Random numbers and their generation

Philosophy
- Uses of random numbers
- What does it mean to be random?

How to
- Using system random generators
- Simple generators
  - Their benefits
  - Their problems
- Even simpler generators

Getting more use out of your RNG
- Changing the range
- Changing the probability distribution
- Shuffling a deck of cards

More general distributions
- Brief review of probability theory
- Possion processes.
- Gaussian distributions
- Box-Muller method

Descriptive statistics
- Gaussian/Poisson in the world of real data
- Central Limit Theorem
- Simulating random arrivals

Why Random Numbers?

Simulation

Testing

Monte Carlo evaluation of integrals/volumes

What does it mean to be random

Truly random (clicks of Geiger counter).

Pseudo random -- program to generate points is statistically uncorrelated with program that uses its output.

Real world random -- method has to work for your example.

System-supplied generators

```
#include <stdlib.h>
#define RAND_MAX ......

void srand(unsigned seed);
int  rand(void);
```

Truly random (clicks of Geiger counter).

NAME

rand, srand - simple random number generator

SYNOPSIS

```
srand(seed)
int seed;

rand()
```

DESCRIPTION

rand() uses a multiplicative congruential random number generator with period 2**32 to return successive pseudo-random numbers in the range from 0 to (2**31)-1.

srand() can be called at any time to reset the random-number generator to a random starting point. The generator is initially seeded with a value of 1.

SYSTEM V DESCRIPTION

rand() returns successive pseudo-random numbers in the range from 0 to (2**15)-1.

SEE ALSO

drand48(3), random(3)

NOTES

The spectral properties of rand() leave a great deal to be desired. drand48(3) and random(3) provide much better, though more elaborate, random-number generators.

BUGS

The low bits of the numbers generated are not very random; use the middle bits. In particular the lowest bit alternates between 0 and 1.

Simple Stuff

A. To get a rand in [0,1], write the code

```
x = rand()/(RAND_MAX + 1.0);
```

B. When using a system supplied RNG, always use higher order bits. That is, to get a random integer between 1 and 10 try

```
j = 1 + (int)(10.0 *rand()/(RAND_MAX + 1.0));
```

instead of

```
j = 1 + (rand()%10);
```

C. Be aware of (and worry about) the value of RAND_MAX. 32767 is just not enough for many applications (this is the value on phoenix and ernie). If you are using a few random numbers, it's ok. But, for many (eg 10^6 as you might use for a Monte Carlo simulation), this means that you repeat each number 30 times.

How do they do it?

System supplied RNG's are typically linear congruential generators.

Sequence is $I_1, I_2, \ldots I_n$ each between 0 and RAND_MAX which are defined by the recurrence

$$I_{j+1} = a\, I_j \; + \; c \pmod{m}$$

m -- modulus
a -- multiplier
c -- increment

Choose m,a,c to make the period as long as possible (ideally m-1). Then you can start anywhere (ie any seed works).

Simplae (BAD) example

$I_0 = a = c = 7;\ m = 10;$

7,6,9,0,7,6,9,0,...

ANSI Committee recommended code

```
unsigned long next=1;

int rand(void)      {
   next = next*1103515245 + 12345;
   return(unsigned int) (next/65536)%32768;
   }

void srand(unsigned int seed)  {
      next=seed;
      }
```

This isn't great, but not a disaster.

Small tweaks can make a generator like this either much better or much worse (e.g. swap high and low order bits ruin this one).

Concerns -- correlation among successive #s

Imagine that you are choosing points in k-dimensional space. Points tend not to fill up the space but to lie on k-1 dimensional flats. There can be no more than $m^{1/k}$ such flats.

k=3, RAND_MAX = 32768 ==> 30 such planes

So, what's the **quick and dirty** best?

$$I_{j+1} = a\, I_j \pmod{m}$$

$a = 7^5$ and $m = 2^{31} - 1 = 2147483647$

Difficulty, doing the multiplication in 64 bit integer.

