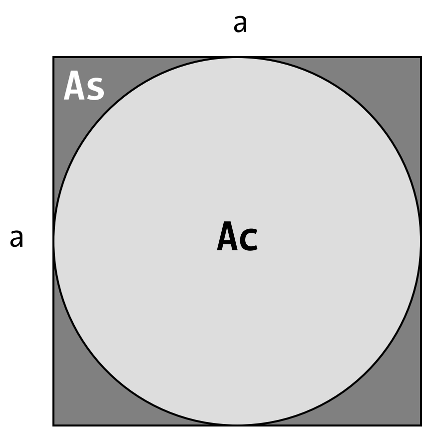
4.1.2.2.1

Random Numbers and Distributions
Session 1

- randomness and pseudorandomness
- linear congruency generators
- how randomness is tested
- random number generators in Perl

let's calculate pi with random numbers



$$A_S = a^2$$

$$A_C = \frac{\pi a^2}{4}$$

$$\frac{4A_C}{A_S} = \pi$$

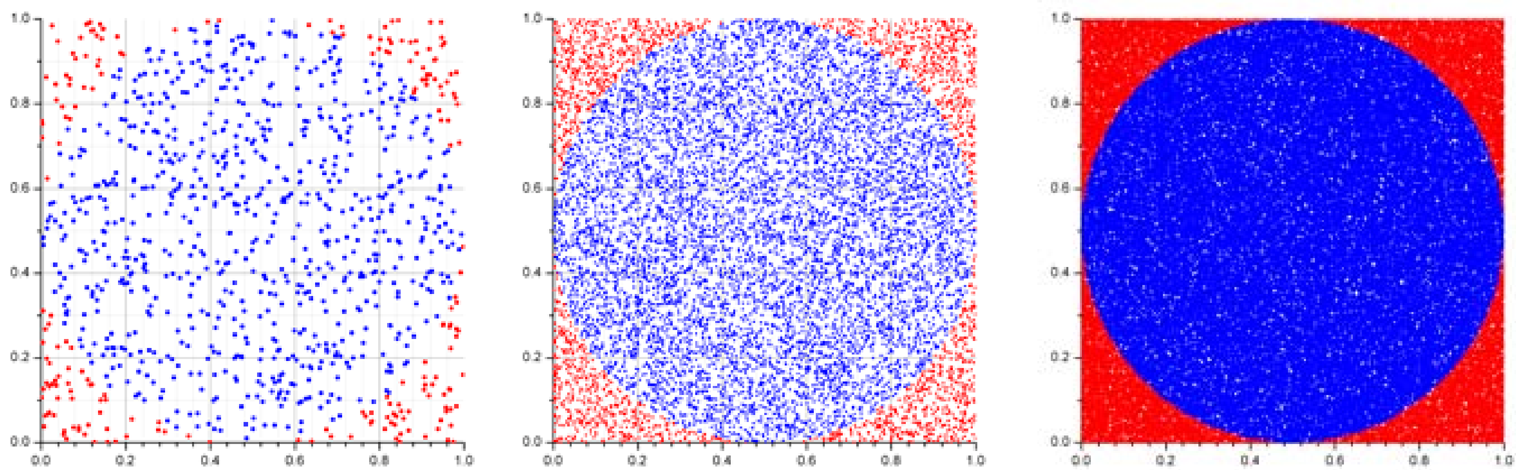
- ratio of the area to the inscribed circle to the area of the inscribed square is a multiple of pi
- select N points at random within the square
  - if you have no computer, drop rice into a square container
  - π = 4 (points within circle) / N

drop rice

- pick two uniformly distributed random numbers using rand()
  - in range 0-1
- calculate distance from center of square bounded by (0,0)-(1,1)
  - report whether point is inside circle

