AMTH250

Random Numbers and Simulation

August 28, 2010

Contents

1 Review of Probability

This is just a reminder of few facts about continuous probability distributions covered in MATH102.

1.1 Definitions

Continuous probability distributions are described by **probability densities**, i.e. real valued functions $\rho(x)$ with the properties:

1. $\rho(x) \geq 0$.
2. $\displaystyle\int_{-\infty}^{\infty} \rho(x)dx = 1$.
3. $\text{Prob}(a \leq x \leq b) = \displaystyle\int_a^b \rho(x)dx$.

The **distribution function** associated with the density $\rho(x)$ is defined by

$$F(x) = \int_{-\infty}^{x} \rho(t)dt.$$

It has the properties:

1. $F(s) = \text{Prob}(x \leq s)$.
2. $\displaystyle\lim_{x \to -\infty} F(x) = 0$
3. $\displaystyle\lim_{x \to \infty} F(x) = 1$
4. $F(x)$ is an increasing function of x.
5. $\dfrac{dF}{dx} = \rho(x)$

It follows from 1. that

$$\text{Prob}(a \leq x \leq b) = F(b) - F(a).$$

The **mean** and **variance** of a density $\rho(x)$ are defined by

$$\mu = \text{Mean}(x) = \int_{-\infty}^{\infty} x\rho(x)dx$$

and

$$\sigma^2 = \text{Var}(x) = \int_{-\infty}^{\infty} (x-\mu)^2 \rho(x)dx$$

The **standard deviation**, σ, is the square root of the variance.

1.2 Uniform Density

The **uniform density** on the interval $[a, b]$ is defined by

$$\rho(x) = \begin{cases} 0 & x < a \\ \frac{1}{b-a} & a \le x \le b \\ 0 & x > b \end{cases}$$

An important special case is the uniform density on $[0, 1]$

$$\rho(x) = \begin{cases} 0 & x < 0 \\ 1 & 0 \le x \le 1 \\ 0 & x > 1 \end{cases}$$

This density has mean $\mu = 1/2$ and variance $\sigma^2 = 1/12$.

1.3 Normal Density

The **normal density** with mean μ and standard deviation σ is defined by

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

The case $\mu = 0$, $\sigma = 1$ is especially important

$$\rho(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$$

2 Random Number Generators

Computer algorithms for generating random numbers are *deterministic* algorithms. Although the sequence of numbers produced by a random number generator appears random, the sequence of numbers is completely predictable and for this reason they are often called **pseudo-random**.

Since computers have only a finite number of different states, the sequence of pseudo-random numbers produced by any deterministic algorithm must necessarily repeat itself. The number of random numbers generated before the sequence repeats is called the *period* of the generator.

A good random number generator should have the following characteristics:

1. *Randomness.* It should pass statistical tests of randomness.

2. *Long Period.* For obvious reasons.

3. *Efficiency.* This is important since simulations often require millions of random numbers.

4. *Repeatability.* It should produce the same sequence of numbers if started in the same state. This allows the repetition and checking of simulations.

Almost all random number generators used in practice produce uniform $[0,1]$ distributed random numbers and from these random numbers with other distributions can be produced if required.

2.1 Algorithms

There are many algorithms for generating pseudo-random numbers.

Linear Congruential Generators

The simplest are the **linear congruential** generators. Starting with the *seed* x_0, these generate a sequence of *integers* by

$$x_k = (ax_{k-1} + c) \bmod M \tag{1}$$

where a, c and M are given integers. All the x_k are integers between 0 and $M-1$. In order to produce floating point numbers these are divided by M to give a floating point number in the interval $[0,1)$.

Below is a Scilab function[1] implementing a linear congruential generator. Here `n` is the number of terms generated `a`, `c` and `m` are as in equation (1) and `x0` is the initial value or seed. A vector of `n+1` values including the seed is returned.

```
function x = lcg(n, a, c, m, x0)
  x = zeros(1,n+1)
  x(1) = x0
  for i = 2:n+1
    x(i) = pmodulo(a*x(i-1)+c, m)
  end
  x = x/m
endfunction
```

The quality of a linear congruential generator depends on the choice of a, b and M, but in any case the period of such a generator is at most M (why?). As we will see later, Scilab's basic `rand` generator is a linear congruential generator.

The following example illustrates a common problem with some random number generators. We will take the linear congruential generator with $a = 1203$, $c = 0$, $m = 2048$. With initial value $x_0 = 1$, this generator has period 512.

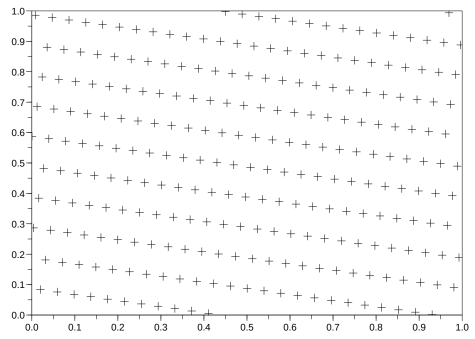
We will run through a full period of the generator:

[1]The version in Scilab functions accompanying this Study Guide has been modified to avoid a problem with rounding error.

```
-->xx = lcg(511, 1203, 0, 2048, 1);

-->xx(1:10)
 ans  =
        column 1 to 4
!   0.0004883    0.5874023    0.6450195    0.9584961 !

        column 5 to 8
!   0.0708008    0.1733398    0.5278320    0.9819336 !

        column  9 to 10
!   0.2661133    0.1342773 !
```

Now take successive pairs of values as the x and y coordinates of a point in the plane and plot the results.

```
-->x = xx(1:2:511);

-->y = xx(2:2:512);

-->plot2d(x,y,style = -1)
```



This "latticing" is a common problem with random number generators. Of course, we have used a very simple generator for which the problem is obvious, for other generators the problem may occur in higher dimensions. That is, taking successive n-tuples of values and plotting them (hypothetically at least) in n-dimensional space the points may tend to lie on a lattice of lower dimensional subspaces.

