pi r_{i,2})$$

$$y = r_{i,1}\sin(2\pi r_{i,2})$$

the **uniformity is lost!**

- The random points are strongly clustered near the origin in the figure.

§10.2 Estimation of Areas and Volumes by Monte Carlo Techniques

Recaps: Multidimensional Integration

- Thus, the quality of the numerical approximation of the integral declines very quickly as the number of variables, k , increases.

- Expressed in other terms, if a constant order of accuracy is to be retained while the number of variables, k , goes up, the number of nodes must go up like n^k .

- These remarks indicate why the **Monte Carlo method for numerical integration becomes more attractive for high-dimensional integration**.

§10.2 Estimation of Areas and Volumes by Monte Carlo Techniques: Numerical Integration

- Now we turn to applications, the first being the **approximation of a definite integral by the Monte Carlo method**.
- If we select the first n elements $x_1, x_2, \ldots, x_n$ from a random sequence in the interval (0, 1), then

§10.2 Estimation of Areas and Volumes by Monte Carlo Techniques: Numerical Integration

- Now we turn to applications, the first being the **approximation of a definite integral by the Monte Carlo method**.
- If we select the first n elements $x_1, x_2, \ldots, x_n$ from a random sequence in the interval (0, 1), then

$$\int_0^1 f(x)\,dx \approx \frac{1}{n}\sum_{i=1}^{n} f(x_i)$$

- Here, the integral is approximated by the average of n numbers $f(x_1)$, $f(x_2), \ldots, f(x_n)$.

- To obtain **random points in the cube**, we assume that we have a random sequence in (0, 1) denoted by $\xi_1, \xi_2, \xi_3, \xi_4, \xi_5, \xi_6, \ldots$
- To get our first random point p_1 in the cube, just let $p_1 = (\xi_1, \xi_2, \xi_3)$.
- The second is, of course, $p_2 = (\xi_4, \xi_5, \xi_6)$, and so on.

- When this is actually carried out, the error is of order $1/\sqrt{n}$, which is not at all competitive with good algorithms, such as the Romberg method.
- However, in **higher dimensions**, the Monte Carlo method can be quite attractive.
- For example, we use

$$\int_0^1 \int_0^1 \int_0^1 f(x, y, z)\, dx\, dy\, dz \approx \frac{1}{n} \sum_{i=1}^{n} f(x_i, y_i, z_i)$$

where (x_i, y_i, z_i) is a random sequence of n points in the **unit cube** $0 \leq x \leq 1$, $0 \leq y \leq 1$, and $0 \leq z \leq 1$.

- If the interval (in a one-dimensional integral) is not of length 1 but, say, is the general case (a, b), then the average of f over n random points in (a, b) is not simply an approximation for the integral but rather for

$$\frac{1}{b-a}\int_a^b f(x)\,dx$$

which agrees with our intention that the function $f(x) = 1$ have an average of 1.

- Similarly, in higher dimensions, the average of f over a region is obtained by integrating and dividing by the area, volume, or measure of that region.
- For instance,

$$\frac{1}{8}\int_1^3\int_{-1}^1\int_0^2 f(x,y,z)\,dx\,dy\,dz$$

is the average of f over the **parallelepiped** described by the following three inequalities: $0 \le x \le 2$, $-1 \le y \le 1$, $1 \le z \le 3$.

- So if (x_i, y_i) denote random points with appropriate uniform distribution, the following examples illustrate **Monte Carlo techniques**:

$$\int_0^5 f(x)\,dx \approx \frac{5}{n}\sum_{i=1}^{n} f(x_i)$$

$$\int_2^5 \int_1^6 f(x,y)\,dx\,dy \approx \frac{15}{n}\sum_{i=1}^{n} f(x_i, y_i)$$

- In each case, the random points should be **uniformly distributed in the regions** involved.

- In general, we have

$$\int_A f \approx (\text{measure of } A) \times (\text{average of } f \text{ over } n \text{ random points in } A)$$

- Here, we are using the fact that the **average of a function on a set** is equal to the integral of the function over the set divided by the measure of the set.

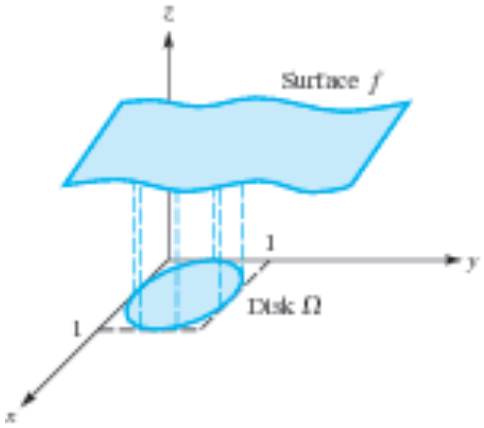
Example and Pseudocode

- Let us consider the problem of obtaining the numerical value of the integral

$$\int_{\Omega} \sin \sqrt{\ln(x+y+1)}\, dx\, dy = \int_{\Omega} f(x,y)\, dx\, dy$$

over the **disk in xy-space**, defined by the inequality

$$\Omega = \left\{(x,y) : \left(x-\tfrac{1}{2}\right)^2 + \left(y-\tfrac{1}{2}\right)^2 \leq \tfrac{1}{4}\right\}$$



Sketch of surface $f(x, y)$ above disk Ω

- Above is a sketch of this domain, with a surface above it.

- We proceed by generating random points in the square and discarding those that do not lie in the disk.
- We take $n = 5000$ points in the disk.
- If the points are $p_i = (x_i, y_i)$, then the **integral is estimated** to be

$$\int_\Omega f(x,y)\,dx\,dy \approx (\text{area of disk } \Omega) \times \begin{pmatrix} \text{average height of } f \\ \text{over } n \text{ random points} \end{pmatrix}$$

$$= (\pi r^2)\Big[\frac{1}{n}\sum_{i=1}^{n} f(p_i)\Big]$$

$$= \frac{\pi}{4n}\sum_{i=1}^{n} f(p_i)$$

- The pseudocode for this example follows.
- Intermediate estimates of the integral are printed when n is a multiple of 1000.
- This gives us some idea of how the correct value is being approached by our averaging process.

```
program Double_Integral
integer i, j:   real sum, vol, x, y;   real array (r_ij)_{1:n×1:2}
integer n ← 5000, iprt ← 1000;   external function f
call Random ((r_ij))
j ← 0;   sum ← 0
for i = 1 to n
|    x = r_{i,1};     y = r_{i,2}
|    if (x − 1/2)^2 + (y − 1/2)^2 ≤ 1/4 then
|    |    j ← j + 1;     sum ← sum + f(x, y)
|    if mod(j, iprt) = 0 then   vol ← (π/4)sum/real(j)
|    output j, vol
vol ← (π/4)sum/real(j)
output j, vol
end program Double_Integral
```

real function $f(x, y)$
real x, y
$f \leftarrow \sin\left(\sqrt{\ln(x + y + 1)}\right)$
end function

- We obtain an approximate value of 0.57 for the integral.