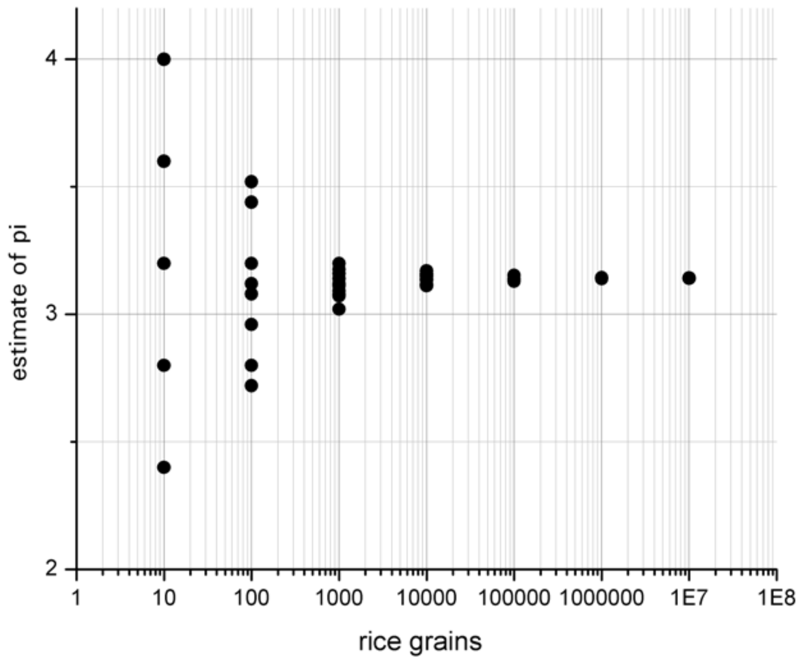
```
my $N = $ARGV[0];
for (0..$N-1) {
    my ($x,$y) = (rand(),rand());
    my $d = (0.5-$x)*(0.5-$x) + (0.5-$y)*(0.5-$y);
    my $dt = int ($d < 0.25);
    print $x,$y,$dt;
}

0.827984035480767 0.347324477508664 1
0.386150703765452 0.519155289512128 1
0.805041420273483 0.437904356978834 1
0.938070443924516 0.767489458434284 0
0.202308556064963 0.78770472900942 1
0.0836395183578134 0.00785069400444627 0
```

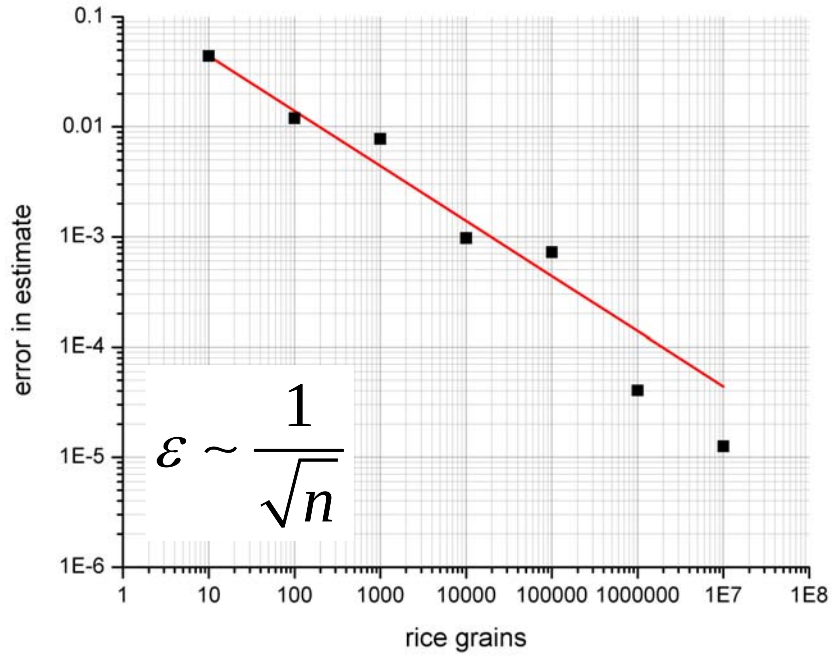
pi to 0.4% with 200,000 random numbers in 1 second



rice, As	1,000	10,000	100,000
inside circle, Ac	794	7,842	78,568
pi (4*Ac/As)	**3.176**	**3.1368**	**3.14272**
error	1%	0.2%	0.04%

pi estimate as function of rice grains – 10 iterations






```
average 10 3.28
average 100 3.104
average 1000 3.1172
average 10000 3.14464
average 100000 3.13932
average 1000000 3.14172
average 10000000 3.1415532
```

error is inversely proportional to the square root of the number of samples

to lower the error by factor 10, you need 100 as many points

to lower the error by a factor 100, you need 10,000 as many points

what is randomness?

- a sequence of numbers is random if there is no correlation between values in the sequence
- computers generally cannot produce random numbers, only pseudo-random numbers
  - pseudo-random numbers are generated by algorithms which, depending on sophistication, produce numbers that are effectively random, if limitations of the algorithm are understood
  - randomness requirements vary with application
    - cryptography – extremely rigorous
- true random numbers are created by harnessing an unpredictable physical process, like radioactive decay
  - kits that generate/monitor white noise (audio, thermal, electronic) are available
  - http://www.fourmilab.ch/hotbits/

why are random numbers useful

- stochastic (probabilistic) simulations
  - your system is described by a probabilistic model
    - coverage process
    - Markov model
  - your system is deterministic but you would like to model noise
    - molecular dynamics
  - your algorithm to solve the problem is stochastic
    - genetic algorithm
    - simulated annealing
- requirement for non-deterministic values
  - random file names
  - random passwords