Fibonacci Generators

Another common type of generator are the **Fibonacci generators**. A typical example generates a sequence by

$$x_k = (x_{k-17} - x_{k-5}) \bmod 1$$

This example uses the previous 17 numbers (and needs 17 seeds to get started). It has a number of advantages compared to congruential generators:

1. The x_k are floating point numbers, so no division is needed.
2. It has a much longer period since the repetition of a single number does not entail repetition of the whole sequence.
3. It can have much better statistical properties.

Combined Random Number Generators

By combining a few simple random number generators we can obtain a generator with better properties. One generator used by Scilab combines four linear congruential generators

$$\begin{aligned}
t_n &= 45991 t_{n-1} \bmod 2147483647\\
u_n &= 207707 u_{n-1} \bmod 2147483543\\
v_n &= 138556 v_{n-1} \bmod 2147483423\\
w_n &= 49689 w_{n-1} \bmod 2147483323\\
x_n &= (t_n - u_n + v_n - w_n) \bmod 2147483647
\end{aligned}$$

This generator has a period of about 2^{123}.

Other Random Number Generators

Modern random number generators are based on number theory and have quite sophisticated implementations. One in common use is the **Mersenne twister** which takes a vector of 625 values as its seed.

2.2 Scilab

Scilab has two random number generators `rand` and `grand`. We will only discuss the simpler generator `rand` which can produce either uniform $[0, 1]$ or normal $\mu = 0, \sigma = 1$ random numbers. Uniform random numbers are the default.

We have already seen how `rand` works:

1. `rand(m, n)` – gives a $m \times n$ matrix of random numbers.
2. `rand(m, n, 'normal')` – gives a $m \times n$ matrix of normally distributed random numbers.
3. `rand(a)` or `rand(a, 'normal')` – for matrix `a` gives a matrix of random numbers the same size as `a`.

4. `rand('seed', 0)` – resets the random number generator to its original state. This is handy if you want to repeat an experiment using the same random numbers.

As mentioned earlier, `rand` is a linear congruential generator:

$$x_n = (ax_{n-1} + c) \bmod M$$

with $a = 843314861$, $c = 453816693$, and $M = 2^{31}$.

Let us check this. First we set the seed of `rand` to `0` (which is the seed on the first call to `rand`) and generate 10000 terms.

```
-->rand("seed",0)

-->x1 = rand(1,10000);
```

Now generate 10000 terms of `lcg`[2] with parameters above and seed `0`.

```
-->x2 = lcg(10001, 843314861, 453816693, 2^31, 0);
```

We drop off the initial term (which is the seed) and look at first few terms:

```
-->x2 = x2(2:10001);

-->x1(1:8)
 ans  =
        column 1 to 4
!   0.2113249    0.7560439    0.0002211    0.3303271 !

        column 5 to 8
!   0.6653811    0.6283918    0.8497452    0.6857310 !

-->x2(1:8)
 ans  =
        column 1 to 4
!   0.2113249    0.7560439    0.0002211    0.3303271 !

        column 5 to 8
!   0.6653811    0.6283918    0.8497452    0.6857310 !
```

and finally find how many terms differ:

```
-->sum(x1 ~= x2)
 ans  =

    0.
```

[2]Use the version in the Scilab functions accompanying this Study Guide, see previous footnote.

3 Some Common Distributions

Two simple change of variable tricks are very important.

3.1 Uniform Distribution

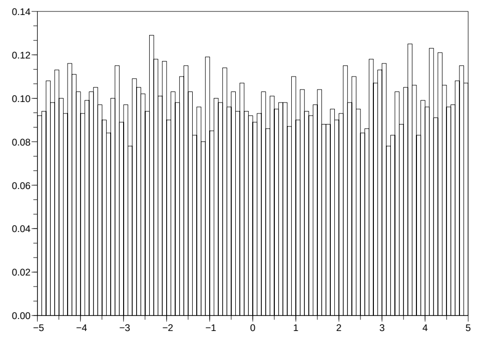
If x is a uniform $[0, 1]$ distribution, then the change of variable

$$y = (b - a)x + a$$

gives a uniform $[a, b]$ distribution.

Here is an example generating a uniform $[-5, 5]$ distribution:

```
-->x =rand(1,10000);

-->y = 10*x - 5;

-->histplot(100,y)
```



3.2 Normal Distribution

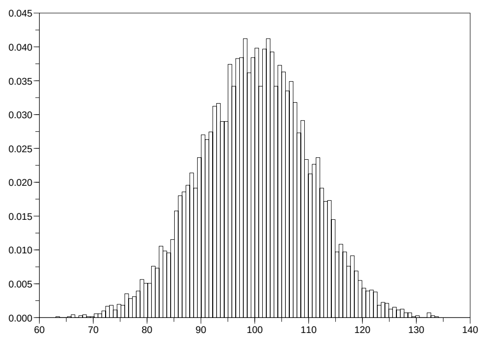
If x has a normal $\mu = 0$, $\sigma = 1$ distribution, then the change of variable

$$y = ax + b$$

gives a normal distribution with $\mu = b$ and $\sigma = a$.

