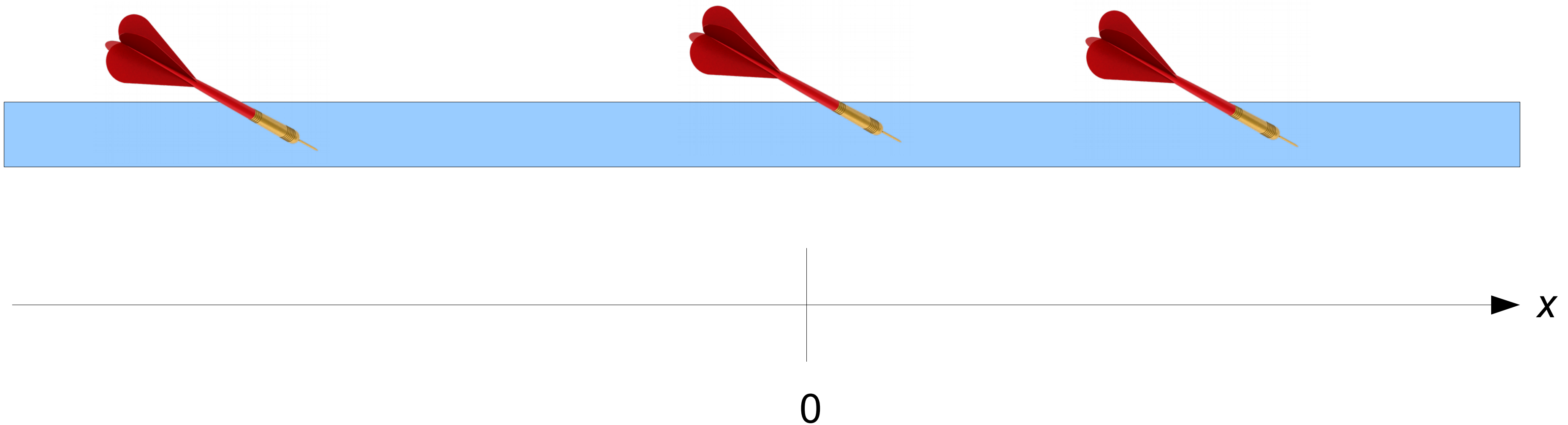
Class Progress

Basics of Linux, gnuplot, C
Visualization of numerical data
Roots of nonlinear equations
  (Midterm 1)
Solutions of systems of linear equations
Solutions of systems of nonlinear equations
Monte Carlo simulation
Interpolation of sparse data points
Numerical integration
  (Midterm 2)
Solutions of ordinary differential equations

Random vs. Pseudorandom



Imagine throwing "random" darts at a real number line. The sequence of *x*-coordinates represents an ideal random variable. We would expect the sequence of *x* values to exhibit:

Uniform sampling - all x values within a given range are equally likely

Uncorrelated - past values have no affect on the next value

Unpredictability - no way to predict the likelihood of a given value

} Only having these two properties is called "pseudorandom"

Modulus Operator

$$a \bmod b$$

Is the remainder left over when *a* is divided by *b* as integers
Examples:

$$7 \bmod 3 = 1$$
$$3 \bmod 16 = 3$$
$$16 \bmod 16 = 0$$
$$17 \bmod 16 = 1$$

In C code, get the integer dividend with the '/' operator, but the modulus with the '%' operator:

```
int i,j,k;
i = 7;
j = 3;
k = i / j;    /* assigns 2 to k */
k = i % j;    /* assigns 1 to k */
```

Generation of Pseudorandom Numbers

$$r_{i+1} = (a \cdot r_i + c) \bmod M$$

Examples:

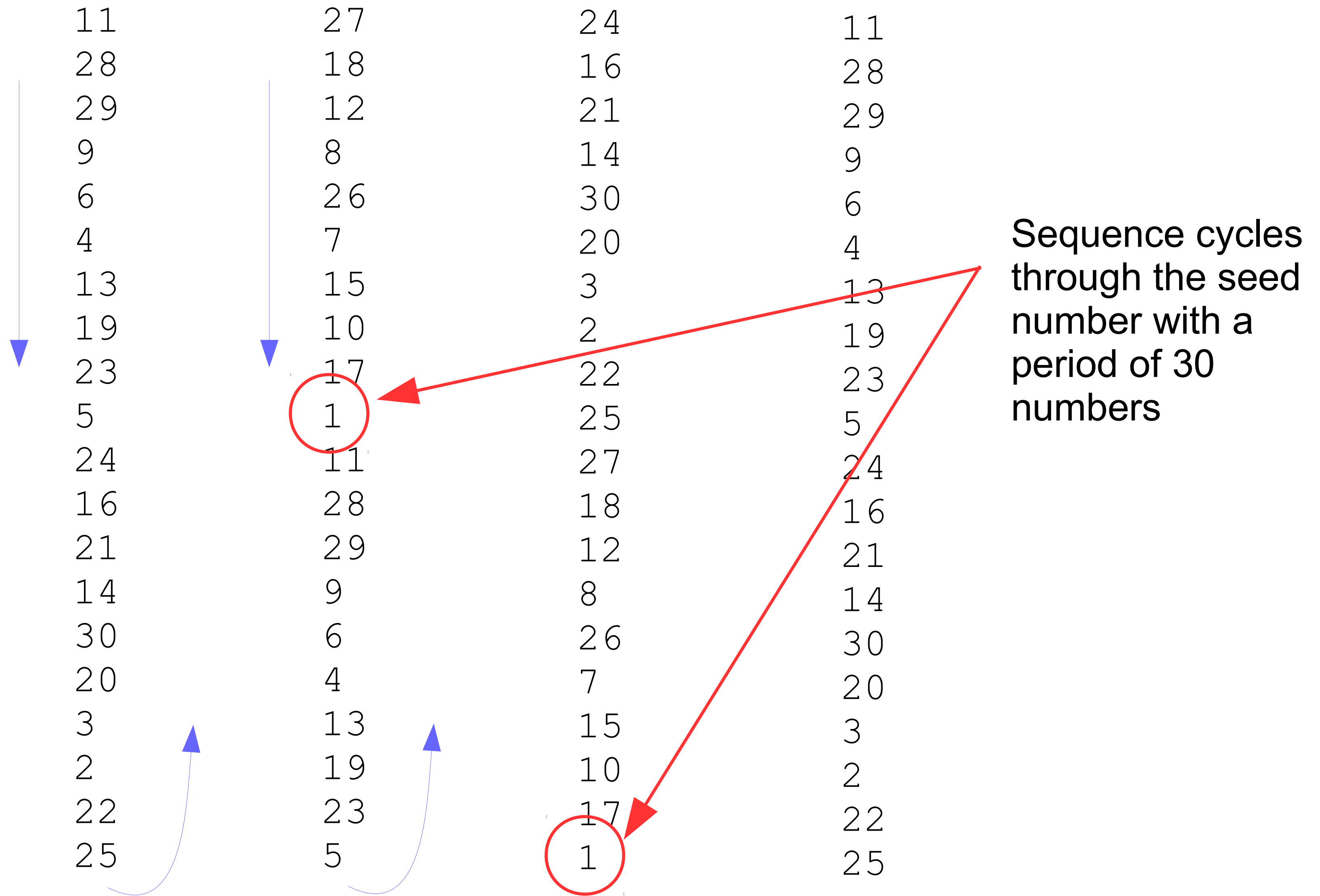
$$a = 11 \quad c = 0 \quad M = 2^5 - 1 = 31$$
$$r_{i+1} = (11 \cdot r_i) \bmod 31$$

$$a = 16807 \quad c = 0 \quad M = 2^{31} - 1 = 2147483647$$
$$r_{i+1} = (16807 \cdot r_i) \bmod 2147483647$$

$$a = 69069 \quad c = 1 \quad M = 2^{32} = 4294967296$$
$$r_{i+1} = ((69069 \cdot r_i) + 1) \bmod 4294967296$$

