Monte Carlo Method: Sampling

John Burkardt (ARC/ICAM)
Virginia Tech

....

Math/CS 4414:
"The Monte Carlo Method: SAMPLING"
https://people.sc.fsu.edu/∼jburkardt/presentations/monte_carlo_sampling.pdf

....

ARC: Advanced Research Computing
ICAM: Interdisciplinary Center for Applied Mathematics

02-04-06 November 2009

Monte Carlo Method: Sampling

- **Overview**
- Can We Compute Randomness?
- MATLAB's RAND Generator
- Geometric Sampling
- Sequential Sampling
- Sampling for Optimization
- Sampling a Nonuniform Distribution
- Estimating Integrals

Overview

This is the second set of talks on the Monte Carlo Method (MCM).

This talk considers the Monte Carlo Method (MCM) as a way of *sampling*.

We are presumably trying to analyze a very large set **X**, but we cannot do so in a simple closed form.

If the set of outcomes is discrete, then perhaps it is very large. If it is continuous, perhaps we need to carry out an integration of a function **f(x)** over **X**, but there is no closed form for the integral or no closed form for the function or our integration region is an unusual shape.

Overview

We might be trying to integrate over a strange domain, perhaps a region bounded by a curve in the plane, or the surface of some shape in 3D.

We might be looking at problems in which each $\mathbf{x}$ represents a configuration of a physical system. The energy of the configuration and the temperature of the system determine the probability of that configuration.

Monte Carlo Method: Sampling

Can We Compute Randomness? (Short Answer: No)

Before we go much further, we should ask how it's possible for the computer to create a string of random numbers.

Of course, it isn't possible. The computer is carrying out some well defined procedure, one that can be predicted or repeated. What comes out of a computer program is called a stream of **pseudorandom numbers**.

We don't care about the philosophy behind this distinction, and we'll go back to calling them random numbers. But even so, it's interesting to ask how it's possible to create a string of pseudorandom numbers?

Can We Compute Randomness? - The LCG

Just like with cards, the basic idea is simply to shuffle the numbers you have.

Let's assume we're interested in positive integers. The following formula is known as a linear congruential generator or **LCG**.

$$x_{k+1} = (a * x_k + c) \mod m$$

What's happening is we're drawing the line y = a * x + c "forever", but using the mod function (like wrap-around in a video game) to bring the line back into the square [0,m] x [0,m]. By doing so, we've induced a map on the integers 0 through m which, if we've chosen a, c and m carefully, will do an almost perfect shuffle.

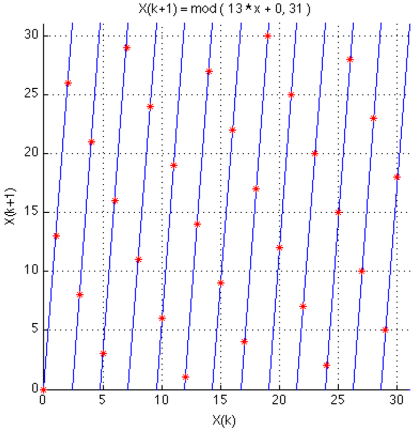
Can We Compute Randomness? - LCG Mapping

Consider the simple map $x_{k+1} = (13 * x_k + 0) \mod 31$

```
    *13    mod 31
 0 ->   0 ->  0
 1 ->  13 -> 13
 2 ->  26 -> 26
 3 ->  39 ->  8
 4 ->  52 -> 21
 5 ->  65 ->  3
 6 ->  78 -> 16
 7 ->  91 -> 29
 8 -> 104 -> 11
 9 -> 117 -> 24
10 -> 130 ->  6
 ...
```

Can We Compute Randomness? - LCG Image

1->13->14->27->10->6->16->22->7->29->5->3->8->11->19->30
->18->17->4->21->25->15->9->24->2->26->28->23->20->12->1



VirginiaTech

Can We Compute Randomness? - PowerMod

You might think that in order to compute the 23rd number in the sequence, you must start with the first one and transform it, successively, 22 times.

Our formula becomes

$$x_{23} = \mathrm{mod}(13 * \mathrm{mod}(13 * ...\mathrm{mod}(13 * x_1), ..., 31), 31)$$

but this can be packed up into

$$x_{23} = \mathrm{mod}(13^{22} * x_1, 31)$$

Now if we can compute $\mathrm{mod}(13^{22}, 31)$ quickly, most of our work is done. This can be done if you have a **PowerMod()** function (and if the powers get big, there's no other way!)

MATLAB doesn't have one built in, but it's not too hard to write. This way, we discover that $\mathrm{mod}(13^{22}, 31) = 9$, so

$$x_{23} = \mathrm{mod}(9 * x_1, 31) = 9.$$

Can We Compute Randomness? - Integers to Reals

Our LCG gives us a method of generating what looks like a random sequence of **integers** between 1 and 30.

Can we get uniform random **reals** in the open interval (0,1)?

(Uniform random number generators traditionally do NOT return a value of 0.0 or 1.0, but always something inbetween.)

We could compute the next element of our LCG sequence, and divide it by 31. The user would see a real value that jumped around in the right way. If the user wanted another value, we would need to remember where we were in the LCG sequence.

Can We Compute Randomness? - The SEED

This procedure of generating a hidden sequence of integers, and computing the uniform real values from these integers, is very old and very common.

The current value of the integer in the LCG is called the **random number seed**. If you know the current value of the seed, you can determine all subsequent steps of the LCG, and all outputs of the uniform random values.

Most random number generators let you find out the current value of the seed. They also let you reset the seed to a particular value. This allows you to repeat a random number sequence, or to guaranteee that you're starting a new one.

Can We Compute Randomness? - A better LCG

Of course a random number generator that shuffles only 30 values won't be of much use to us.

A more ambitious LCG has the form

$$\begin{aligned} SEED =& (16807 * SEED + 0) \mod 2147483647 \\ R =& SEED/2147483647 \end{aligned}$$

This is the random number generator that was used in MATLAB until version 5. It perfectly shuffles the integers from 1 to 2,147,483,646, and then repeats itself.

Can We Compute Randomness? - LCG is just a start

Modern random number generators have become more sophisticated than the LCG approach.

They need to do this partly because 2 billion random numbers is no longer enough for some of the Monte Carlo simulations that are carried out!

These days, instead of a single integer seed, the random number generator you use might instead have a complicated "state" vector, containing 100 or so integers, which it uses to vastly increase the amount of shuffling that goes on.

However, if you understand how the LCG works, you should be able to understand the basic techniques involved in generating what looks like randomness out of a simple formula!

Monte Carlo Method: Sampling

MATLAB's RAND Generator - Usage

MATLAB's main random number generator is **RAND()**:

```
a = rand ( )         <-- a scalar value;
b = rand ( 5, 1 )    <-- a column vector;
c = rand ( 1, 5 )    <-- a row vector;
d = rand ( 3, 4 )    <-- a matrix;
e = rand ( 5 )       <-- a 5x5 matrix;
```

MATLAB's RAND Generator - Internal State

The random number generator must have an internal seed or state so that it can generate the next random value. When you start up the MATLAB program, this internal state is always initialized to the same value. So every time you start up MATLAB and ask for 5 random numbers, this is what you will get:

```
  >> r = rand(5,1)


r =
    0.8147
    0.9058
    0.1270
    0.9134
    0.6324
>>
```

VirginiaTech

MATLAB's RAND Generator - 3 Generators

Perhaps you want your program to use a *different* random sequence each time; or perhaps you want to repeat the same random sequence several times.