Here is an example generating a normal $\mu = 100$, $\sigma = 10$ distribution:

```
-->x = rand(1,10000,'normal');

-->y = 10*x + 100;

-->histplot(100,y)
```



3.3 Uniform Discrete Distributions

It is quite common to want to generate uniformly distributed random integers. Typically they will be in the range $0 \leq k \leq n$ or $1 \leq k \leq n$. These can be obtained from uniform $[0, 1]$ random floating numbers x by

1. `floor((n+1)*x)` gives integers in the range $0 \leq k \leq n$.
2. `floor(n*x+1)` gives integers in the range $1 \leq k \leq n$.

Here is an example generating integers in the range 1 to 10 (inclusive):

```
-->x = rand(1,30);

-->y = floor(10*x+1)
 y  =
         column  1 to 10
!   9.    7.    9.    3.    4.    9.    9.    1.    2.    4. !

         column 11 to 20
!   9.    2.    4.    2.    7.    6.    3.    6.    10.    8. !


!   4.    7.    5.    4.    8.    2.    7.    1.    10.    1. !
```

4 Non-Uniform Random Numbers

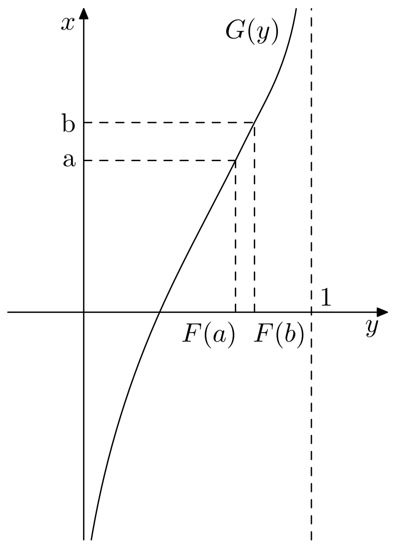
4.1 Inverse Transform Method

Consider a probability density $\rho(x)$ and let $F(x)$ be the corresponding distribution function. Then

$$F : \mathbb{R} \to [0, 1]$$

and since $F(x)$ is an increasing function it has an inverse function $G(y)$ with

$$G : [0,1] \to \mathbb{R}$$



Let y be uniformly distributed on $[0,1]$ and set

$$x = G(y).$$

Then

$$\text{Prob}(a \leq x \leq b) = \text{Prob}(F(a) \leq y \leq F(b)) = F(b) - F(a)$$

Where the last equality follows from the uniformity of y. The statement

$$\text{Prob}(a \leq x \leq b) = F(b) - F(a)$$

is equivalent to the statement that x has distribution function $F(x)$ and hence x has density $\rho(x)$. In summary:

> *Given a probability density $\rho(x)$ with distribution function $F(x)$, let $G(y)$ be the inverse function of $F(x)$. Then if we take y to be uniformly distributed on $[0,1]$, $x = G(y)$ has probability density $\rho(x)$.*

Exponential Distribution

This is defined by

$$\rho(x) = \begin{cases} 0 & x < 0 \\ \lambda e^{-\lambda x} & x \geq 0 \end{cases}$$

with distribution function

$$F(x) = \begin{cases} 0 & x < 0 \\ 1 - e^{-\lambda x} & x \geq 0 \end{cases}$$

(Here λ is a parameter).

Setting

$$y = F(x) = 1 - e^{-\lambda x}$$

and solving for x we obtain the inverse function

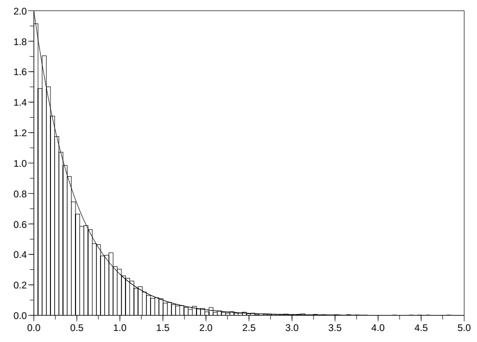
$$x = G(y) = -\frac{\ln(1-y)}{\lambda}.$$

Lets see how it works in Scilab; we will take $\lambda = 2$.

```
-->y = rand(1,10000);

-->x = - log(1-y)/2;

-->histplot(100,x)
```

We can compare the histogram to exact density:

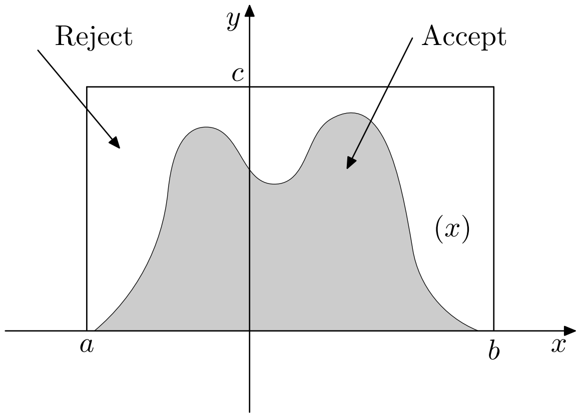
```
-->xx = 0:0.01:5;

-->plot2d(xx, 2*exp(-2*xx))
```



One drawback of the inverse transform method is that we need to know the inverse of the distribution function. For most distributions there is no explicit formula for this inverse and we must resort to approximations or seek another method.