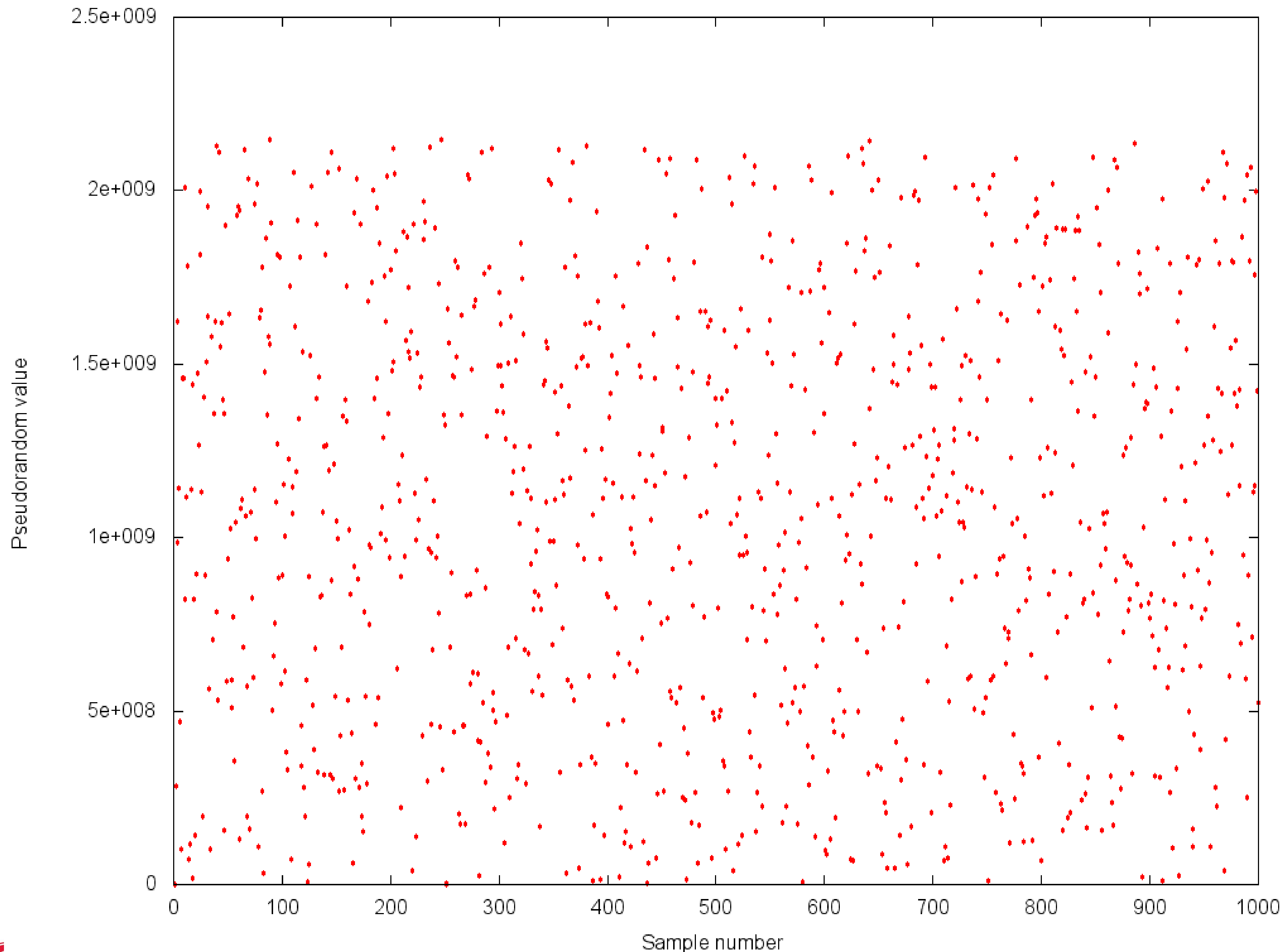
Note: Don't seed this type of generator with c=0 with 0 !!
Select *a*, *c* and *M* so that *M*-1 unique numbers generated, excluding one disallowed value, and period = *M*-1