You can do this, because MATLAB allows you to get or set the internal state.

This is when you find out MATLAB has *three* built in random number generators, all hiding inside the **rand()** function:

- the LCG generator, default for versions 1 to 4;
- the Marsaglia generator, default for versions 5 to 7.3;
- the Mersenne twister, the default since version 7.4.

MATLAB's RAND Generator - Get the State

To get the current state of any of the three generators:

- **s = rand ('seed')**, the LCG;
- **s = rand ('state')**, the Marsaglia generator;
- **s = rand ('twister')**, the Mersenne twister.

The value **s** that is returned is an integer that describes the current internal state.

MATLAB's RAND Generator - Set the State

To choose the random number generator, and set its state to a saved value (or to an arbitrary integer value you choose):

- **rand ('seed', s)**, the LCG;
- **rand ('state', s)**, the Marsaglia generator;
- **rand ('twister', s)**, the Mersenne twister.

Once you have chosen a generator and set the seed, you can start calling **rand** again.

MATLAB's RAND Generator - Example

In the following example, $\mathbf{a} = \mathbf{e}$ and $\mathbf{b} = \mathbf{f}$.

```
rand ( 'seed', 12345 )
a = rand ( )
b = rand ( )
rand ( 'seed', 54321 )
c = rand ( )
rand ( 'twister', 12345 )
d = rand ( )
rand ( 'seed', 12345 )
e = rand ( )
f = rand ( )
rand ( 'state', 12345 )
g = rand ( )
```

MATLAB's RAND Generator - Other Functions

MATLAB includes other functions useful for Monte Carlo work:

- **randn()** returns normally distributed random values;
- **randperm()** returns a random permutation (card shuffling);
- **random('name', x, params)** random values from 23 PDF's;
- **pdf('name', x, params)** evaluates the PDF;
- **cdf('name', x, params)** evaluates the CDF;
- **icdf('name', x, params)** evaluates the inverse CDF;
- **bar(x,y)** makes bar graphs of data;
- **hist(x,bin_num)** makes a histogram of scattered data.

Monte Carlo Method: Sampling

Geometric Sampling - RAND samples [0,1]

We can think of the output of **rand()**, or any stream of uniform random numbers, as a method of "sampling" the unit interval.

We would expect that if we plotted the points along a line, they would be roughly evenly distributed, like a lot of paint drops spattered from far away.

Random numbers are actually a very good way of exploring or sampling many geometrical regions. Since we have a good source of random numbers for the unit interval, it's worth while to think about how we could modify such numbers to sample these other regions.

Geometric Sampling - Sampling [A,B]

As a very simple case, suppose we need uniform random numbers, but we want them in the interval **[A,B]**.

We can still use **rand()**, but now we have to shift and scale. The numbers have to be shifted by **A**, and scaled by **B-A**:

$$\begin{aligned} R &= \mathrm{rand}(); \\ S &= A + (B - A) * R; \end{aligned}$$

or we can do this for hundreds of values at once:

$$\begin{aligned} R &= \mathrm{rand}(10000, 1); \\ S &= A + (B - A) * R; \end{aligned}$$

Geometric Sampling - Sampling the Unit Square

We might need uniform random numbers in the unit square.

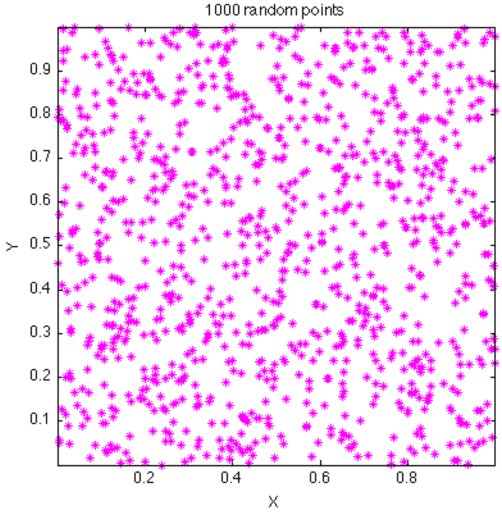
We could compute an **x** and **y** value separately for each point

$$x = \text{rand}();$$
$$y = \text{rand}();$$

or we can do this for hundreds of values at once:

$$xy = \text{rand}(2, 10000);$$

Geometric Sampling - Sampling the Unit Square



Geometric Sampling - The Unit Circle

Suppose we need to evenly sample points in the unit circle, center 0 and radius 1?

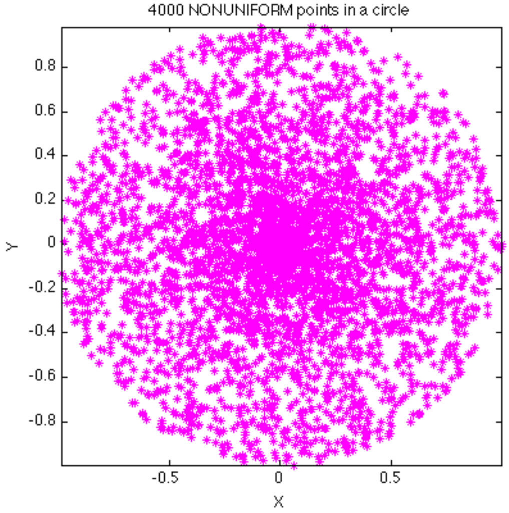
Without thinking very hard, we could try to choose a radius from 0 to 1, and then choose an angle between 0 and 2π using the "stretching" method:

$$\begin{aligned} r &= \mathrm{rand}(); \\ t &= 2 * \pi * \mathrm{rand}(); \end{aligned}$$

but although this seems "random" it is not a **uniform** way to sample the circle. More points in a circle have a big radius than a small one, so choosing the radius uniformly actually is the wrong thing to do!

VirginiaTech

Geometric Sampling - The Unit Circle



Geometric Sampling - The Unit Circle

Can we fix the problem? And can we understand why the fix works?

The problem is that when we choose the radius to vary uniformly, we're saying there are the same number of points at every radius **r**. But of course, there aren't. If we double **r**, the area of the circle increases by a factor of 4. So to measure area uniformly, we need r^2 to be uniformly random, in other words, we want to set **r** to the *square root* of a uniform random number.

$$r = \sqrt{\mathrm{rand}()};$$
$$t = 2 * \pi * \mathrm{rand}();$$

Geometric Sampling - The Unit Circle by Rejection

Here's another idea that has a chance of working.

Suppose we sample the square that contains the circle, and then only plot the points that are in the circle?