4.2 Acceptance/Rejection Method

This method too has its drawbacks. We will need to assume that we are working with a density which is zero outside of a finite interval. Suppose that $\rho(x)$ is zero outside of the interval $[a, b]$ and furthermore that $\rho(x)$ is bounded above by c.

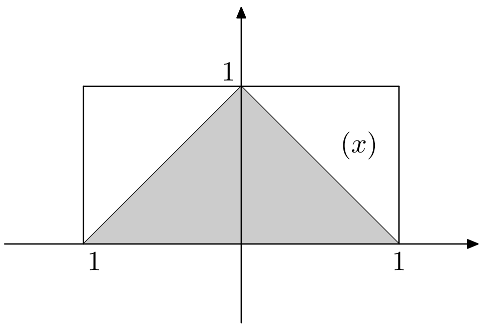


Now generate points (x_i, y_i) with x_i uniformly distributed in $[a, b]$ and y_i uniformly distributed in $[0, c]$. If $y_i \leq \rho(x_i)$ then we accept the value x_i, if $y_i > \rho(x_i)$ then we reject the value x_i. The values x_i which are accepted will have the probability density $\rho(x)$.

An Example

Consider the hat-shaped probability density defined on $[-1, 1]$ by

$$\rho(x) = \begin{cases} x + 1 & -1 \leq x \leq 0 \\ 1 - x & 0 \leq x \leq 1 \end{cases}$$



We want to write a Scilab function to generate n random numbers with this density using the acceptance/rejection method. One thing to keep in mind when using the acceptance/rejection is that we do not know before hand how many random numbers we need to generate.

First we need the density function [3]:

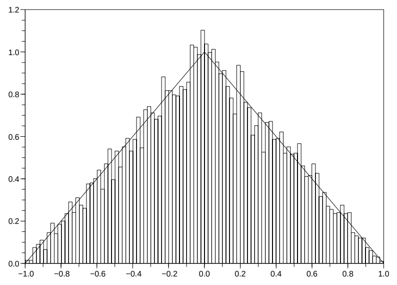
```
function y = rhohat(x)
  if (x < 0) then
    y = x + 1
  else
    y = 1 - x
  end
endfunction
```

Now the random number generator:

```
function x = randhat(n)
  x = zeros(1,n)
  k = 0               // keep count of numbers generated
  while (k < n)
    xx = -1 + 2*rand(1,1)      // uniform on [-1,1]
    yy = rand(1,1)
    if (yy <= rhohat(xx))      // accept xx
      k = k + 1
      x(k) = xx
    end
  end
endfunction
```

And testing it

```
-->x = randhat(10000);

-->histplot(100, x)

-->xx = -1:0.01:1;

-->plot2d(xx, rhohat(xx))
```



[3]The version of `rhohat` in the Scilab functions accompanying this Study Guide has been modified to allow a vector `x` as the argument.

4.3 Special Distributions

The two methods discussed above, acceptance/rejection and inverse transform, when applied with some ingenuity can be used to generate random numbers with almost any distribution. However for most distributions there are specific techniques that perform better than these more general techniques. The Scilab function `grand` has facilities for generating random numbers from most of common distributions.

Normal Distribution

Here is a clever method for generate normally distributed random numbers: generate two uniform $[0, 1]$ random numbers x_1 and x_2, then

$$y_1 = \sin(2\pi x_1)\sqrt{-2\ln x_2}$$

and

$$y_2 = \cos(2\pi x_1)\sqrt{-2\ln x_2}$$

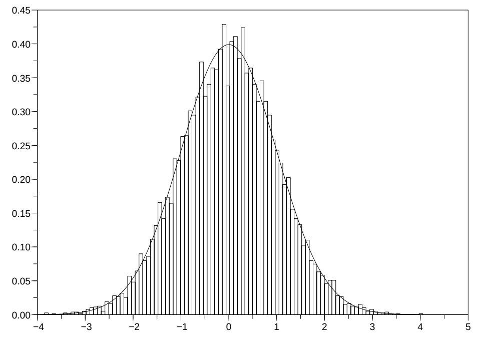
are both normally distributed with $\mu = 0$, $\sigma = 1$.

Here is how it works in Scilab:

```
function x = randnorm(n)  // assumes n even
  m = n/2
  x1 = rand(1,m)
  x2 = rand(1,m)
  y1 = sin(2*%pi*x1).*sqrt(-2*log(x2));
  y2 = cos(2*%pi*x1).*sqrt(-2*log(x2));
  x = [y1 y2]
endfunction

-->x = randnorm(10000);

-->histplot(100,x)
```

We can compare the histogram to exact density:

```
-->xx = -4:0.01:4;

-->yy = exp(-(xx.^2)/2)/sqrt(2*%pi);

-->plot2d(xx, yy)
```



5 Monte-Carlo Integration

5.1 One Dimension

We wish to evaluate an integral

$$I = \int_a^b f(x)dx.$$

The average value of $f(x)$ is

$$\bar{f} = \frac{1}{b-a} I$$

so that

$$I = (b-a)\bar{f}.$$

Now let X be a uniformly distributed random variable with values in the interval of integration $[a,b]$. Then the density is

$$\rho(x) = \begin{cases} 0 & \text{if } x < a \text{ or } x > b \\ \dfrac{1}{b-a} & \text{if } a \le x \le b. \end{cases}$$