Trick (see the text) for doing it all in 32 bits.

Further extensions -- mix the bits between calls and get better numbers.

Final caveats -- *KNOW YOUR APPLICATION*

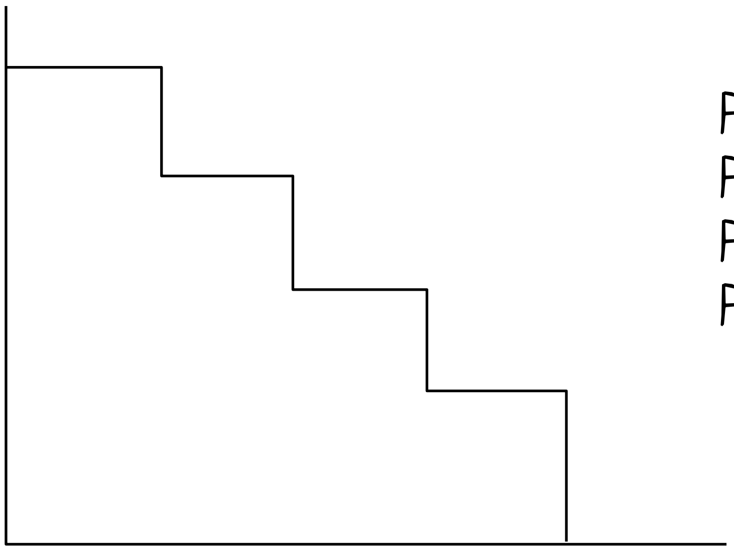
Integer between 1 and N, see above.

How to pick against other distributions

How to select points geometrically

How to shuffle a deck of cards?

How to pick in biased fashion



$p(1) = .4$
$p(2) = .3$
$p(3) = .2$
$p(4) = .1$

Choosing points geometrically.
Want random points in 2D.

How to?

1. Choose x and y randomly
2. Choose a random radius and a random angle θ to place the point in polar coordinates.

1. tends to be a box
2. tends to be a circle but puts most points too close to the origin

Shuffling a deck of cards (from Knuth)

To Shuffle $X_1, X_2, \ldots X_m$

P1: Set $j = m$;

P2: Generate a random number U (uniform between 0 and 1).

P3: Set $k = fl(jU) + 1$ (k is now a random number between a and j).
Exchange X_k and X_j.

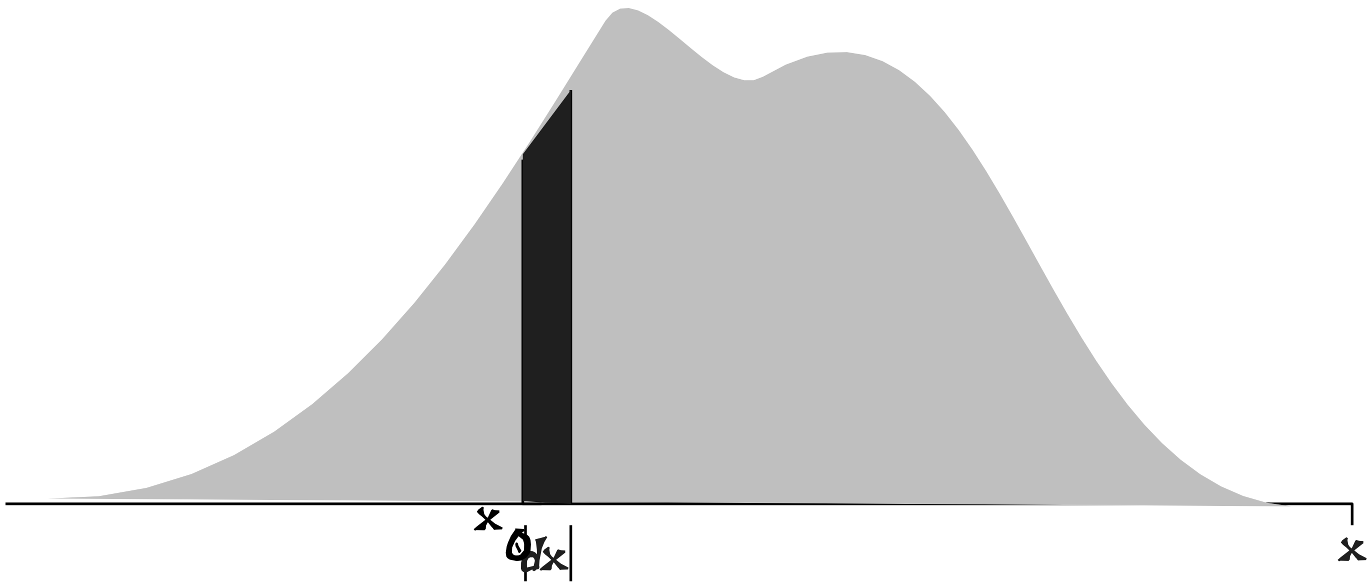
P4: Decrease j by 1. If $j > 1$, return to P1.