Example for M=31, seed=1

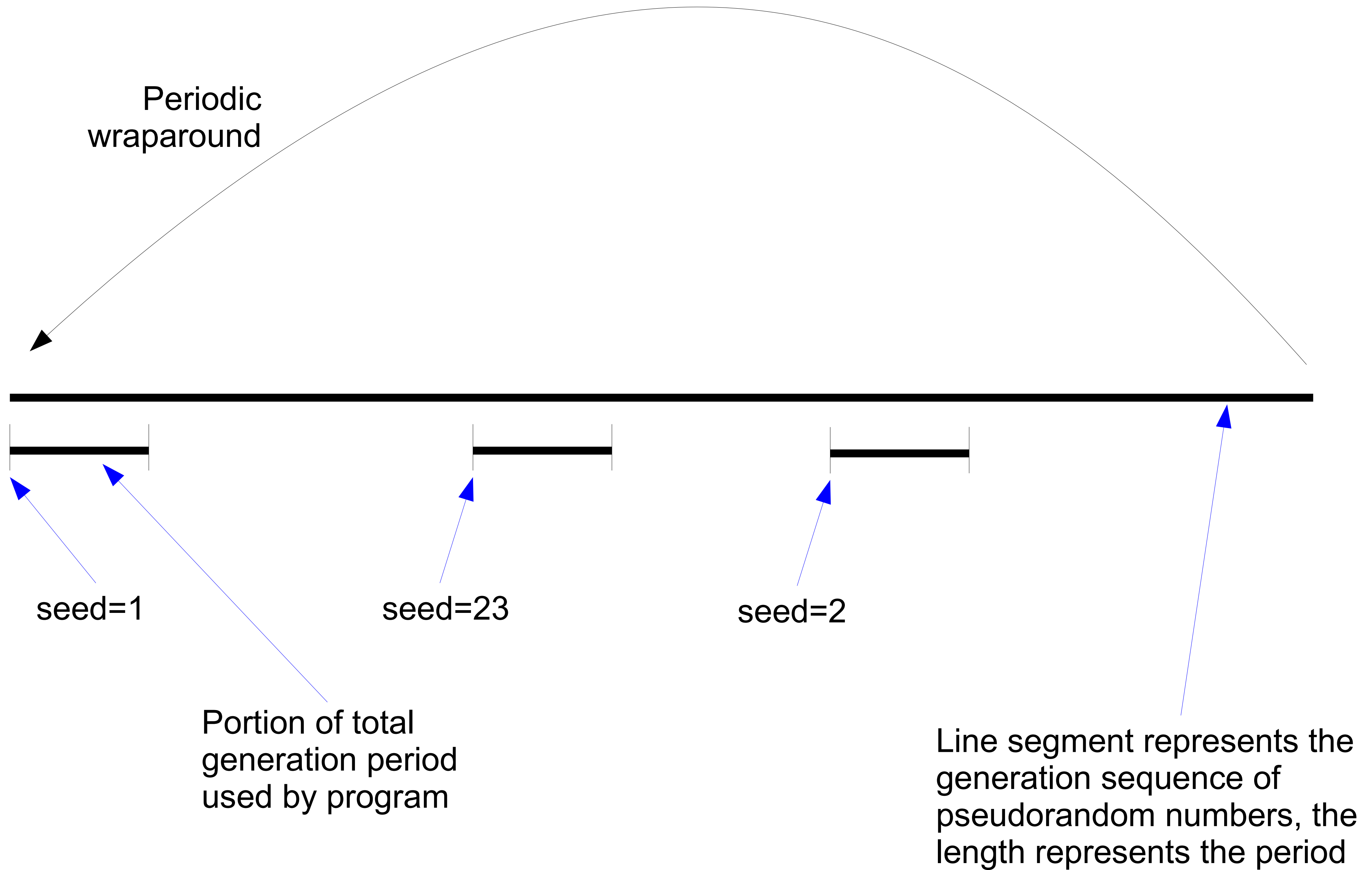


Period = 30, that is, in 30 cycles the seed will occur again

Example for $M=2^{31}-1$, seed=1, 1000 samples



Seeding Pseudorandom Number Generators



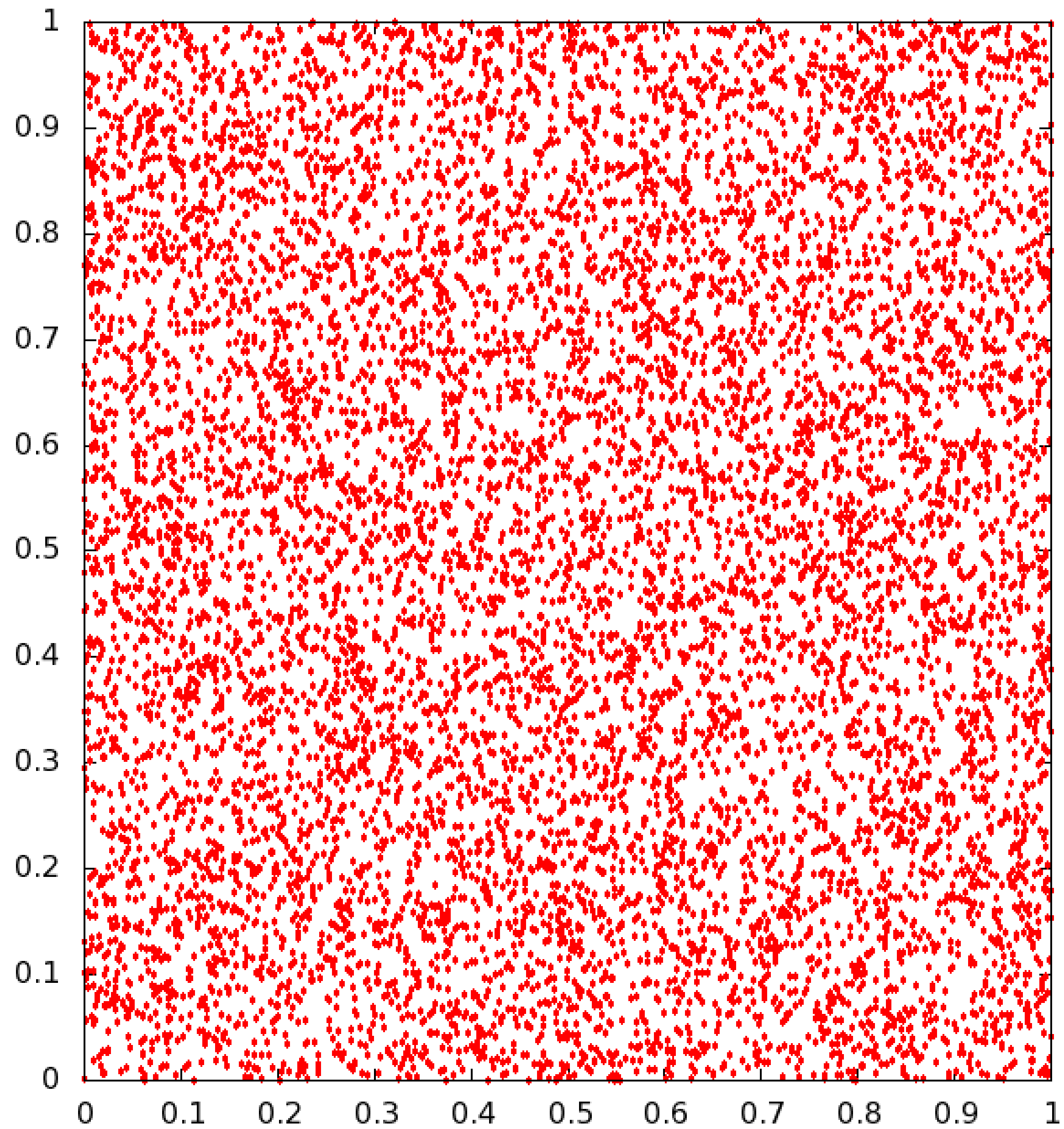
Pseudorandom Generator C Code

```
#include <stdio.h>
#include <stdlib.h>
#include "random.h"

static unsigned long int random_state = 1;

void random_seed(unsigned long int seed) {
  random_state = seed;
  return;
}

double random_gen(void) {
  unsigned long long int temp_state;

  temp_state = random_state;
  temp_state = (16807ULL * temp_state) % 2147483647ULL;
  random_state = temp_state;
  return(((double) random_state) * 4.6566128752459e-10);
}
```

Instead of returning an integer, return a floating point number between 0.0 and 1.0, which is more useful for numerical work