For the random variable $f(X)$ we compute the expected value

$$\mu = \int f(x)\rho(x)dx = \int_a^b f(x)\frac{1}{b-a}dx = \frac{1}{b-a}\int_a^b f(x)dx = \bar{f}.$$

The variance of $f(X)$ is

$$\sigma^2 = \int (f(x)-\mu)^2\rho(x)dx = \int_a^b (f(x)-\mu)^2\frac{1}{b-a}dx = \frac{1}{b-a}\int_a^b (f(x)-\mu)^2 dx.$$

Now for n uniformly distributed mutually independent random variables $X_1, \ldots, X_n$ we define

$$\bar{f}_n = \frac{1}{n}\sum_{i=1}^{n} f(X_i).$$

Expected value and variance are

$$E(\bar{f}_n) = \mu = \bar{f}$$
$$\text{var}(\bar{f}_n) = \frac{\sigma^2}{n}.$$

where σ^2 is the variance of $f(x)$.

It is usual to take the standard deviation as a measure of error. Then we have

$$\text{Error} = \frac{\sigma}{\sqrt{n}}$$

The important point here is that the error goes to zero like $1/\sqrt{n}$. This says, for example, that to decrease the error by a factor of 1000, we must increase the sample size by a factor of 1000000.

Approximation of $\bar{f}$ by $\bar{f}_n$ is called Monte Carlo integration.

Implementation

Here is a Scilab implementation of Monte-Carlo integration of a function `f` over the interval `[a,b]` using `n` points.

```
function ii = monte1d(f, a, b, n)
  x = (b-a)*rand(1,n) + a   // choose n random points in [a,b]
  fx = f(x)                 // evaluate f(x) at each point
  ii = (b-a)*sum(fx)/n      // (b-a) * average of function
endfunction
```

Example

Let us estimate the integral

$$I = \int_0^{2\pi} e^{-x}\sin(x)dx$$

The exact value, from integration by parts, is

$$I = \frac{1}{2}(1 - e^{-2\pi}) = 0.4990663.$$

Here is a Monte-Carlo estimate using Scilab. The first step is to define the function we want to integrate

```
-->function y = f(x)
-->  y = exp(-x).*sin(x)
-->endfunction

-->ii = monte1d(f, 0, 2*%pi, 100000)
 ii  =
    0.4983886
```

We can compare to the exact answer and find the relative error:

```
-->ie = (1 - exp(-2*%pi))/2
 ie  =
    0.4990663

-->err = abs(ii - ie)/ie
 err  =
    0.0013579
```

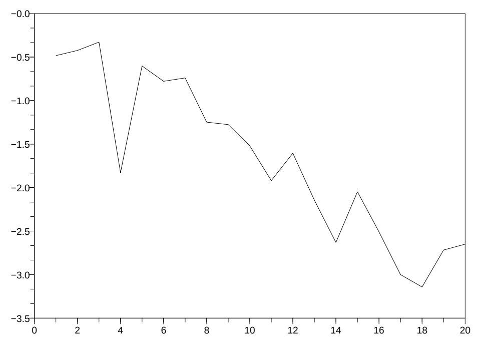
i.e. with 10,000 points we get a relative error of the order of 0.1%.

Let us see how the error varies with the number of points `n`. We will perform a Monte-Carlo approximation with $2, 4, 8, \ldots, 2^{20}$ points using a simple `for` loop and then compute the error for each approximation:

```
-->in = zeros(1,20);

-->for k = 1:20
-->  in(k) = monte1d(f, 0 , 2*%pi, 2^k);
-->end

-->err = abs(in - ie)/ie
 err  =
         column 1 to 4

!   0.3288851    0.3764572    0.4697016    0.0148549 !

         column 5 to 8
!   0.2502828    0.1665171    0.1823111    0.0565038 !

         column  9 to 12
!   0.0530368    0.0301861    0.0120509    0.0248556 !

         column 13 to 16
!   0.0071772    0.0023424    0.0089532    0.0031000 !

         column 17 to 20
!   0.0009980    0.0007203    0.0019182    0.0022399 !
```

As expected the error decreases as the number of points increases. Plotting the log of the error is the most instructive:

```
-->plot2d(log10(err))
```



We compare now our results in the example with the theory. The variance of $f(x) = e^{-x} \sin x$ is

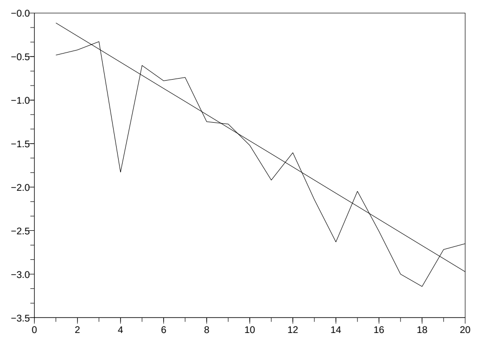
$$\sigma^2 = \int_0^{2\pi} (f(x) - \bar{f})^2 \approx 1.19$$

so the expected error in Monte-Carlo integration is

$$E_n = \frac{\sigma}{\sqrt{n}} \approx \frac{1.09}{\sqrt{n}}$$

We can create a semi-log plot of the estimated Monte-Carlo error as follows:

```
-->n = 1:20;

-->ee = 1.09*(2 .^n).^(-1/2);

-->plot2d(log10(ee))
```



The error in the computed values show random fluctuations, as is to be expected, but that the theory gives a good account of the error.