On to Probability theory

Transforming from the discrete to the continuous.

Work with probability distributions

$p(x)$



prob(x between x_0 and $x_0 + dx$) = $p(x_0)dx$

$$\int_{-inf}^{inf} p(x)dx = 1$$

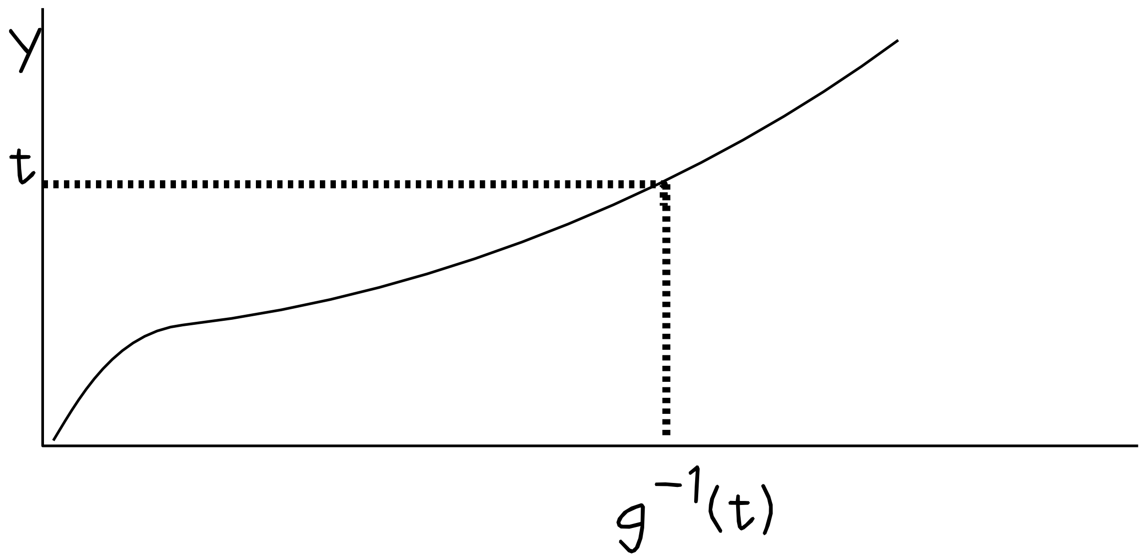
Cumulative Distribution Function

$$F(y) = prob(x<=y) = \int_{-inf}^{y} p(x)\, dx$$

$p(x) = dF(x)/dx$

Generate x and then compute $y = g(x)$
where $g(.)$ is monotonic, differentiable,
so $g^{-1}(.)$ is well defined, unique

$$prob(y <= t) = prob(x <= g^{-1}(t))$$



If x uniform on [0,1]