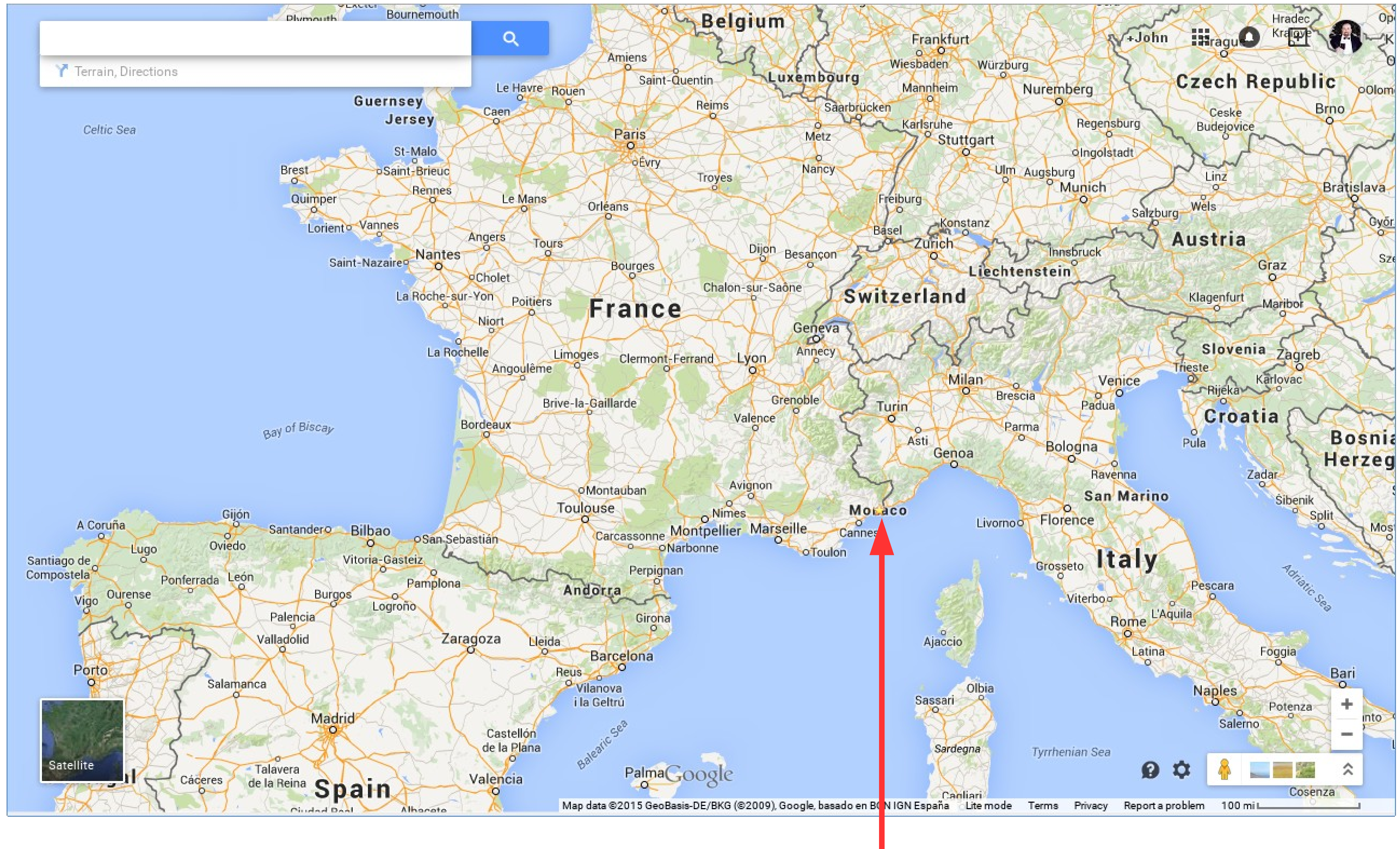
Graphical Correlation Test

10,000 points with $x = r_{2i}$ and $y = r_{2i+1}$

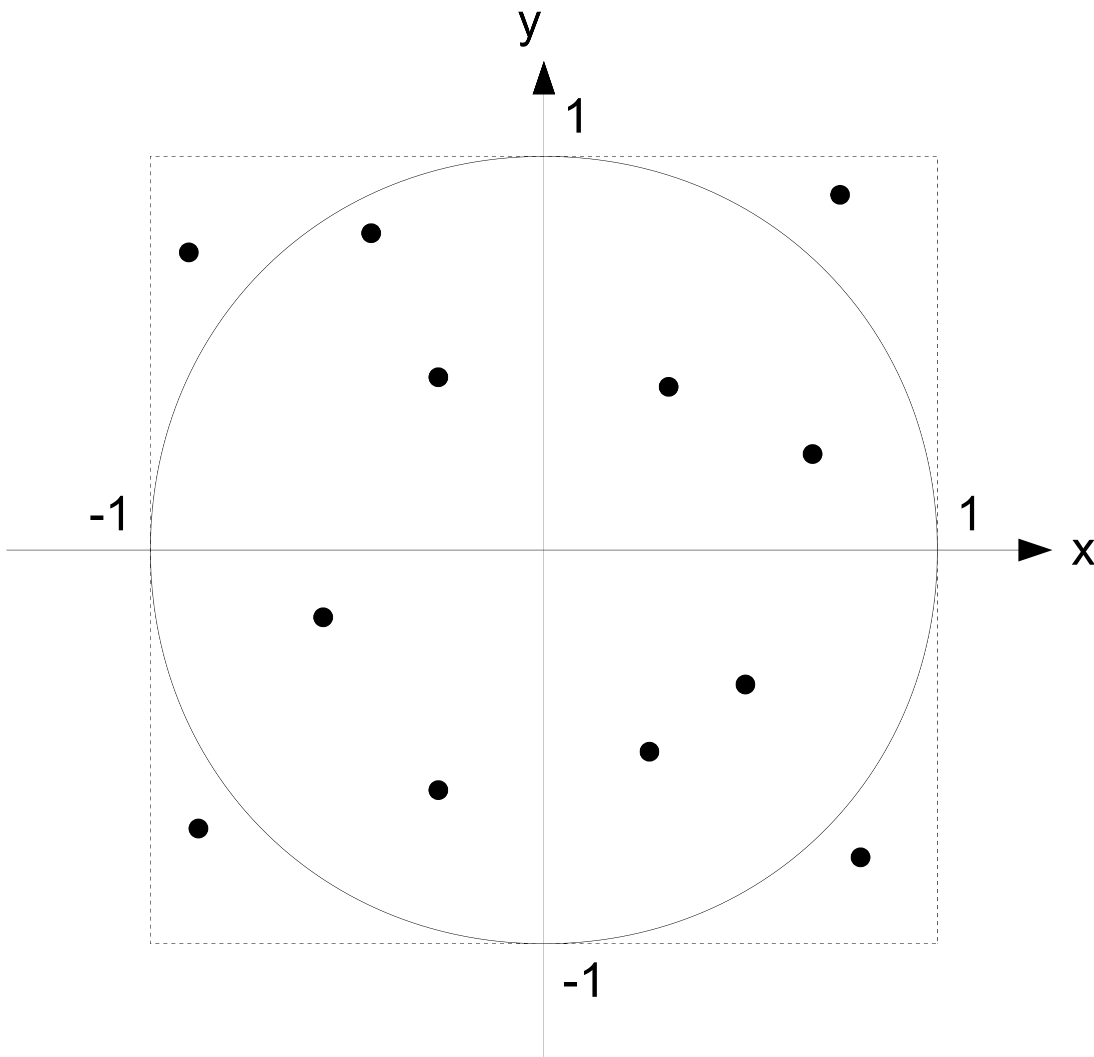


What Is This Building?