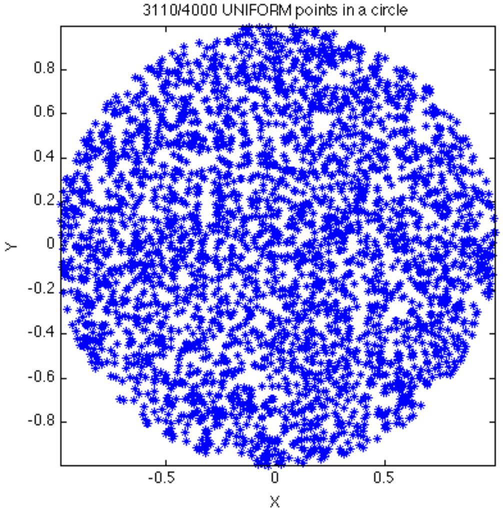
Now we're sure to get uniformly distributed points (very good!)

We'll also have to *reject* many points we compute (bad).

```
x = -1 + 2 * rand ( );
y = -1 + 2 * rand ( );
i = find ( x.^2 + y.^2 < 1 )
plot ( x(i),  y(i), 'b*' )
```

Notice that I was able to use 3110 points out of 4000 generated. Is there any significance in this ratio?

Geometric Sampling - The Unit Circle by Rejection



Geometric Sampling - The Surface of the Unit Sphere

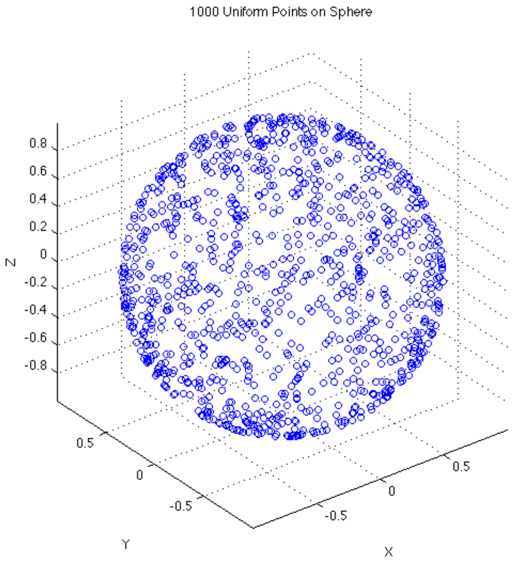
Another important shape to sample is the **surface** of the unit sphere. We can do this, but we'll have to cheat slightly and use *normal* random numbers rather than uniform ones.

It turns out that if you want a point on the surface of the unit sphere, you can just make a vector of 3 normal random numbers, and scale by its length:

```
xyz = randn ( 3, 1000 );
for j = 1 : 1000
  xyz(:,j) = xyz(:,j) ./ norm ( xyz(:,j) );
end
scatter3 ( xyz(1,:), xyz(2,:), xyz(3,:) )
```

This will also work for higher dimensional spheres!

Geometric Sampling - The Surface of the Unit Sphere



iaTech

Monte Carlo Method: Sampling

Sequential Sampling - Repeating a Random Process

If a process that includes randomness can be repeated many times, then we obtain a finite sequence of outcomes, $r_1, r_2, r_3, \ldots, r_M$.

To try to understand the PDF that generated these values, we can use histograms, or compute statistical properties.

As the sequence size M increases, we naturally expect a more accurate estimate. Can we say how fast the accuracy improves?

Sequential Sampling - Average and Variance

If a sequence **R** of random numbers is generated from a particular PDF, then the average and variance of the sequence should approximate the average and variance of the PDF.

The average and variance of the PDF are determined by weighted summation (or integration in the continuous case).

The average and variance of the sequence are determined by:

$$\mathrm{ave}(R) = \frac{1}{M}\sum_{j=1}^{M} r_j$$

$$\mathrm{var}(R) = \frac{1}{M}\sum_{j=1}^{M}(r_j - \mathrm{ave}(R))^2$$

VirginiaTech

Sequential Sampling - Average Formulas

Compare averages for sequence and PDF, discrete or continuous:

$$\text{ave(M sequence)} = \frac{1}{M} \sum_{j=1}^{M} r_j$$

$$\text{ave(discrete pdf)} = \sum_{i=1}^{N} p(r_i) r_i$$

$$\text{ave(continuous pdf)} = \int_a^b p(r) r \, dr$$

The expression $\frac{1}{M}$ in the sequence formula is similar to the probability $p()$. If a value r_j occurs **K** times in the sequence, we could replace them by the expression $\frac{K}{M} r_j$, which now looks like an estimate for the probability (PDF) of that value!

Sequential Sampling - Estimating the PDF Average

So given a sequence that comes from some PDF, we can compute its average and variance.

If we think of this as an effort to estimate the actual average and variance of the underlying PDF, we have just started to use the **Monte Carlo** method.

The idea is simple:

$$\lim_{M\to\infty} \mathrm{ave}(\mathrm{M\,sequence}) = \mathrm{ave}(\mathrm{pdf})$$
$$\lim_{M\to\infty} \mathrm{var}(\mathrm{M\,sequence}) = \mathrm{var}(\mathrm{pdf})$$

These statements are true (in a probabilistic sense) for "reasonable" PDF's.

VirginiaTech

Sequential Sampling - Convergence of Estimates

In particular, the **Law of Large Numbers** states that, as $M \to \infty$, the average of samples from a PDF will converge to the average of the PDF.

The **Central Limit Theorem** tells us that, as $M \to \infty$, the error in the estimated average will tend to be $O(\frac{\sigma^2}{\sqrt{M}})$, where σ is the standard deviation and σ^2 is the variance of the PDF.

This means that convergence is faster if the variance is small, and convergence is pretty slow as **M** increases. Roughly speaking, to make the error go down by a factor of 10, we have to use 100 times as many points!

Sequential Sampling - Example Program

Let's generate sequences from a uniform random generator.

The exact PDF average is 1/2, the exact variance is 1/12.

```
m = 1;
for m_log = 0 : 7
  r = rand ( m, 1 );
  r_ave = sum ( r ) / m;
  r_var = var ( r );
  fprintf ( 1, '  %7d  %f  %f\n', m, r_ave, r_var );
  m = m * 10;
end
```

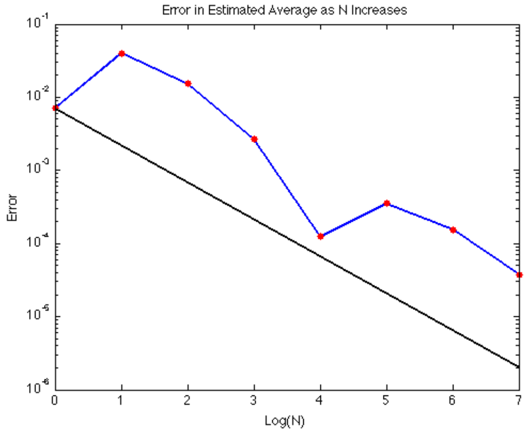
Sequential Sampling - Example Results

Here is are the results for MATLAB's **rand()** function:

```
       M    Ave         Ave-1/2    Var          Var-1/12

       1  0.492922  -7.08e-03  0.000000  -8.33e-02
      10  0.460068  -3.99e-02  0.072765  -1.06e-02
     100  0.515335   1.53e-02  0.081740  -1.59e-03
    1000  0.497364  -2.64e-03  0.083231  -1.03e-04
   10000  0.499876  -1.24e-04  0.084185   8.52e-04
  100000  0.500352   3.52e-04  0.083267  -6.61e-05
 1000000  0.500154   1.54e-04  0.083434   1.00e-04
10000000  0.499962  -3.77e-05  0.083304  -2.94e-05
```

Sequential Sampling - Example Plot

Here we plot the error in the estimated average.
The error (blue) decreases like the function $1/\sqrt{M}$ (black).
(The **slope** is the important thing in this picture)



Sequential Sampling - Averages Can Be Integrals

Our example, estimating the average of the uniform random PDF, may seem like something very contrived and useless.