5.2 Estimating Volumes

What is the volume of a unit sphere in 4-dimensions? We can obtain an estimate by Monte-Carlo methods.

The unit sphere is the region

$$x_1^2 + x_2^2 + x_3^2 + x_4^2 \leq 1$$

If we generate random points in the four dimensional cubic region $[-1,1] \times [-1,1] \times [-1,1] \times [-1,1]$ which contains the unit sphere, then an estimate of the volume of the sphere

$$\frac{\text{Volume of Sphere}}{\text{Volume of Cube}} \approx \frac{\text{No. Points in Sphere}}{\text{No. of Points in Cube}}$$

By symmetry we get the same result if work in a single quadrant, say $[0,1] \times [0,1] \times [0,1] \times [0,1]$.

Here is the calculation in Scilab. First we write a function `monte4d` to do the calculation:

```
function v = monte4d(n)
  k = 0              \\ count of number of points in sphere
  for i = 1:n
    x = rand(1,4)        \\ x = point in unit cube
    if (norm(x) <= 1)    \\ point x lies in sphere
      k = k+1
    end
  end
  v = 16*k/n             \\ 16 quadrants!
endfunction

-->monte4d(100000)
 ans  =

    4.91392
```

By the way, the exact answer is $\pi^2/2 = 4.9348$

5.3 Multiple Integrals

Generalizing the formula for Monte-Carlo integration in one dimension, we have a formula for Monte-Carlo integration in any number of dimensions: if

$$I = \int_\Omega f \, dV$$

then

$$I \approx \text{Volume of } \Omega \times \text{Average Value of } f \text{ in } \Omega$$

For one-dimensional integration problems Monte-Carlo integration is quite inefficient. For high-dimensional integration it is a useful technique. The reasons for this are twofold:

1. The fact that error is proportional to $1/\sqrt{n}$ does not depend on the dimension. In other words it performs just as well in high dimensions as in one dimension.

2. It can easily handle regions with irregular boundaries. The method used in the previous section to estimate volumes can easily be adapted to Monte-Carlo integration over any region.

Example

We will estimate the integral

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

where Ω is the disk

$$\left(x-\frac{1}{2}\right)^2 + \left(y-\frac{1}{2}\right)^2 \le \frac{1}{4}$$

Since the disk Ω is contained within the square $[0,1]\times[0,1]$, we can generate x and y as uniform $[0,1]$ random numbers, and keep those which lie in the disk Ω.

```
function ii = monte2da(n)
  k = 0              // count no. of points in disk
  sumf = 0           // keep running sum of function values
  while (k < n)      // keep going until we get n points
    x = rand(1,1)
    y = rand(1,1)
    if ((x-0.5)^2 + (y-0.5)^2 <= 0.25) then  // (x,y) is in disk
      k = k + 1                              // increment count
      sumf = sumf + sin(sqrt(log(x+y+1)))    // increment sumf
    end
  end
  ii = (%pi/4)*(sumf/n)     // %pi/4 = volume of disk
endfunction

-->monte2da(100000)
 ans  =

    0.5679196
```

Example

In the Monte-Carlo approximation

$$I \approx \text{Volume of } \Omega \times \text{Average Value of } f \text{ in } \Omega$$

we can estimate the volume of the region Ω at the same time as we estimate the average of the function f.

We generate points in a volume V – usually rectangular – containing Ω. If we generate n points in V of which k lie in the region Ω then

$$\text{Volume } \Omega \approx \frac{k}{n} \text{Volume } V$$

Since

$$\text{Average Value of } f \text{ in } \Omega \approx \frac{1}{k} \sum f$$

the k's cancel and we have

$$I \approx \text{Volume } V \times \frac{1}{n} \sum f.$$

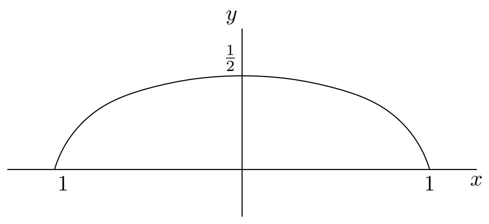
where the sum is over the points lying in the region Ω.

We will now evaluate the integral

$$I = \iint_{\Omega} y \, dx dy$$

over the semi-elliptical region Ω given by

$$x^2 + 4y^2 \leq 1, \quad y \geq 0$$



Although there is a simple formula for the area of an ellipse, we will use Monte-Carlo to estimate the area as we perform the integration. Since the region Ω is contained within the rectangle $[-1, 1] \times [0, 1/2]$, we generate x and y as uniform $[-1, 1]$ and uniform $[0, 1/2]$ random numbers respectively. We will generate `n` random numbers, and use those lying in Ω to form the sum of the function. Note that the rectangular region has area $= 1$.

```
function ii = monte2db(n)
  sumf = 0            // keep running sum of function values
  for i = 1:n
    x = 2*rand(1,1) - 1       // x in [-1,1]
    y = rand(1,1)/2           // y in [0,1/2]
    if (x^2 + 4*y^2 <= 1)     // point lies in region
      sumf = sumf + y         // increment sumf
    end
  end
  ii = sumf/n                 // Volume = 1
endfunction

-->monte2db(100000)
 ans  =

    0.1668539
```

The exact value is 1/6.