Hint: Its Location...



Monte Carlo Simulation of the Value of π



Generate N points (where N is large) with random x and y coordinates within the range $-1 < x < +1$, $-1 < y < +1$

Count m points that land inside circle of radius 1, with

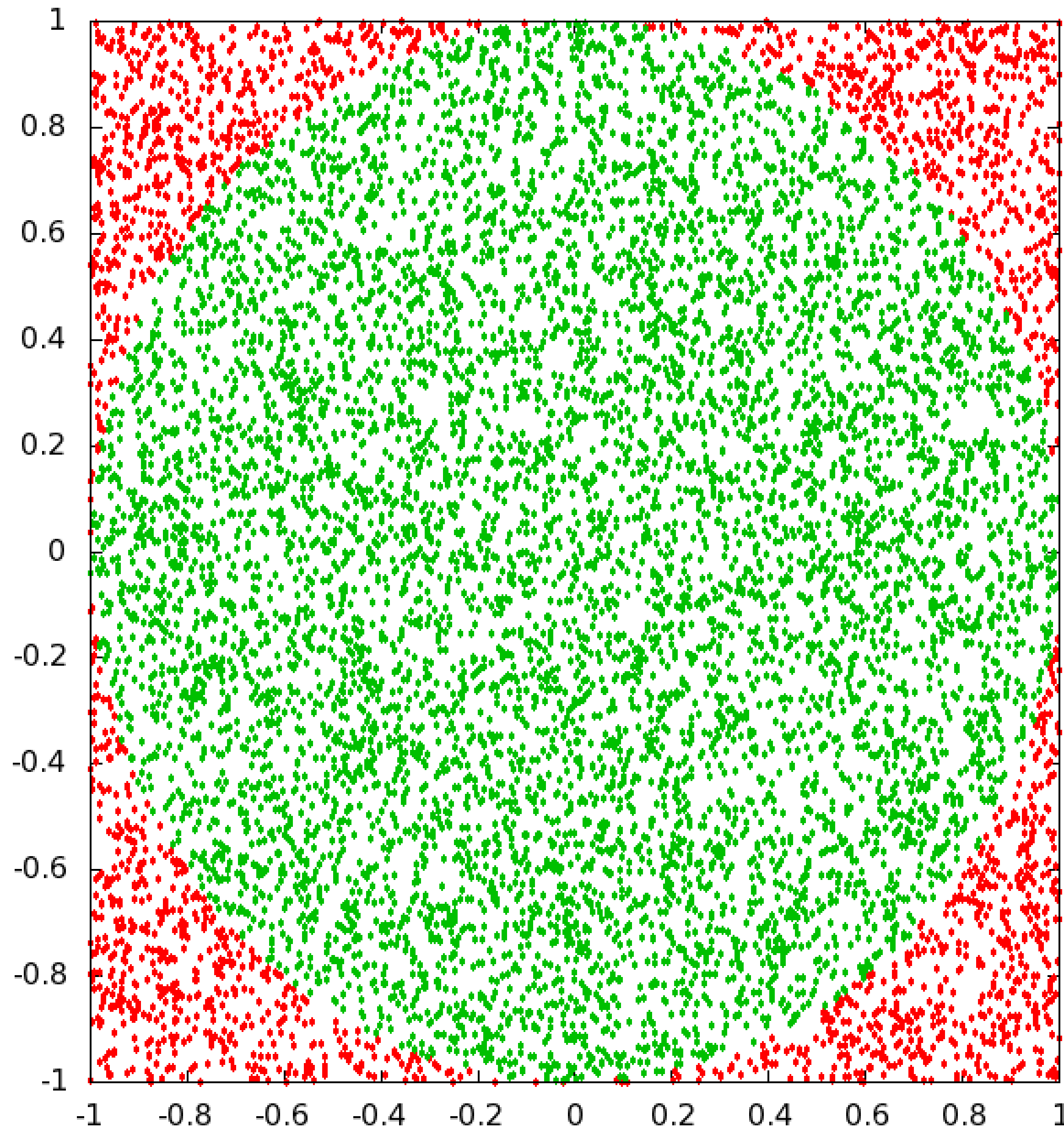
$$\sqrt{x^2 + y^2} \leqslant 1$$

$$\frac{m}{N} \approx \frac{\text{Area of circle}}{\text{Area of square}} = \frac{\pi}{4}$$

$$\text{So } \pi \approx 4\frac{m}{N}$$

Monte Carlo Simulation of the Value of π

green points are within circle, red points are outside



Generating Random Points on a Unit Circle

set x to a uniform random variable between -1 and 1
set y to a uniform random variable between -1 and 1
this sample is a point (x, y) somewhere in the unit square
calculate $R_{sample} = \sqrt{x^2 + y^2}$
if $R_{sample} > 1$ discard this sample and start over with a different sample
otherwise extend sample onto unit circle with