But actually, when we estimated the average of the uniform distribution, we estimated the value of the integral of a function:

$$\int_0^1 p(x) x \, dx = \int_0^1 x \, dx \approx \frac{1}{M} \sum_{j=1}^{M} x_i$$

and we have some idea of how the error behaves for this approximation.

These ideas, along with our ability to sample various geometric regions, will become very useful for dealing with integration of difficult functions over unusual domains.

VirginiaTech

Monte Carlo Method: Sampling

Optimization

The optimization problem is given a region **R** and a scalar-valued function $f(x)$, and is required to find a point $x*$ at which $f(x)$ attains its global minimum.

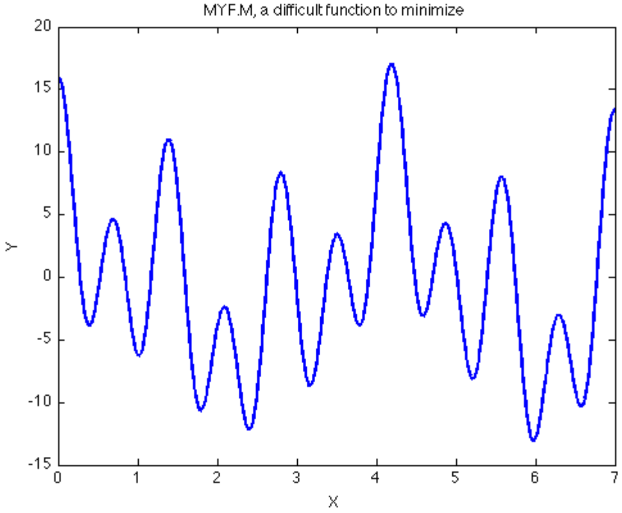
Unless $f(x)$ is a very simple function, it is liable to have many local minima. Most minimization algorithms can be "trapped" by a local minimum. In general, the only way to guarantee that you have found the global minimum is to look *everywhere*!

Obviously, we can't do that. But, unless $f(x)$ is a really awful function, each minimum, including the global one, will be surrounded by a "basin of attraction", that is, an area with the property that if a minimization algorithm can get this close, it will then roll right into that minimum. So we don't really have to look at every point in the space, just at a lot of different regions.

Optimization: A Nasty Function

```
function f = myf ( x )

%% MYF evaluates a function to be minimized over 0 <= x <= 7.
%
  f =     cos (       x ) ...
    + 5 * cos ( 1.6 * x ) ...
    - 2 * cos ( 2.0 * x ) ...
    + 5 * cos ( 4.5 * x ) ...
    + 7 * cos ( 9.0 * x );

  return
end
```

Optimization: Image of the Nasty Function



Optimization: Strategy for the Nasty Function

So here's our strategy to minimize the function.

1. Initialize **xbest** $= 0$ and **fbest** $= f(xbest)$;
2. We'll hit a golf ball onto the green, (choose a random starting point in [0,7]).
3. Let the ball roll into the cup (run a minimization algorithm, which will find a local minimizer $x*$.)
4. If the value of $f(x*)$ is lower than **fbest**, set **xbest** $= x*$, **fbest** $= f(x*)$.
5. If we've hit 500 balls, or if **fbest** hasn't decreased in the last 20 tries, terminate.

Optimization: Why Monte Carlo?

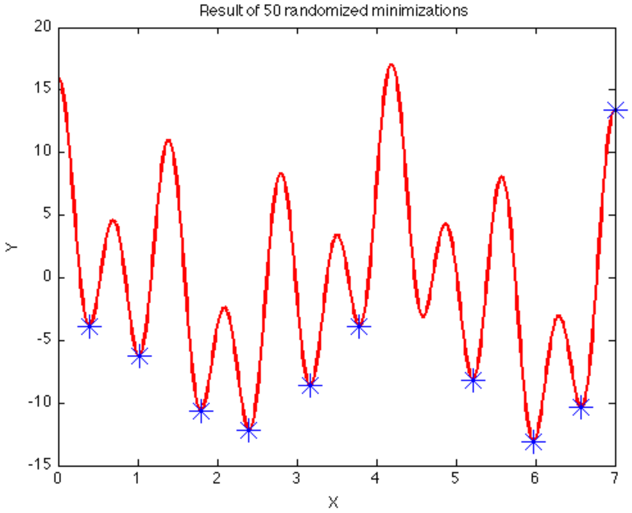
It would be fair to ask why we bother randomizing this algorithm. Why not just divide up the region into 500 equally spaced points; then you will guarantee that you sample the space just as well (even better, because you know there can't be any large gaps).

There are other reasons for wanting to randomize the algorithm, especially if we look at more complicated geometry or constraints, but the point that is raised here is a fair one. However, notice that the randomized algorithm can stop at any time; moreover, it can continue as long as we want. There is no particular "scale" to the randomized algorithm.

But for the evenly spaced grid, we must carry out all 500 steps in order to complete the grid. If we want more accuracy, we have to start all over again, and carefully choose a grid size that will avoid repeating values we already investigated.

Optimization: Results for the Nasty Function

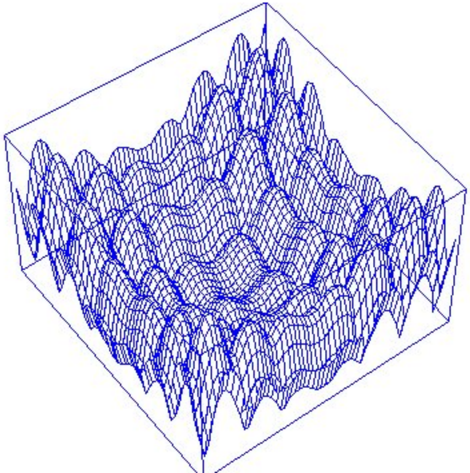
Here is the result of 50 random initial values.
Note that we actually find a local minimum at 7!



niaTech

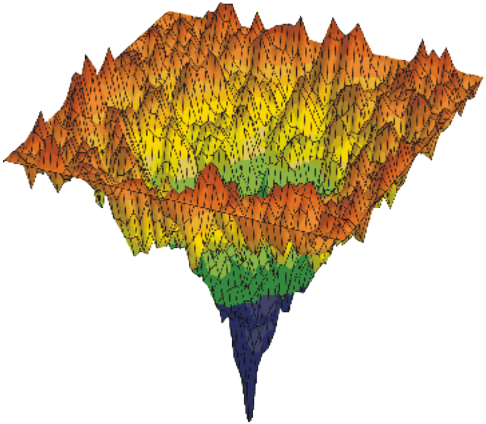
Optimization: Multiple Dimensions

Here is an optimization problem in 2D that shows how quickly things get difficult. It's called the "Alpine Problem", and we can think of the Monte Carlo approach as dropping skiers off all over these slopes. They all slide downhill and report their location.