6 Simulation

6.1 Two Dice

Simulation using random numbers is a technique for estimating probabilities of composite events. The idea is to use a random number generator to simulate the corresponding elementary events and to count frequencies of the composite events we are interested in. Using computer simulation is much faster than actually performing the random experiment (e.g. throwing dice and recording the results) a large number of times.

This can be illustrated by examples.

The first example is an easy one which will show the effective use of Scilab. Suppose I toss two unbiased dice. What is the probability that the sum of the numbers showing is less than or equal to 4? A simple counting argument shows that the answer is $6/36 = .16667$.

We will simulate 10 tosses of the dice. The number showing on each die is an integer uniformly distributed between 1 and 6. We saw how to generate such random numbers in §3.3. We will represent the results of our simulation by a 2×10 array.

```
-->x = floor(6*rand(2,10) + 1)
 x  =

!   2.    4.    1.    6.    2.    3.    6.    3.    5.    5. !
!   2.    1.    1.    1.    5.    6.    6.    5.    2.    6. !
```

Now we can calculate the sum of the two dice, noting that `sum(x,'r')` sums over the *columns* of the matrix `x` while `sum(x,'c')` sums over the *rows*:

```
-->s = sum(x,'r')
 s  =

!   4.     5.     2.     7.     7.     9.     12.     8.     7.     11. !
```

What we need now is to count how many of these are ≤ 4. We can use **boolean arrays** to do this efficiently:

```
-->ss = (s <= 4)
 ss  =

! T F T F F F F F F F !
```

Here `T` and `F` stand for true and false, and this array just tells us whether the corresponding elements of `s` are ≤ 4. The boolean elements `T` and `F` also have the numerical values 1 and 0 respectively, so we can find the number of cases in which the sum of the two dice is ≤ 4 by taking the sum of this array.

```
-->xx = sum(ss)
 xx  =

    2.
```

So in 2 of our 10 simulations, the sum was less than or equal to four.

We can write a function to do this:

```
function y = dice2(n)
  x = floor(6*rand(2,n) + 1)
  s = sum(x, 'r')
  ss = (s <= 4)
  y = sum(ss)/n
endfunction

-->dice2(100000)
 ans  =

    0.16727
```

6.2 Estimating Errors

The discussion of errors in Monte-Carlo integration is also applicable to general simulation problems. The expected error in n trials is

$$\text{Error} = \frac{\sigma}{\sqrt{n}}$$

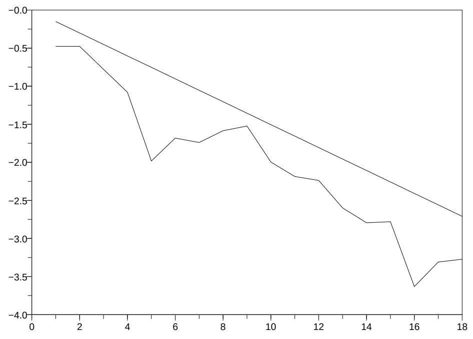
Unfortunately we do not usually know the value of σ which is the variance of the probability we are estimating. It is usually possible to get an estimate of σ from the simulation itself, but we will mainly be concerned with the $n^{-1/2}$ factor which determines how the error depends on n.

Let us look at the error in our dice problem for $2, 4, \ldots, 2^{18}$ trials:

```
-->dn = zeros(1,18);

-->for k = 1:18
-->  dn(k) = dice2(2^k);
-->end

-->dn

 dn  =

         column 1 to 7
!   0.5    0.5    0.    0.25    0.15625    0.1875    0.1484375 !

         column  8 to 11
!   0.140625    0.1367188    0.1767578    0.1601562 !

         column 12 to 15
!   0.1608887    0.1691895    0.1682739    0.1650085 !

         column 16 to 18
!   0.1669006    0.1661758    0.1661339 !
```

According to the theory the error should be proportional to $n^{-1/2}$. Simply superimposing a line of slope -1/2 should indicate whether the error really is decreasing as theorized.

```
-->err = abs(dn-1/6);

-->n = 2 .^(1:18);

-->ee = n.^{-1/2};

-->plot2d(log10(err))

-->plot2d(log10(ee))
```



6.3 Two Loaded Dice

Suppose now that our dice are loaded, so that the probabilities are:

Outcome	1	2	3	4	5	6
Probability	.20	.14	.22	.16	.17	.11
Cumulative	.20	.34	.56	.72	.89	1.00

How can we simulate tossing these dice? Suppose x is uniformly distributed on $[0, 1]$. If $x \in [0, .20]$ we assign the value 1, if $x \in [.20, .34]$ we assign the value 2 etc. A Scilab function to assign these probabilities is easy to write using `if-then-else` statements, but we can do better by using boolean arrays:

```
function y = ldice(x)
  x1 = (x >= 0.20)
  x2 = (x >= 0.34)
  x3 = (x >= 0.56)
  x4 = (x >= 0.72)
  x5 = (x >= 0.89)
  y = 1*x1 + 1*x2 + 1*x3 + 1*x4 + 1*x5  + 1
endfunction
```

In the function above, each of `x1` to `x5` is a boolean vector, we multiply `1` to convert to a numerical value and the add the results.

```
-->x = rand(1,8)
 x  =
         column 1 to 4
!   0.0976644    0.8918166    0.1762568    0.0862309 !

         column 5 to 8
!   0.2136821    0.9119316    0.5557921    0.5549552 !
```

```
-->ldice(x)
 ans  =

!   1.    6.    1.    1.    2.    6.    3.    3. !
```

Our function to estimate the probability that the sum of the two dice is less than or equal to 4 is nearly the same as the previous example:

```
function y = ldice2(n)
  x = ldice(rand(2,n))
  s = sum(x, 'r')
  ss = (s <= 4)
  y = sum(ss)/n
endfunction

-->ldice2(100000)
 ans  =

    0.20316
```

You might like to calculate the exact answer and compare.