definitions

- *stochastic* – pertaining to chance, synonymous with random
- *deterministic* – not stochastic
- *uniformly distributed* – random values with constant probability over their range
- *uniform random deviate (urd)* – a random number from a uniform distribution, usually in the range 0-1
- *gaussian random deviate (grd)* – a random number from a normal distribution, usually mean=0 stdev=1
- *white noise* – no correlation between values, all frequencies present (hiss of radio)
- *coloured noise* – correlation between successive random numbers, certain frequencies more distinct than others
  - pink noise – hiss mixed with rumble
  - brown noise – rumbling

pseudorandom number generators (PRNGs)

- simple PRNGs work using linear congruential generators (LCG, Lehmer 1949)
  - first number is a user-defined seed
  - next number is a function of previous number using the following recursion

$$r_{i+1} = (ar_i + c) \bmod m$$

  - a,c,m are diligently chosen
    - only a few well-known combinations are to be used!
    - 0 <= a < m, 0 <= c < m
  - LCG(m,a,c,seed)

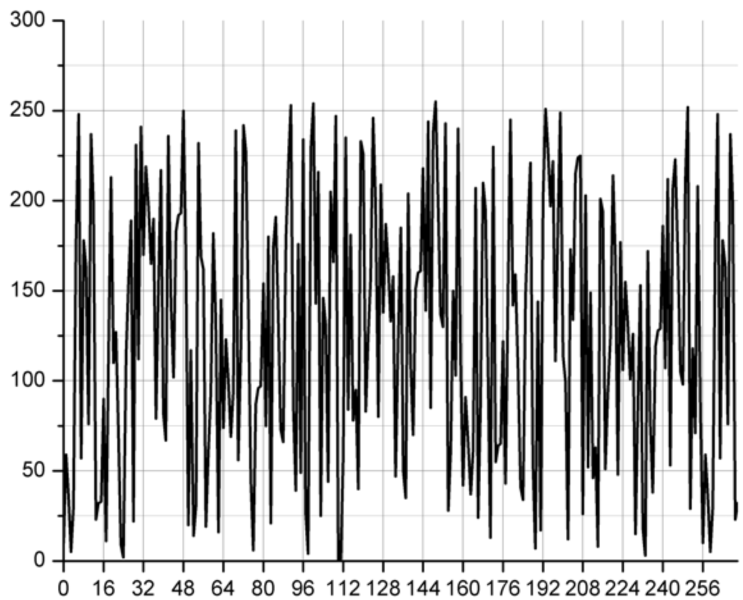
PRNG lists and references
www.taygeta.com/rwalks/node1.html
linux.duke.edu/~mstenner/free-docs/gsl-ref-1.1/gsl-ref_17.html
random.mat.sbg.ac.at/results/karl/server/server.html
triumvir.org/rng
csep1.phy.ornl.gov/rn/node9.html

D.H. Lehmer. Mathematical methods in large-scale computing units. In *Proc. 2nd Sympos. on Large-Scale Digital Calculating Machinery, Cambridge, MA, 1949*, pages 141-146, Cambridge, MA, 1951. Harvard University Press.

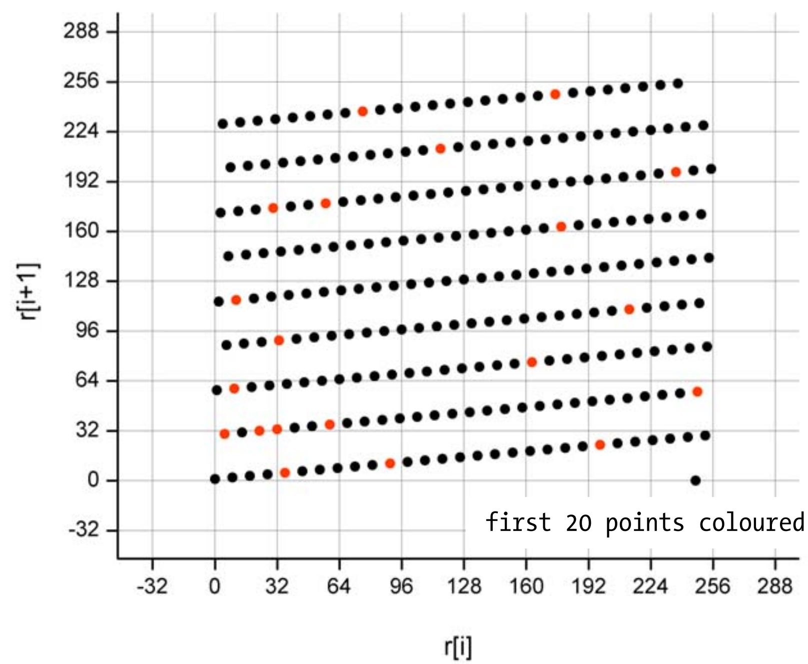
example of LCG

- initial conditions a=57 c=1 m=256 r=10
  - period is 256 (maximum possible)