Optimization: Multiple Dimensions

A protein starts as a linear string and folds into a shape with minimal energy. The path to the minimal energy may be bumpy! We can use the Monte Carlo approach to select many initial foldings of the protein, some of which may then evolve into the lowest energy state.



Monte Carlo Method: Sampling

Nonuniform Distributions - Normal Distribution

MATLAB has a built in generator for normal random variables. These values will have zero mean, and unit variance:

$$R = \mathrm{randn}();$$

and, as with **rand()**, you can create row and column vectors and arrays as well. Instead of a seed, **randn()** has a "state". If you want to restart a sequence, you save the state and then reset it:

$$\begin{aligned} \mathrm{saved\_state} =& \mathrm{randn}(\text{`state`}); \\ R1 =& \mathrm{randn}(); \\ & \mathrm{randn}(\text{`state`}, \mathrm{saved\_state}); \\ R2 =& \mathrm{randn}(); \end{aligned}$$

R1 and **R2** will be the same.

Nonuniform Distributions - Normal Distribution

It is always useful to have alternate ways of doing something. Recall that we said that the sum of uniform random variables tends to behave like a normal random variable.

We don't know how the normal values are computed, so maybe it would be faster to compute 10 uniforms and add them instead.

This is a fine idea, and in fact there are programs that use this approximation.

Nonuniform Distributions - Normal Distribution

However, it takes a little tuning to get this right. Let's assume we are going to use our trick to try to approximate a normal random variable with mean 0 and variance 1.

Can you see why our scheme is **not** going to have the right mean? Can you see how to fix it?

Our suggested approximation to a normal random number is now

$$S = \text{sum}(\text{rand}(10, 1)) - 5;$$

We've gotten the mean right, but is the variance going to be OK? (As it turns out, yes...)

Now compare the timings to compute 100,000 normal values using **randn()** and using our approximation.

Nonuniform Distributions - Normal Distribution

```
n = 1;
for logn = 0 : 6
  tic;
  r1 = randn ( n, 1 );
  t1 = toc;
  tic;
  r2 = sum ( rand ( n, 10 ), 2 );
  t2 = toc;
  fprintf ( 1, '  %10d  %f  %f\n', n, t1, t2 );
  n = n * 10;
end
```

VirginiaTech

Nonuniform Distributions - Normal Distribution

The results suggest that RANDN() is VERY efficient!
If RANDN takes as much time as RAND, that would explain our factor of 10.

```
      N   RANDN()    Sum RAND

      1   0.000043   0.000621
     10   0.000037   0.000056
    100   0.000070   0.000450
   1000   0.000120   0.001624
  10000   0.001108   0.017840
 100000   0.011241   0.151950
1000000   0.068690   1.021350
```

Nonuniform Distributions - Normal Distribution

```
n = 1;
for logn = 0 : 6
  r = zeros(n,1);  <-- Set aside space in advance!
  tic;
  for i = 1 : n
    r(i) = randn ( 1, 1 );
  end
  t1 = toc;
  tic
  for i = 1 : n
    r(i) = sum ( rand ( 1, 10 ), 2 );
  end
  t2 = toc;
  fprintf ( 1, '  %10d  %f  %f\n', n, t1, t2 );
  n = n * 10;
end
```

Nonuniform Distributions - Normal Distribution

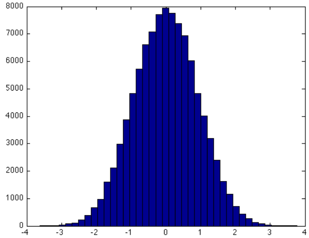
Here we can make 2 conclusions:
Avoid loops if you can do it all at once, and...
RANDN() is only about twice as efficient as our scheme now!

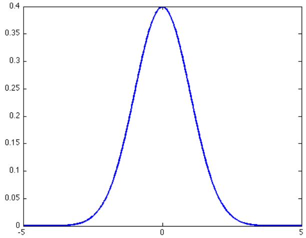
```
      N     RANDN()    Sum RAND


      1   0.000048   0.000025
     10   0.000067   0.000085
    100   0.000395   0.000676
   1000   0.004002   0.006763
  10000   0.038408   0.067445
 100000   0.373558   0.672546
1000000   3.735183   6.716099
```

Nonuniform Distributions

Our approximate normal is not as efficient as RANDN; It also has the flaw that it can never generate a value less than -10 or more than +10, although the true normal distribution can generate *any* finite value.

Let's go ahead and compare the histogram of 100,000 values to the true PDF:





Nonuniform Distributions

What do we do if we want to generate a sequence of samples from some other probability density function?

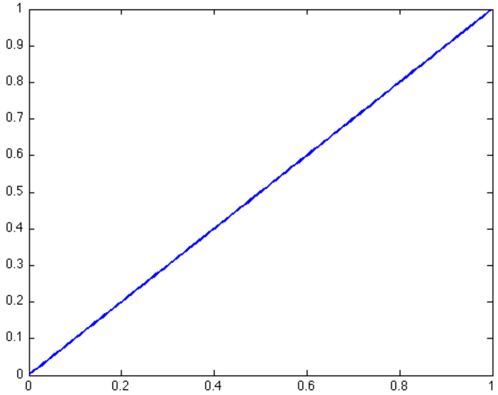
Suppose we can compute (or estimate) the CDF; remember that **cdf(x)** is the probability that a randomly chosen value is less than or equal to **x**.

Now suppose that we can compute the *inverse* function of the CDF, or **ICDF**. Given a probability **p**, then icdf(p) returns a value **x** so that cdf(x) $= p$.

So the **ICDF** function is given a number between 0 and 1 (the probability) and returns an element **x** of the set, and it does this with the right probability density.

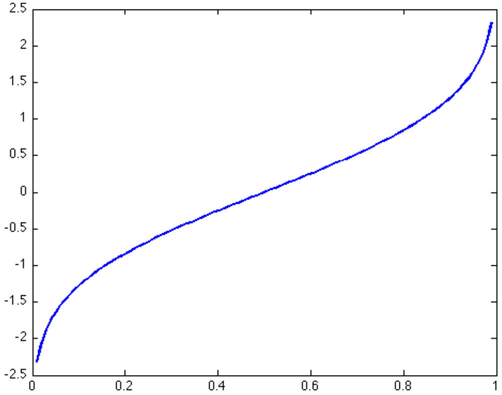
Nonuniform Distributions: Uniform ICDF

The uniform ICDF is simply $x = \mathrm{icdf}(p) = p$;



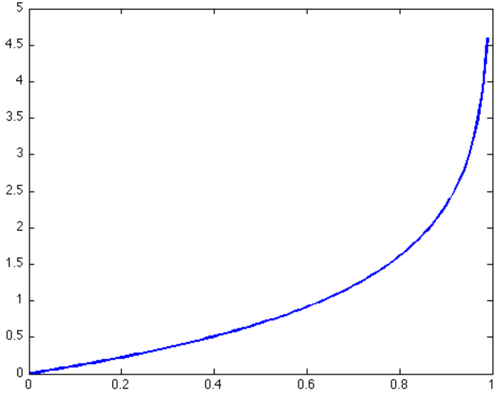
Nonuniform Distributions: Normal ICDF

The normal ICDF is $x = \mathrm{icdf}(p) = \sqrt{2}\,\mathrm{erf}^{-1}(2p-1)$;



Nonuniform Distributions: Exponential ICDF

The exponential ICDF is $x = \mathrm{icdf}(p) = -\log(1 - p)$;
$\mathrm{icdf}(p) \to \infty$ as $p \to 1$.