$\text{prob}(y <= t) = g^{-1}(t)$

So, $p(t) = d(g^{-1}(t))/dt$

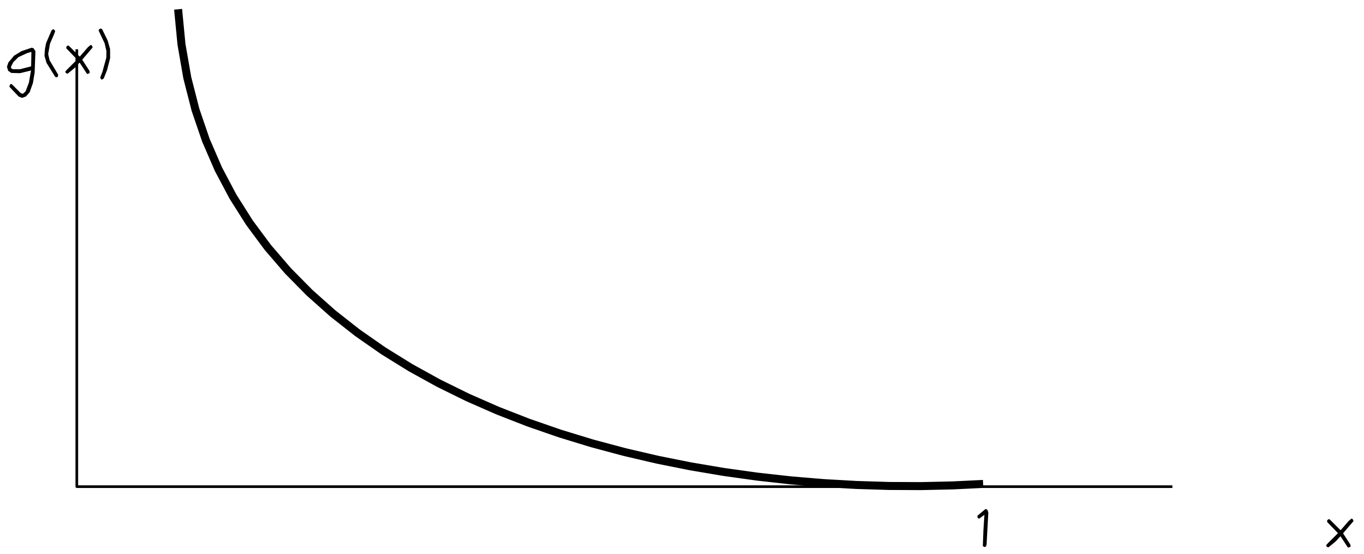
(actually $|d(g^{-1}(t))/dt|$ to cover the monotonic decreasing case)

EXAMPLE

$$g(x) = -1/\lambda \ln x$$

$$x = e^{-\lambda g(x)} = e^{-\lambda t}$$

$$p(t) = |-\lambda e^{-\lambda t}| = \lambda e^{-\lambda t}$$



Physical Interpretation

Suppose an event occurs in a small interval with probability $\lambda\Delta t$,(e.g. λ is the expected number of events per second).