$$r_{i+1} = (ar_i + c) \bmod m$$

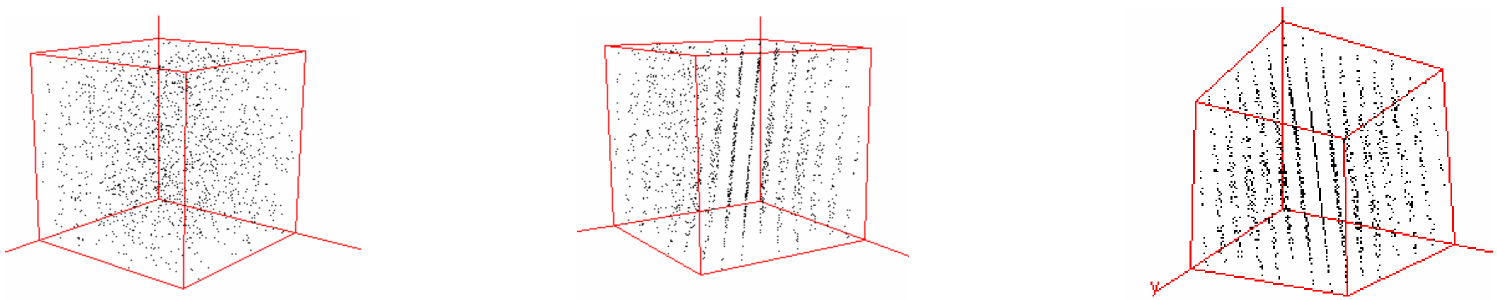


r_{i+1} vs r_i

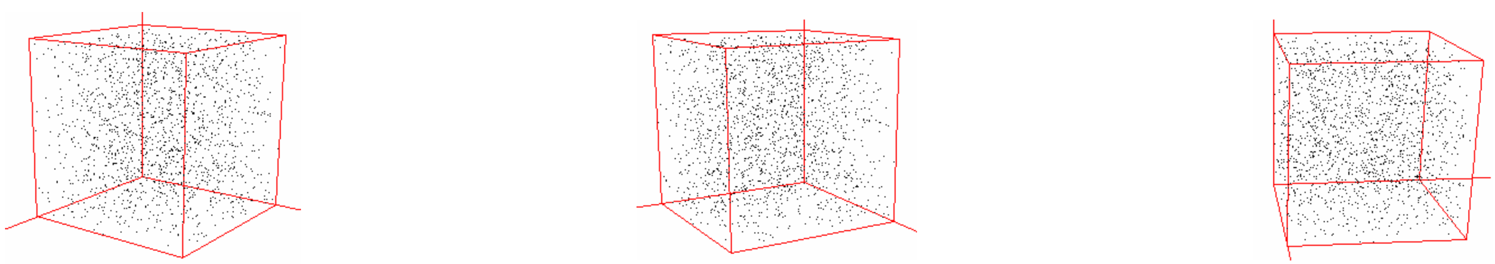


values from LCGs fall on hyperplanes

- LCG k-vectors fall on k-1 dimensional planes
  - (x,y,z) triplets will fall onto 2D planes
  - lattice structure is used to rate LCG constant (minimize distance between planes)
  - there are at most $m^{1/k}$ such planes, but often far fewer if constants are poorly chosen



RANDU, IBM mainframe, a=65539 m=2^31 draw your own lattices at www.cs.pitt.edu/~kirk/cs1501/animations/Random.html



a=16807 m=2,147,483,647=2^31-1
Park, S.K. and K.W. Miller, 1988; Random Number Generators: Good Ones are Hard to Find, Comm. of the ACM, V. 31. No. 10, pp 1192-1201