$$x \leftarrow \frac{x}{R_{sample}}$$

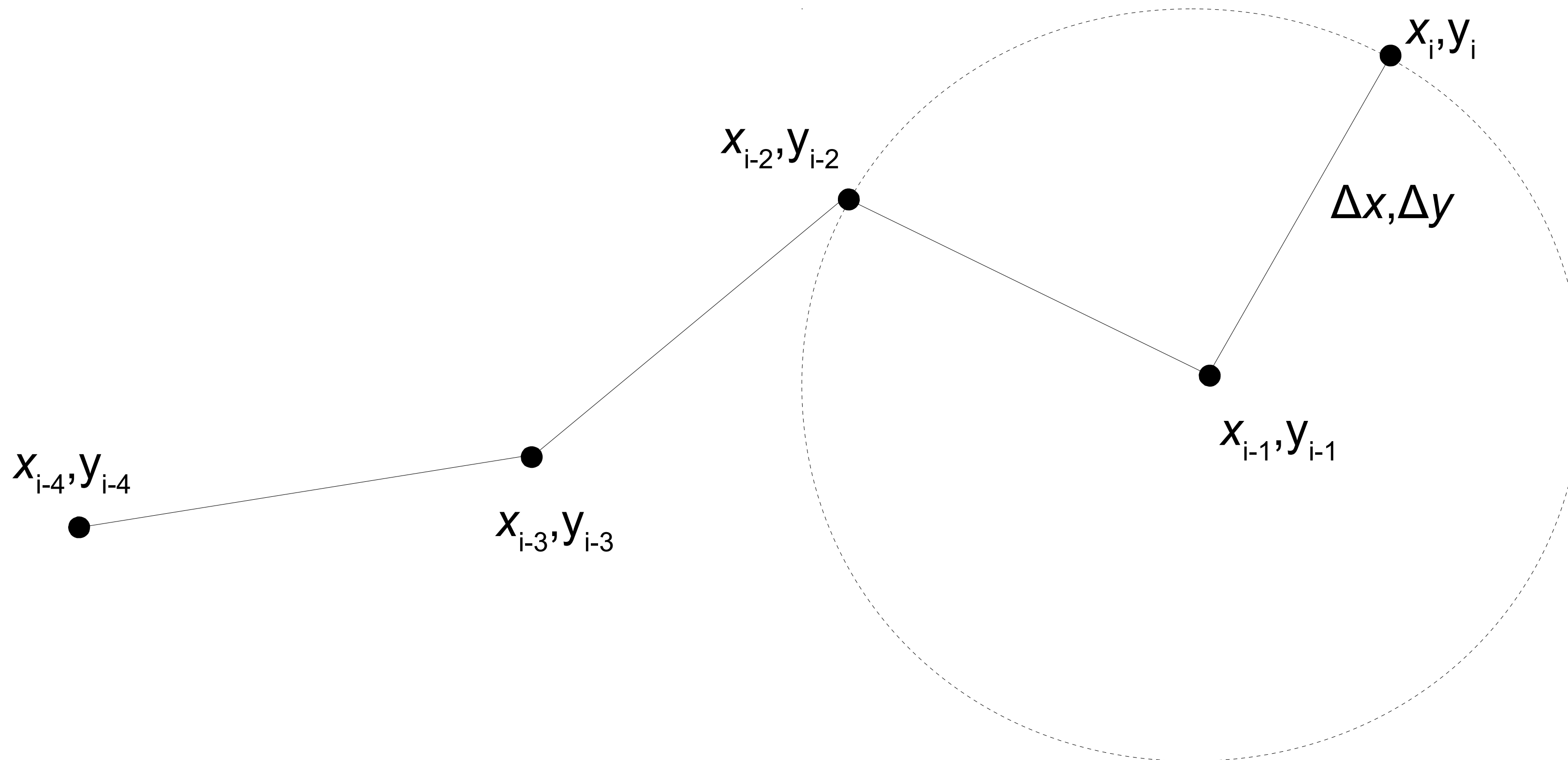
$$y \leftarrow \frac{y}{R_{sample}}$$

Two-Dimensional Random Walk

Initialize $x_0 = 0, y_0 = 0$

Generate Δx and Δy on unit circle

Update $x_i = x_{i-1} + \Delta x$ and $y_i = y_{i-1} + \Delta y$



Two-Dimensional Random Walk

D = Total distance traveled

$$D^2 = (\Delta x_1 + \Delta x_2 + \Delta x_3 + \cdots + \Delta x_n)^2 + (\Delta y_1 + \Delta y_2 + \Delta y_3 + \cdots + \Delta y_n)^2$$

$$\begin{aligned} &= \Delta x_1^2 + \Delta x_2^2 + \Delta x_3^2 + \cdots + \Delta x_n^2 \\ &+ 2\Delta x_1 \Delta x_2 + 2\Delta x_1 \Delta x_3 + \cdots \\ &+ \Delta y_1^2 + \Delta y_2^2 + \Delta y_3^2 + \cdots + \Delta y_n^2 \\ &+ 2\Delta y_1 \Delta y_2 + 2\Delta y_1 \Delta y_3 + \cdots \end{aligned}$$

For large n, cross terms will cancel out, and on average

$$D^2 \approx \langle \Delta x_1^2 + \Delta x_2^2 + \Delta x_3^2 + \cdots + \Delta x_n^2 + \Delta y_1^2 + \Delta y_2^2 + \Delta y_3^2 + \cdots + \Delta y_n^2 \rangle$$