Nonuniform Distributions: Generating a Sequence

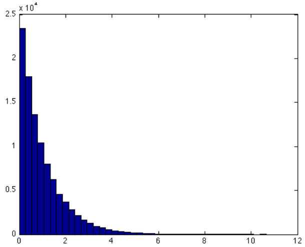
If we have the ICDF, then we can easily generate a sequence of sample values:

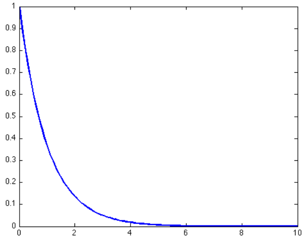
Here, we generate N sample values of the exponential PDF.

```
rand ( 'state', 12345 );  <-- Set the seed;
u = rand ( n, 1 );        <-- Column vector;
e = - log ( 1 - u );      <-- Apply ICDF.
```

Nonuniform Distributions: Generating a Sequence

Compare the histogram of 100,000 values to the PDF:





Nonuniform Distributions: No ICDF?

Suppose we don't have an inverse CDF, only have the CDF.

To generate a sequence, I pick a random number **R** and now I need to know the value of **X** so that

$$\mathrm{cdf}(X) = R.$$

If I had the ICDF, I could just write:

$$X = \mathrm{icdf}(R)$$

Instead, I have to try to solve a nonlinear equation! I know that **X** is in the interval **[A,B]**, and CDF is monotone. So I can try to use the bisection method to solve the equation

$$F(X) = \mathrm{cdf}(X) - R = 0$$

This is painful, but it will work.

Nonuniform Distributions: Solve by Bisection

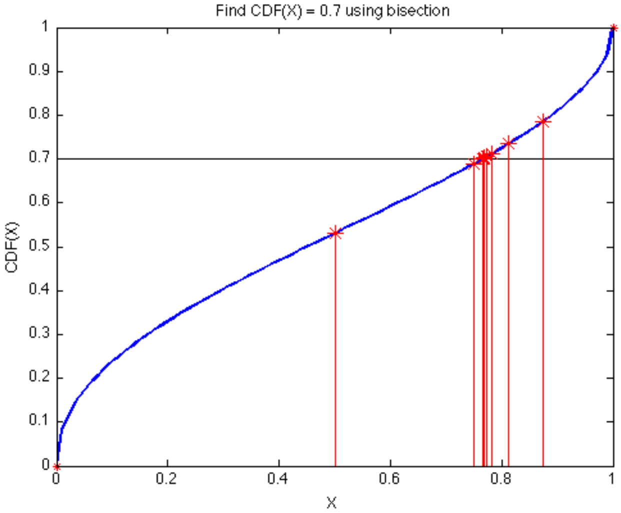
```
a = 0.0; fa = betacdf ( a, 0.45, 0.5 );
b = 1.0; fb = betacdf ( b, 0.45, 0.5 );

while ( 1 )
  c = 0.5 * ( a + b );
  fc = betacdf ( c, 0.45, 0.5 );
  if ( abs ( fc - 0.7 ) < 0.00001 )
    break
  elseif ( fc - 0.7 < 0.0 )
    a = c;
  else
    b = c;
  end
end
```

Nonuniform Distributions: Bisection Output

```
F(0.000000) = 0.000000
F(1.000000) = 1.000000
F(0.500000) = 0.530387
F(0.750000) = 0.688904
F(0.875000) = 0.785831
F(0.812500) = 0.734265
F(0.781250) = 0.711021
F(0.765625) = 0.699840
F(0.769531) = 0.702611
F(0.767578) = 0.701224
F(0.766602) = 0.700532
F(0.765869) = 0.700013
F(0.765747) = 0.699927
F(0.765808) = 0.699970
F(0.765839) = 0.699991
```

Nonuniform Distributions: Bisection Plot

Seek X so that BetaCDF(X,0.45,0.5) = 0.7



niaTech

Monte Carlo Method: Sampling

Integrals: The Riemann Sum

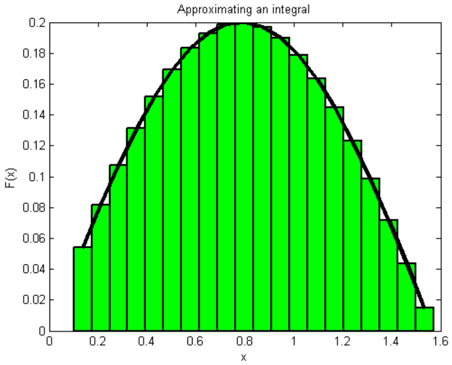
The very definition of the integral invokes a kind of sampling.

A Riemann sum divides the interval [a,b] into subintervals, and approximates the integral by taking an arbitrary point from each subinterval multiplying by the subinterval length, and summing:

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{N} f(x_i) * (b_i - a_i)$$

The Riemann integral is defined as the limit of all possible such Riemann sums, as the subinterval width goes to zero.

Integrals: The Riemann Sum



Integrals: Monte Carlo Approximation

The Monte Carlo approach to this problem can be written as

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{N} f(x_i)\frac{b-a}{N}$$

where the x_i's are chosen uniformly at random from [a,b].

This can be thought of as a “sloppy” Riemann sum
or as an attempt to invoke the mean value theorem for integrals:

$$\int_a^b f(x)\,dx = (b-a) * \overline{f(x)} \approx (b-a) * \frac{1}{N}\sum_{i=1}^{N} f(x_i)$$

Here, we regard the sum of values of $f(x)$ as an estimate of $\overline{f(x)}$, the mean value of f.

Integrals: An Integral over [0,1]

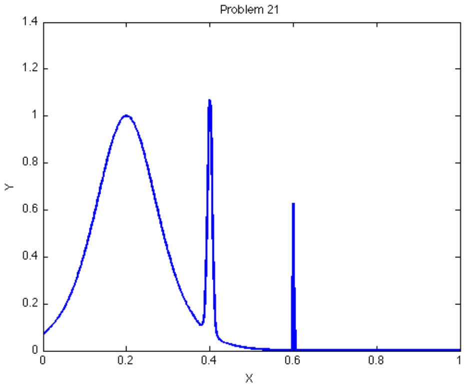
Here's an example integrand for the unit interval:

$$\begin{aligned} f(x) = & \text{sech}^2(10.0 * (x - 0.2)) \\ & + \text{sech}^2(100.0 * (x - 0.4)) \\ & + \text{sech}^2(1000.0 * (x - 0.6)) \end{aligned}$$

It's hard to know from this formula what to expect!