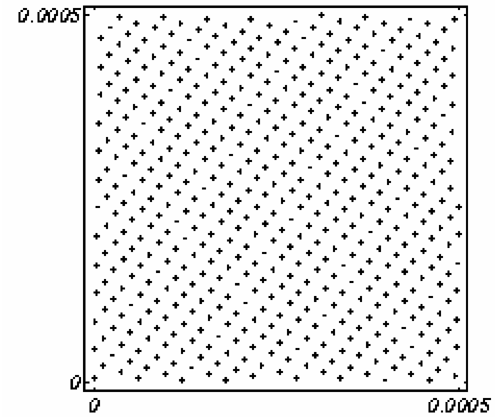
properties of PRNGs

- PRNGs are deterministic – for a given seed you always get the same sequence
- LCGs have a period – numbers eventually repeat
  - never use a PRNG for a significant portion of its period, switch seeds instead and sample another sequence [*read more about your LCG if you are sampling many numbers*]
- LCGs may require warmup time
  - don’t sample a sequence immediately
- LCGs may produce numbers with obvious correlations
  - successive values in Park-Miller minimal standard (a=16807 m=2,147,483,647) can differ only by multiple of a (16,807). Therefore small values tend to be followed by smaller than average values
  - Park-Miller fails chi-squared test after on the order of 10,000,000 values have been sampled (less then 1/100th of the period of the LCG)
  - subsequences of LCG output may have long-range correlations – beware

http://www.cs.berkeley.edu/~daw/rnd/index.html

some common LCGs

- rand() in ANSI C
  - LCG(2^{31},1103515245,1234,1234)
  - low bits are not very random (right-most digits)
- drand48() in ANSI C
  - LCG(2^{48},25214903917,11,0)
- Perl uses one of the following, depending on what is available on your system
  - drand48()
  - random() [non-linear feedback shift register]
  - rand()

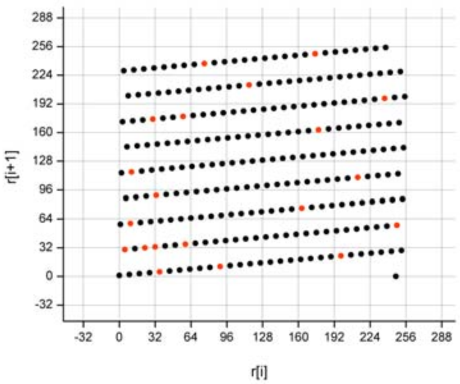


http://www.foo.be/docs/tpj/issues/vol2_2/tpj0202-0008.html
http://www.foo.be/docs/tpj/issues/vol1_4/tpj0104-0002.html
http://homepage.mac.com/afj/lfsr.html
http://en.wikipedia.org/wiki/Linear_feedback_shift_register

how is randomness tested

- entropy
  - information content
  - “resistance to compression”
  - entropy should be as high as possible
- chi-squared test
  - N random numbers selected from range s=1..k. n_s is the number of times s appears, p_s is the probability that a number is s (1/k) [should sample Np_s>5 points]
  - x is chi-square distributed with k-1 degrees of freedom
- Monte Carlo value of pi
- lag plots and k-dimensional plots
  - spectral test computes distance between hyperplanes

$$x = \sum_{s=1}^{k} \frac{n_s^2}{Np_s} - N$$

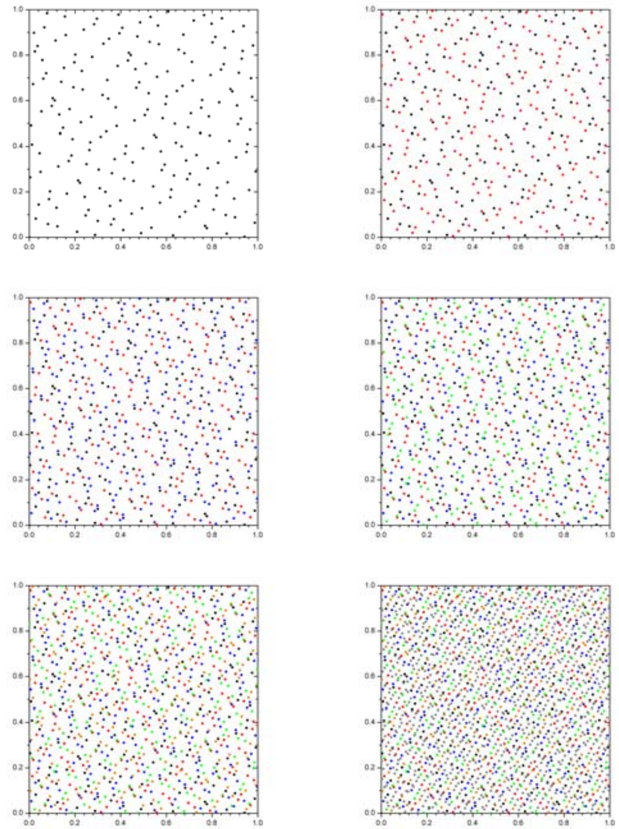


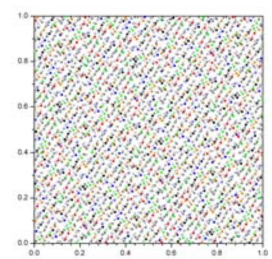
Marsaglia DIEHARD Battery Of Tests on Randomness
http://stat.fsu.edu/pub/diehard/

http://world.std.com/~franl/crypto/random-numbers.html

sub-random sequences

- plotting pairs of numbers LCG does not fill space evenly
  - depending on the LCG, there may be large holes
- our calculation of pi had an error of 1/sqrt(N)
- to sample a space more evenly, a grid is better
  - but you need to know how finely to make the grid when you start
  - you cannot sample a grid until you reach convergence – you are committed to sample the entire grid
- a sub-random sequence (quasi-random) fills space more evenly than LCG values
  - points maximally avoid each other