prob(exactly k events in time t) =

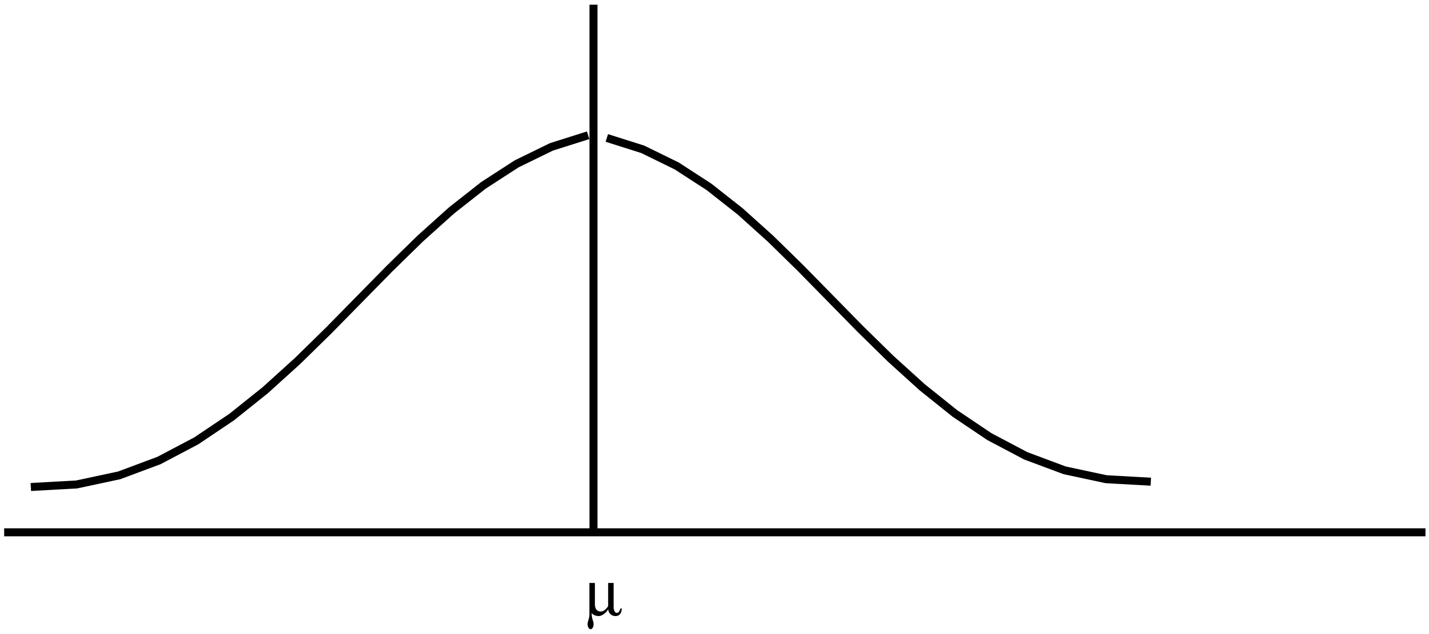
$$\frac{(\lambda t)^k}{k!} e^{-\lambda t} \quad \text{(Poisson distribution)}$$

prob(0 events) = $e^{-\lambda t}$

prob. density of time t to 1st event = $\lambda e^{-\lambda t}$

expected value = $1/\lambda$ (prove this by Calculus)

EXAMPLE 2 (Gaussian = Normal = Bell-shaped)



$$p(x) = \left(\frac{1}{\sigma (2\pi)^{1/2}} \right) e^{-(x-\mu)^2/2\sigma^2}$$

arises everywhewre (central limit theorem)
sum of small independent errors

Used to model experimental errors

How to generate?

Simple method -- pick N independent samples from a uniform distribution and average.

Quick and dirty but of limited range

Box-Muller method (transform in 2D)

Start with x_1, x_2

independent uniform random on [0,1]

Define $g(x_1, x_2) = (y_1, y_2)$ where

$$y_1 = (-2 \ln x_1)^{1/2} \cos (2\pi x_2)$$
$$y_2 = (-2 \ln x_1)^{1/2} \sin (2\pi x_2)$$

So $g^{-1}(y_1, y_2) = (x_1, x_2)$ where

$$x_1 = e^{-1/2(y_1^2 + y_2^2)}$$
$$x_2 = 1/(2\pi) \arctan(y_2/y_1)$$

Compute the Jacobian determinant
(ij entry is partial of x_i wrt y_j) to get p(y).

Useful fact $d/dy \arctan(y) = 1/(1 + y^2)$

$$p(y_1,y_2) = \left(1/(2\pi)^{1/2} e^{-y_1^2/2}\right) *$$
$$\left(1/(2\pi)^{1/2} e^{-y_2^2/2}\right)$$

Box - Muller (cont.)

Why does this work?

We have to go to 2D because we can't evaluate the integral in 1D.

We can evaluate in 2D by going to polar coordinates.