Integrals: An Integral over [0,1]

From the plot, it should be clear that this is actually a difficult function to integrate.



Integrals: An Integral over [0,1]

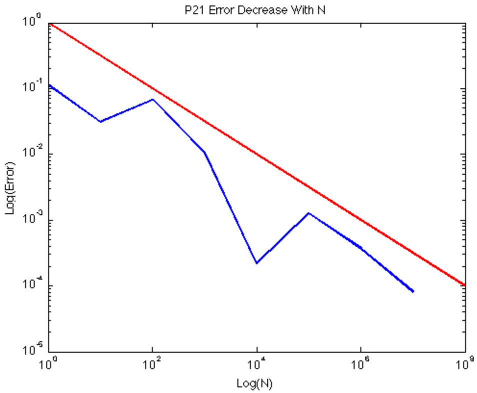
If we apply the Monte Carlo Method, we get "decent" convergence:

```
       1  0.096400  1.1e-01
      10  0.179749  3.1e-02
     100  0.278890  6.8e-02
    1000  0.221436  1.0e-02
   10000  0.210584  2.1e-04
  100000  0.212075  1.2e-03
 1000000  0.211172  3.6e-04
10000000  0.210721  8.1e-05

   Exact 0.21080273631018169851...
```

VirginiaTech

Integrals: An Integral over [0,1]

Moreover, we see the same fairly regular decrease in error with the increasing number of points, which should tend to a slope of $-\frac{1}{2}$ on a log-log plot.



Integrals: An Integral over [2,7]

Let's take our optimization function and integrate it over [2,7].

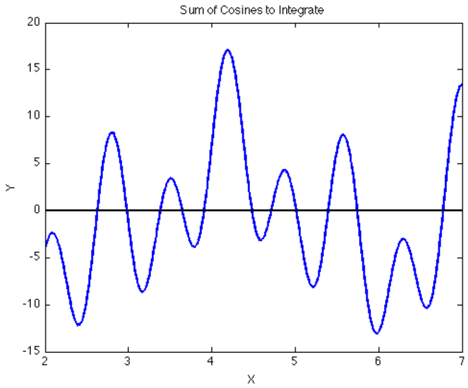
$$
\begin{aligned}
f(x) = \cos(x) \\
+5 * \cos(1.6 * x) \\
-2 * \cos(2.0 * x) \\
+5 * \cos(4.5 * x) \\
+7 * \cos(9.0 * x);
\end{aligned}
$$

We know this is a wiggly function. We just want to practice shifting the interval from [0,1] to [A,B].

```
X = 2.0 + ( 7.0 - 2.0 ) * rand ( n, 1 );
```

Integrals: An Integral over [2,7]

You should recognize this plot.



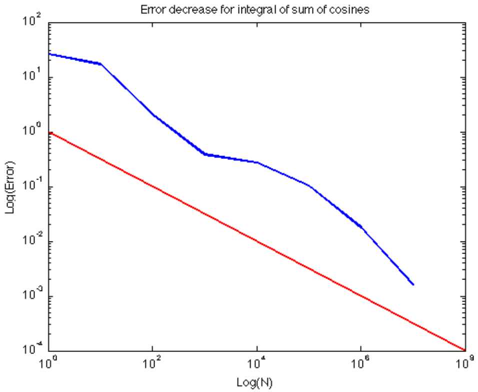
Integrals: An Integral over [2,7]

If we apply the Monte Carlo Method, we get "decent" convergence:

```
       1    21.486936   2.6e+01
      10   -21.701206   1.7e+01
     100   -2.472846    2.0e+00
    1000   -4.911594    3.8e-01
   10000   -4.253230    2.7e-01
  100000   -4.424016    1.0e-01
 1000000   -4.509781    1.7e-02
10000000   -4.529185    1.6e-03

   Exact   -4.527569...
```

Integrals: An Integral over [2,7]

Again, we see the approximation to a slope of $-\frac{1}{2}$ on a log-log plot.



Integrals: Convergence Depends on Variance

The two functions both seem "hard" to integrate, but for one, we got an error 20 times smaller. Why?

The error depends in part on the **variance** of the function.

To get a feeling for the variance of these two functions, note:

*The first function has an integral of about 0.2 over the interval [0,1], then its average value if 0.2. The function ranges between 0 and 1, so the pointwise variance is never more than 0.64.

*The second function has an integral of about -4.5 over an interval of length 5, for an average value of about -1. Its values vary from -14 to +16 or so, so the pointwise variance can be as much as 200.

Integrals: MATLAB commands for 2D

In order to define functions of two variables and plot them, it will be useful to take advantage of some MATLAB commands:

- **[X,Y]=meshgrid(x,y)** takes x and y vectors and makes a table;
- **surf(X,Y,Z)** makes a surface plot of **Z(X,Y)** (data must be matrices);
- **i = find (condition)**: indices that satisfy a condition
- logical expressions are numbers, equal to 0 (false) or true (1); We can use them in arithmetic!

Integrals: An Integral over a Square

Since we know how to sample a square, let's try to use the Monte Carlo method to compute the integral of

$$f(x, y) = |x^2 + y^2 - 0.75|$$

over the square [-1,+1]x[-1,+1].
Because of the absolute value function, **f(x)** is not differentiable. It would be tricky to try to work out the integral, even though this is a pretty simple function.
Now we need to use pairs of random numbers to go from [0,1] to [-1,+1]x[-1,+1]:

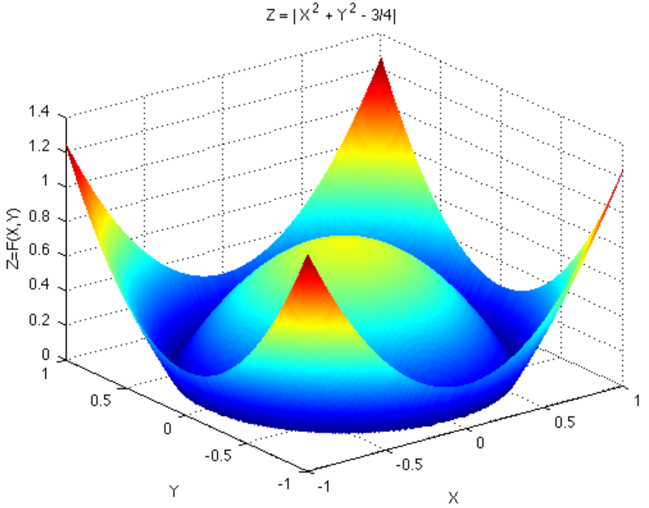
```
X = -1.0 + 2.0 * rand ( n, 1 );
Y = -1.0 + 2.0 * rand ( n, 1 );
```

Integrals: An Integral over a Square

How do we plot a function **f(x,y)**?

```
x = -1.0 : 0.01 : +1.0;
y = -1.0 : 0.01 : +1.0;

[ X, Y ] = meshgrid ( x, y );

Z = abs ( X .* X + Y .* Y - 0.75 );

surf ( X, Y, Z, ...
  'FaceColor', 'Interp', 'EdgeColor', 'Interp' )

xlabel ( 'X' );
ylabel ( 'Y' );
zlabel ( 'Z=F(X,Y)' )
title ( 'Z = | X^2 + Y^2 - 3/4|' )
```