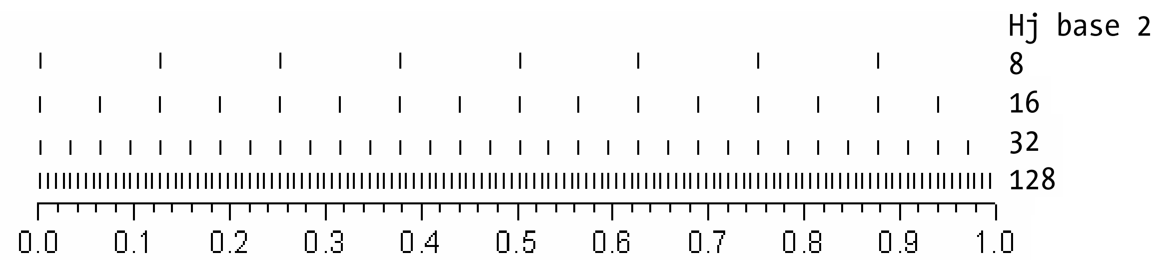




Halton's sequence
100, 200, 400, 800, 1000, 2000, 4000 points

Halton's sequence

- elements in the sequence are defined as follows
  - pick a prime, b
  - element Hj, is computed as follows
    - express j in base b (e.g. if j=50 and b=3, 50 base 3 = 1212)
    - reverse the digits and put a decimal in front (1212 becomes 0.2121)
    - convert back to base 10 (0.2121 base 3 = 0.8642)
- to fill n-dimensional space, use a separate sequence for each dimension
  - generally first n primes are used (my example uses 3 and 5)



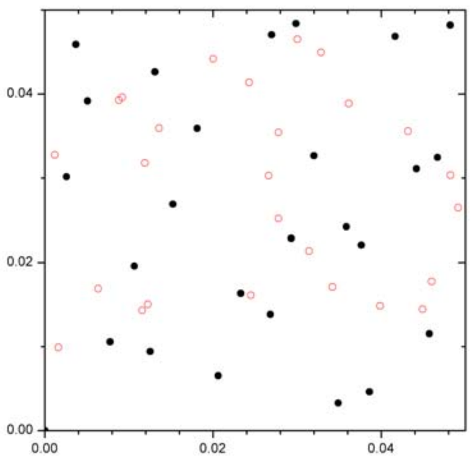
calculating Halton's sequence

- the Math::BaseCalc module makes convenient converting to/from bases

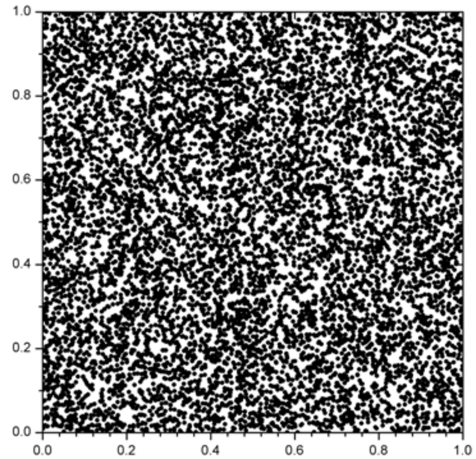
```
use Math::BaseCalc;

$\="\n" ; $,=" ";

my $bobj;
for my $b (3,5) {
    $bobj->{$b} = Math::BaseCalc->new(digits=>[0..$b-1]);
}
for my $i (0..10000) {
    my $halton;
    for my $b (@bases) {
            # convert i to base b
            my $n = $bobj->{$b}->to_base($i);
            # reverse digits
            my $nr = join("", reverse split("",$n));
            # add radix to front
            my $nrr = "0.$nr";
            # convert back to decimal and store in hash
            $halton->{$b} = $bobj->{$b}->from_base($nrr);
    }
    print map {$halton->{$_}} @bases;
}
```