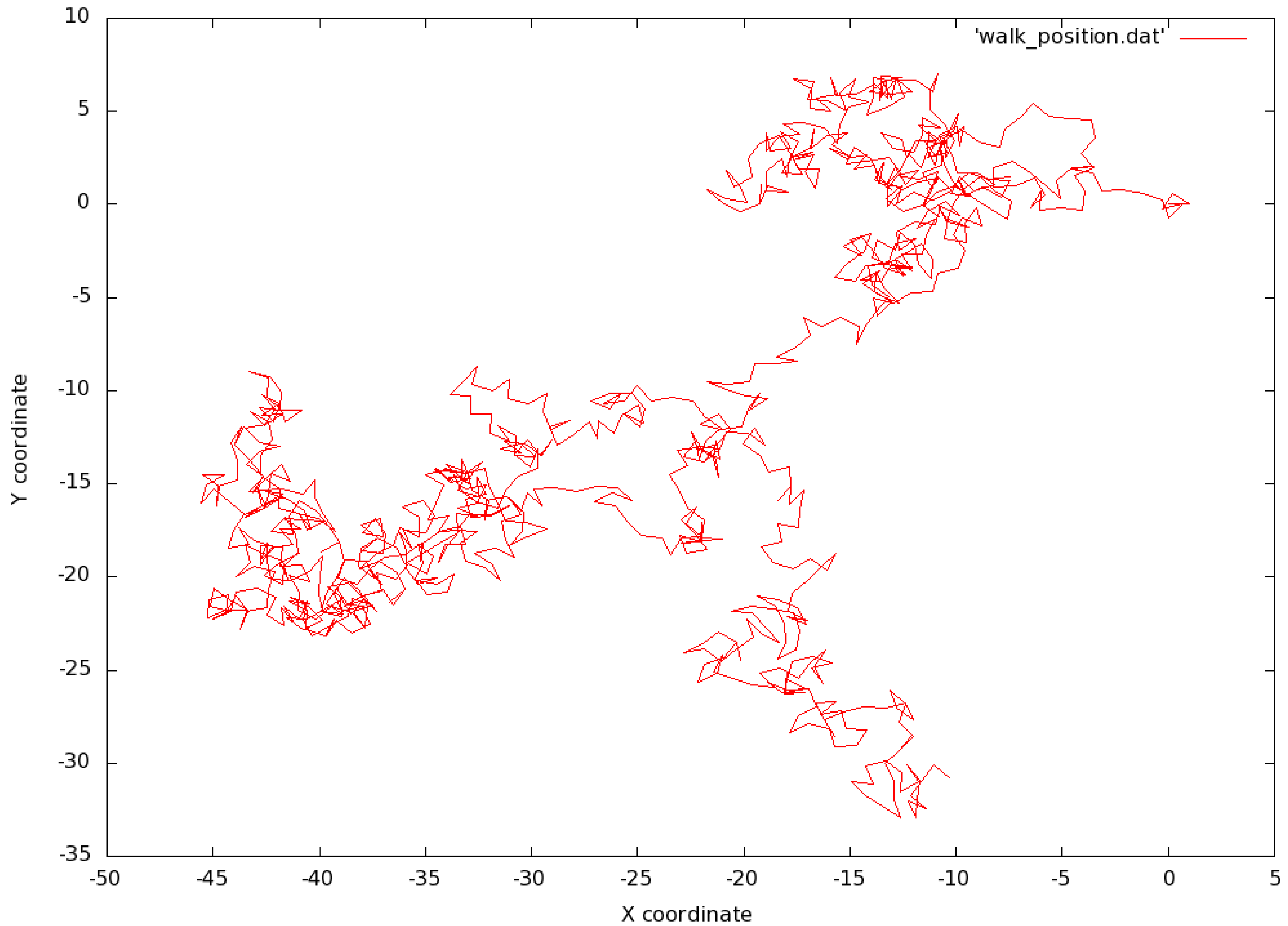
$$= n \cdot \langle r^2 \rangle$$

So $D \approx \sqrt{n}\, r_{rms}$

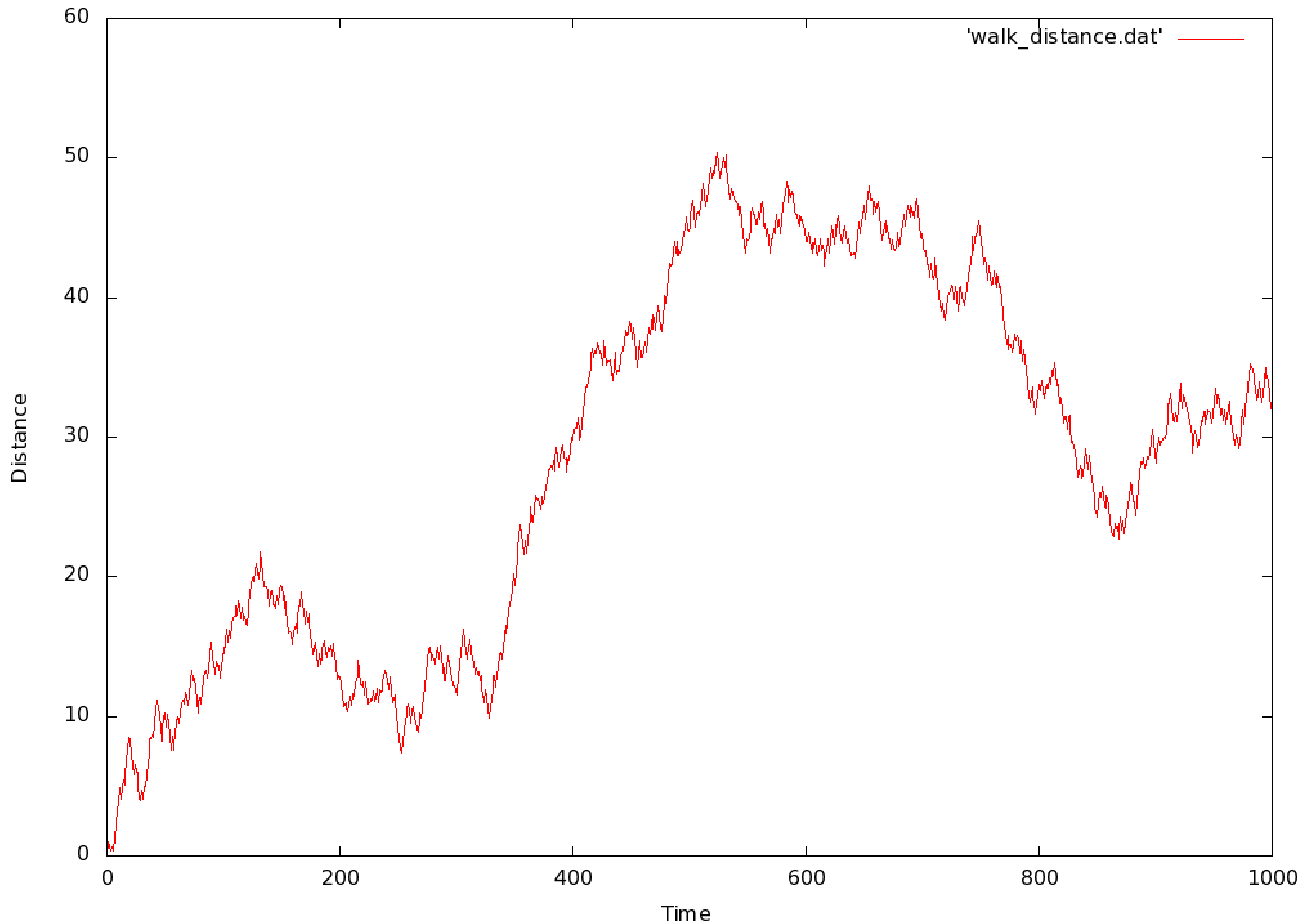
Random Walk C Code in Lab Exercise

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "random.h"