6.4 The Birthday Problem

Suppose we have N people in the room. What is the probability that (at least) two people share the same birthday.

To solve this by simulation we can proceed as follows:

1. Generate N random birthdays.
2. Check if two coincide.

Generating the random birthdays is easy, just generate a random vector of integers in the range 1 to 365 (we will ignore leap years). To check whether such a vector contains two numbers the same, we first sort the birthdays, and then only have to check whether neighbouring components of the vector are equal.

```
function p = birthdays(n, trials)
  k = 0
  for i = 1:trials
    bs = floor(365 * rand(1,n) + 1)
    bs = sort(bs)
    for j = 1:(n-1)
      if (bs(j) == bs(j+1))    // we have found a match
        k = k+1                // increment count
        break                  // break out of for loop
      end
    end
  end
  p = k/trials
endfunction
```

Only 23 people are needed for the probability that two have the same birthday to be greater than 0.5. Let us check this:

```
-->birthdays(22, 100000)
 ans  =

    0.47559

-->birthdays(23, 100000)
 ans  =

    0.50831

-->birthdays(24, 100000)
 ans  =

    0.53805
```

For $N = 100$ it is almost certain that two people will share a birthday:

```
-->birthdays(100, 100000)
 ans  =

    1.
```

7 Appendix – Boolean Matrices

The boolean values are `%t`, printed `T`, for **true** and `%f`, printed `F`, for **false**. Matrices, and particularly vectors, of boolean values are often useful in simulation.

7.1 Comparison and Logical Operators

Boolean matrices are usually constructed by applying comparison operators:

`==`	equal
`~=`	not equal
`<`	less than
`>`	greater than
`<=`	less than or equal to
`>=`	greater than or equal to

to real matrices.

```
-->a=rand(4,4)
 a  =

!   0.2806498    0.1121355    0.8415518    0.1138360 !
!   0.1280058    0.6856896    0.4062025    0.1998338 !
!   0.7783129    0.1531217    0.4094825    0.5618661 !
!   0.2119030    0.6970851    0.8784126    0.5896177 !

-->bb = a > 0.4
 bb  =

! F F T F !
! F T T F !
! T F T T !
! F T T T !
```

The usual matrix operations, `+`, `-` and `*` are undefined for boolean matrices. The logical operators

`&`	and
`|`	or
`~`	not

apply element by element to boolean matrices.

```
-->~bb
 ans  =

! T T F T !
! T F F T !
! F T F F !
! T F F F !
```

7.2 Real and Boolean Matrices

Boolean matrices may be converted to real 0-1 matrices, 0 = false, 1 = true, with the `bool2s` command:

```
-->bool2s(bb)
 ans  =

!   0.    0.    1.    0. !
!   0.    1.    1.    0. !
!   1.    0.    1.    1. !
!   0.    1.    1.    1. !

-->sum(ans)
 ans  =

    9.
```

We could have got the same result with `sum(bb)`, though this would seem to contradict the rule that we can't add booleans. However this can be quite handy for counting the number of components of a vector or matrix satisfying some condition, e.g.

```
-->sum(a > 0.4)
 ans  =

    9.
```

Scilab allows arithmetic operations where one operand is boolean and the other a number. In this case the boolean values are converted to 0-1 values. For example;

```
-->bb
 bb  =

! F F T F !
! F T T F !
! T F T T !
! F T T T !

-->bb*4
 ans  =
!   0.    0.    4.    0. !
!   0.    4.    4.    0. !
!   4.    0.    4.    4. !
!   0.    4.    4.    4. !
```

```
-->bb-1
 ans  =

! - 1.  - 1.    0.  - 1. !
! - 1.    0.    0.  - 1. !
!   0.  - 1.    0.    0. !
! - 1.    0.    0.    0. !
```

7.3 find

The `find` operator returns the indices of the true components of boolean vector or matrix. It is most useful for vectors, so we will concentrate on those:

```
-->a = rand(1,12)
 a  =
         column 1 to 4
!   0.5878720    0.4829179    0.2232865    0.8400886 !

         column 5 to 8
!   0.1205996    0.2855364    0.8607515    0.8494102 !

         column  9 to 12
!   0.5257061    0.9931210    0.6488563    0.9923191 !

-->bb = a > 0.4
 bb  =

! T T F T F F T T T T T T !

-->find(bb)
 ans  =

!   1.    2.    4.    7.    8.    9.    10.    11.    12. !
```

We can omit the intermediate step of constructing the boolean vector:

```
-->ii = find(a > 0.4)
 ii  =

!   1.    2.    4.    7.    8.    9.    10.    11.    12. !
```

The vector of indices constructed in this way can be used to extract the corresponding components of the vector, in this case all the components greater than 0.4:

```
-->a(ii)
 ans  =
        column 1 to 4
!   0.5878720    0.4829179    0.8400886    0.8607515 !

        column 5 to 8
!   0.8494102    0.5257061    0.9931210    0.6488563 !

        column 9
!   0.9923191 !
```