Halton’s sequence fills space evenly

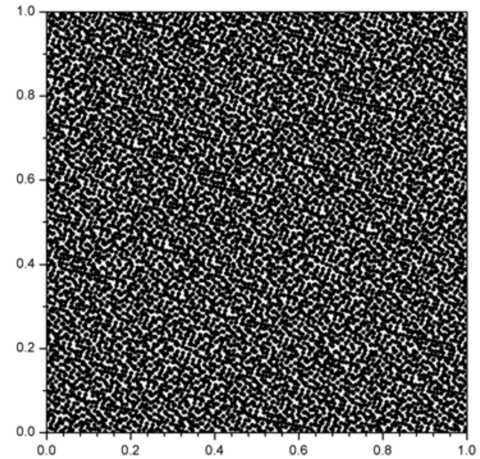
- I generated 10,000 random points over [0,1[^{2} using rand()
- I generated 10,000 random points over [0,1[^{2} Halton’s method (base 3 and 5)



rand() - red
Halton - black



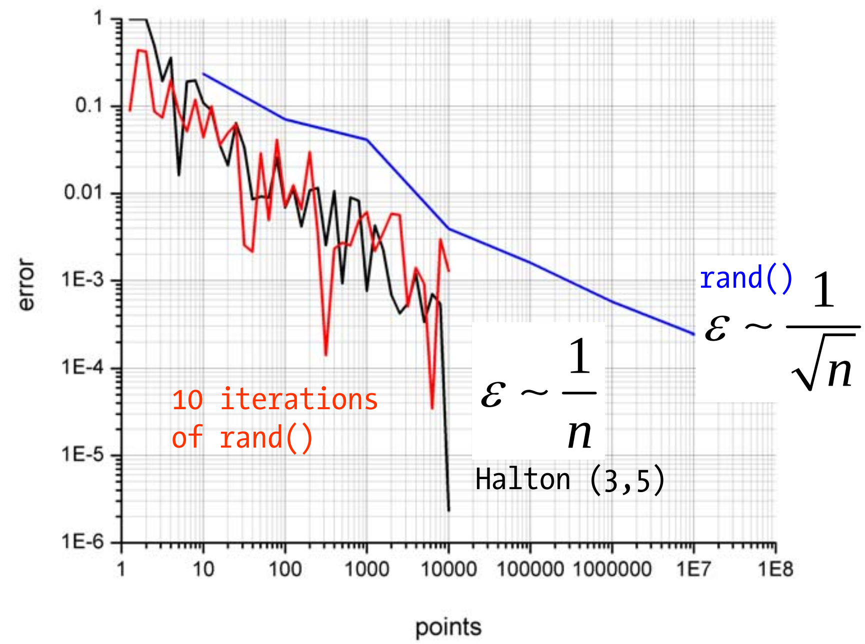
10,000 rand() points



10,000 Halton’s sequence

do we get better estimate of pi?

- Halton's sequence fills space more evenly, and therefore the pi estimate approaches the value of pi faster
- I calculated pi with the same approach as described using three point generators
  - (xi,yi) = (rand(), rand())
  - (xi,yi) = (rand(), rand()), 10 iterations
  - (xi,yi) = ($H_{i,3}$,$H_{i,5}$)
- estimate error using Halton's sequence drops like 1/n rather than 1/sqrt(n)



ways to use PRNs in your code

- shuffling of a list of items
  - two idioms here – shuffle values or indexes
    - assign a[i] a random element from array
    - assign a[random index] = a[i]

```
# shuffle values
@a = sort { rand() <=> rand() } @a;
# shuffle indexes
@a[ sort { rand() <=> rand() } (0..@a-1)] = @a;
```

- throttled shuffling
  - you can shake the array – a little or a lot – and displace elements up to a certain distance from their position
  - increase k to get more shake

```
@a[ sort { $a+k*rand() <=> $b+k*rand() } (0..@a-1)] = @a;
```

random strings

- what's their use?
  - temporary file names
  - passwords
  - random genomic sequence

```
my @c = qw(a t g c);

# create one index value at a time, fetch array value
print @c[rand(@c)] for (1..1024);
# create all index values at once, fetch array via slice
print @c[ map { rand(@c) } (1..10) ];
```

```
my @v = qw(a e i o u);
# three sets of vowels, one set of consonants
my @c = (@v,@v,"a".."z");
# konutl ucoxou ruigwo
print @c[ map { rand(@c) } (1..10) ];
```

random sequence with specific GC content

```
my $gc = 0.4;

if(rand () < $gc) {
  if(rand() < 0.5) {
    print “g”;
  } else {
    print “c”;
  }
} elsif {
  if(rand() < 0.5) {
    print “a”;
  } else {
    print “t”;
  }
}
```