Integrals: An Integral over a Square



Integrals: An Integral over a Square

How do we program the Monte Carlo Method for a square?

```
exact = 1.433812586520645;
area = 4.0;
n = 1;
for logn = 0 : 7

  x = -1.0 + 2.0 * rand ( n, 1 );
  y = -1.0 + 2.0 * rand ( n, 1 );

  result = area * sum ( abs ( x.^2 + y.^2 - 0.75 )) / n;

  fprintf ( 1, '  %8d  %f  %e\n', ...
    n, result, abs ( result - exact ) );
  n = n * 10;
end
```

Integrals: An Integral over a Square

Our convergence:

```
       1   0.443790   9.900230e-01
      10   1.547768   1.139549e-01
     100   1.419920   1.389309e-02
    1000   1.464754   3.094170e-02
   10000   1.430735   3.077735e-03
  100000   1.428808   5.004836e-03
 1000000   1.432345   1.467762e-03
10000000   1.433867   5.394032e-05

 Exact     1.433812586520645
```

Integrals: An Integral over a Circle

Let's do an integral in the circle; instead of working with the efficient method that uses all the random numbers, we'll do the rejection method, to show how that works.
Our integrand will be

$$f(x,y) = x^2$$

and we will integrate over the circle of radius 3 centered at (0,0).

For this problem, the function is smooth, but the region is not a rectangle. It's not hard to work out the exact answer using polar coordinates, because the circle is special. But the Monte Carlo approach with rejection would handle an ellipse, or a region shaped like Mickey Mouse, just as easily.

Integrals: An Integral over a Circle

Now we need to use pairs of random numbers to go from [0,1] to [-3,+3]x[-3,+3] and then reject those that lie outside the circle.

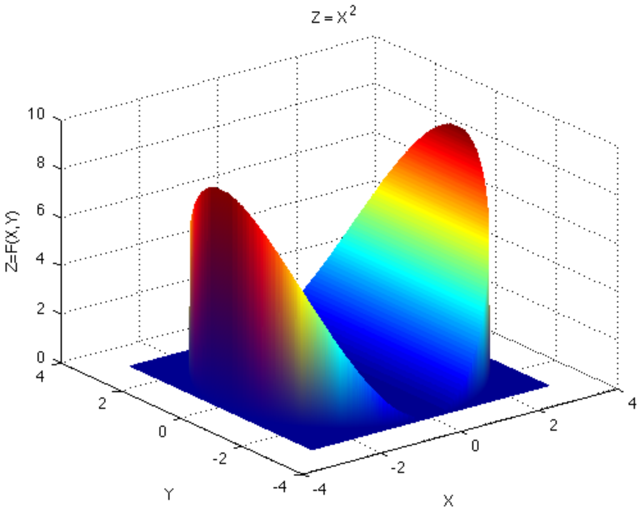
```
X = -3.0 + 6.0 * rand ( n, 1 );
Y = -3.0 + 6.0 * rand ( n, 1 );
i = find ( X .* X + Y .* Y <= 9 );
```

Integrals: An Integral over a Square

How do we plot a function **f(x,y)** on a circle? SURF wants a rectangle of data. We zero out the function outside the circle.

```
r = 3;
x = -r : 0.01 : +r;
y = -r : 0.01 : +r;

[ X, Y ] = meshgrid ( x, y );

Z = ( X .* X + Y .* Y <= r * r ) .* ( X .* X );

surf ( X, Y, Z, ...
  'FaceColor', 'Interp', 'EdgeColor', 'Interp' )
xlabel ( 'X' );
ylabel ( 'Y' );
zlabel ( 'Z=F(X,Y)' )
title ( 'Z = X^2' )
```

Integrals: An Integral over a Circle



Integrals: An Integral over a Circle

```
exact = 63.617251235193313079;
area = pi * r * r;
n = 1;
for logn = 0 : 7
  x = - r + 2 * r * rand ( n, 1 );
  y = - r + 2 * r * rand ( n, 1 );
  i = find ( x .* x + y .* y <= r * r );
  n2 = length ( i );
  result = area * sum (  x(i) .* x(i) ) / n2;
  fprintf ( 1, '  %8d  %f  %e\n', ...
    n2, result, abs ( result - exact ) );
  n = n * 10;
end
```

Integrals: An Integral over a Circle

If we apply the Monte Carlo Method, we get "decent" convergence:

```
      1   74.493641   1.087639e+01
      9   52.240562   1.137669e+01
     78   62.942561   6.746898e-01
    784   61.244910   2.372341e+00
   7816   62.653662   9.635895e-01
  78714   63.405031   2.122204e-01
 785403   63.669047   5.179565e-02
7853972   63.624209   6.958217e-03

Exact     63.617251235193313079
```

Conclusions

We've concentrated on the Monte Carlo method as a means of sampling. This gave us an alternate means of solving integration problems. Of course, there are already other methods for dealing with integration problems.

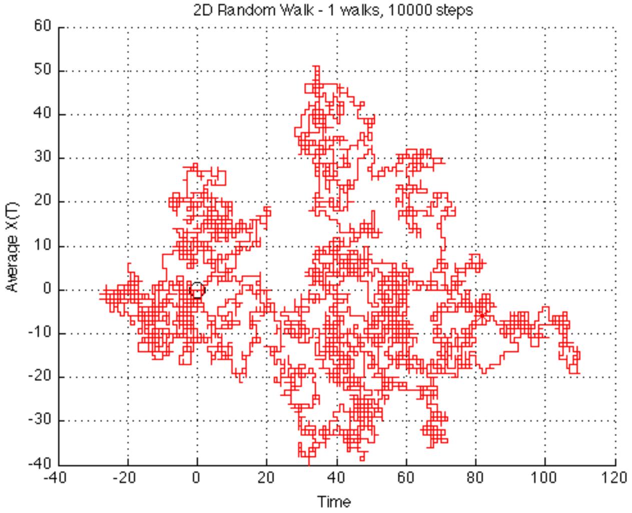
We will next consider a Monte Carlo application in which no other method is available, the field of **simulation**.

We will consider the common features of tossing a coin many times, watching a drop of ink spread into water, and observing a drunk stagger back and forth along Main Street!

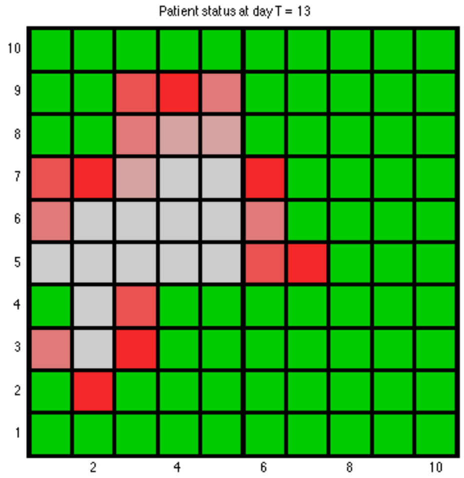
Using ideas from the Monte Carlo method, we will be able to pose and answer questions about these random processes.

Conclusions: Traffic Simulation

Conclusions: Random Walks



Integrals: Disease Spread


Patient status at day T = 13