- very unPerly
- we can do better

random sequence with specific GC content

```
my @g = qw(g c);
my @a = qw(a t);
my $gc = 0.4;

# trinary a?b:c operator can be useful
print rand() < $gc ? $g[rand(@g)] : $a[rand(@g)];
```

```
my @g = qw(a t g c);
my $gc = 0.4;

print $g[ 2*(rand() < 0.4) + rand(@g/2) ];
```

- do we really need to generate two random numbers?

random sequence with specific GC content

- generate one random number, r
  - pick g if r < gc/2, otherwise
  - pick c if r < gc, otherwise
  - pick a if r < (1+gc)/2, otherwise
  - pick t
- we are using a uniformly distributed random number to generate a number samples from a different distribution

```
my @g = qw(g c a t);
my $gc = 0.4;

my $r = rand();
my @c = ($gc/2,$gc,(1+$gc)/2,1);
for $i (0..@c-1) {
  next unless $r < $c[$i];
  print $g[$i];
  last;
}
```

what if our genome isn’t finished?

- let base pair “n” appear 1% of the time

```
my @g = qw(g c a t n);
my $gc = 0.4;

my $r = rand();
my @c = ($gc/2,$gc,(1+$gc)/2,0.99,1);
for $i (0..@c-1) {
  next unless $r < $c[$i];
  print $g[$i];
  last;
}
```

seeding

- PRNGs are pseudo-random because they produce exactly the same sequence for a given seed
  - makes debugging easier, since you can re-create the same sequence over and over
- if the PRNG is good, the seed should not matter
- if you do not seed your sequence, Perl will run srand() to do so
  - combination of time, process ID etc is used as a seed
  - if you call srand(), **call it only once**
- if you want a sequence that is hard to predict, use an unguessable seed
  - normally this is not cruicial, unless you're doing crypto

```
srand (time ^ $$ ^ unpack "%L*", `ps axww | gzip`);
```

- Netscape's SSL implementation was compromised because their choice of seed was very predictable (time of day + process ID + parent process ID)

http://www.cs.berkeley.edu/~daw/papers/ddj-netscape.html

/dev/random and /dev/urandom

- most UNIXes have /dev/random
  - a special system “device” that spits out random bits
  - kernel-based
  - based on variety of system characteristics that are extremely difficult to predict
    - environmental noise from device drivers
- /dev/random monitors entropy and blocks when entropy drops until entropy levels increase to produce sufficiently “random” bits
- /dev/urandom does not block and will produce as many bits as required
  - use /dev/random if you need crypto-strength bits

```
# get some random chars (0-255)
open(R,"/dev/random");
while(1) {
  read(R,$x,1); # read one byte at a time
  print unpack("C",$x); # display as number
}
```

http://linux.about.com/library/cmd/blcmdl4_urandom.htm

pseudo-randomness on CPAN

- as usual, there are modules offering PRNGs other than built-in rand()
- Math::Random
  - based on C randlib
  - uniform, normal, chi, exponential, poisson, gamma distributions and others
- Math::Random::MT
  - Mersenne Twister generator
  - period of 2^{19937}-1
- Math::TrulyRandom
  - uses timing of interrupts
  - circa 1996
  - I could not get this to work
- Net::Random
  - get random values from on-line sources (e.g. fourmilab.ch's HotBits)

`http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/earticles.html`

benchmarking

```
# seed the Mersenne Twister
use Math::Random::MT;
use Time::HiRes qw(gettimeofday tv_interval);
my $mt = Math::Random::MT->new(time);
# get time now
my $t0 = [gettimeofday];
# generate 1 million values
$mt->rand() for (0..1e7);
# compute numbers per second
print int($N/tv_interval($t)),"MT values per second";
```

- 250,000 MT values per second
- 400,000 randlib values per second
- 1,750,000 rand() values per second

String::Random

```
use String::Random;

$foo = new String::Random;
# 3 random digits - pattern set by regex
$foo->randregex('\d\d\d');
# 3 printable characters - pattern set by
$foo->randpattern("..."); # Prints 3 random printable characters
$foo->randpattern("CCcc!ccn")

c Any lowercase character [a-z]
C Any uppercase character [A-Z]
n Any digit [0-9]
! A punctuation character [~`!@$%^&*()-_+={}[]|\:;"'.<>?/#,]
. Any of the above
s A "salt" character [A-Za-z0-9./]
b Any binary data

$foo = new String::Random;
$foo->{'b'} = [ qw(a t g c) ];
$foo->randpattern("bbbbbb")

# aecd
print random_string("0101",
                    ["a", "b", "c"],
                    ["d", "e", "f"]);
```

4.1.2.2.1

Random Numbers and Distributions
Session 1

- pseudo-randomness is not easy
- next time, we'll see how to generate values from known and arbitrary probability distributions