int main(int argc,char *argv[]) {
  double x,y,dx,dy,radius;
  int n_steps,i_step;
  FILE *position,*distance;
  if (argc != 3) {
    fprintf(stderr,"%s <N steps> <seed>\n",argv[0]);
    exit(1);
  }
  position = fopen("walk_position.dat","w");
  distance = fopen("walk_distance.dat","w");
  n_steps = atoi(argv[1]);
  random_seed(atoi(argv[2]));
  fprintf(position,"0 0\n");
  fprintf(distance,"0 0\n");
  i_step = 0;
  x = 0.0;
  y = 0.0;
  while (i_step < n_steps) {
    dx = ... ;
    dy = ... ;
    radius = ... ;
    if (radius > 1.0) continue;
    dx = ... ;
    dy = ... ;
    x += dx;
    y += dy;
    fprintf(position,"%.8f %.8f\n",x,y);
    fprintf(distance,"%d %.8f\n",i_step + 1,sqrt((x*x) + (y*y)));
    i_step++;
  }
  fclose(position);
  fclose(distance);
  exit(0);
}
```

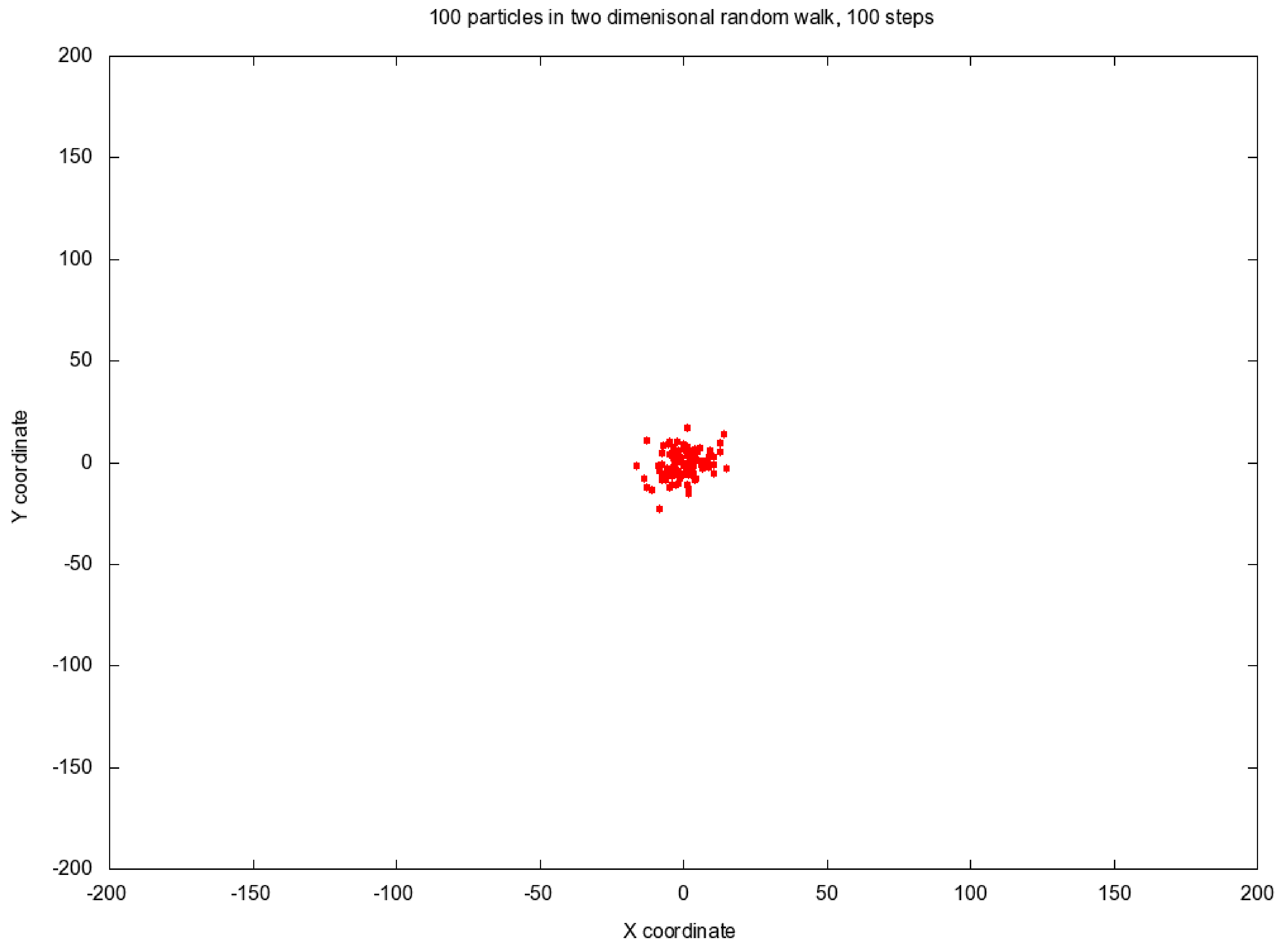
Random Walk



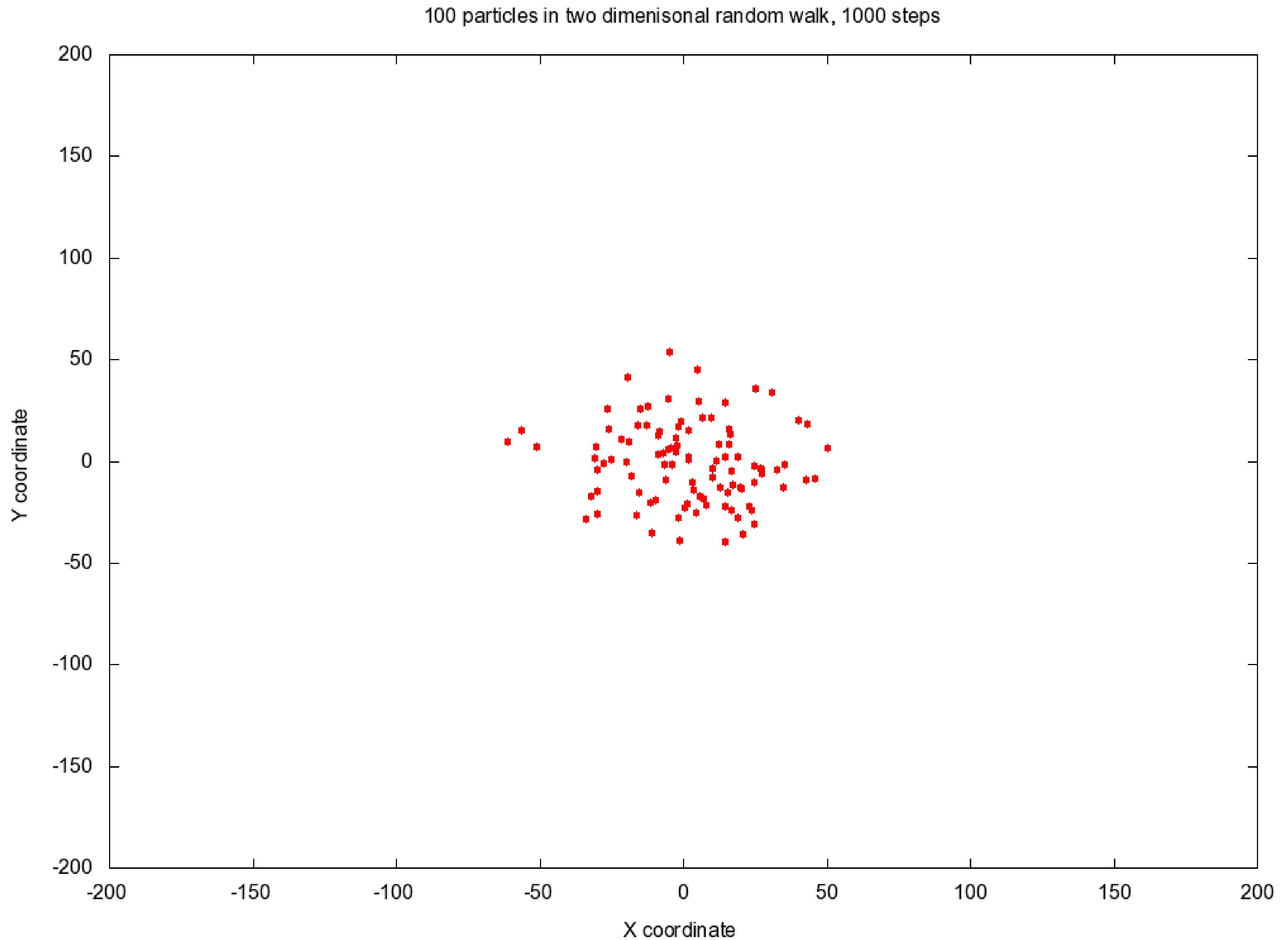
Random Walk



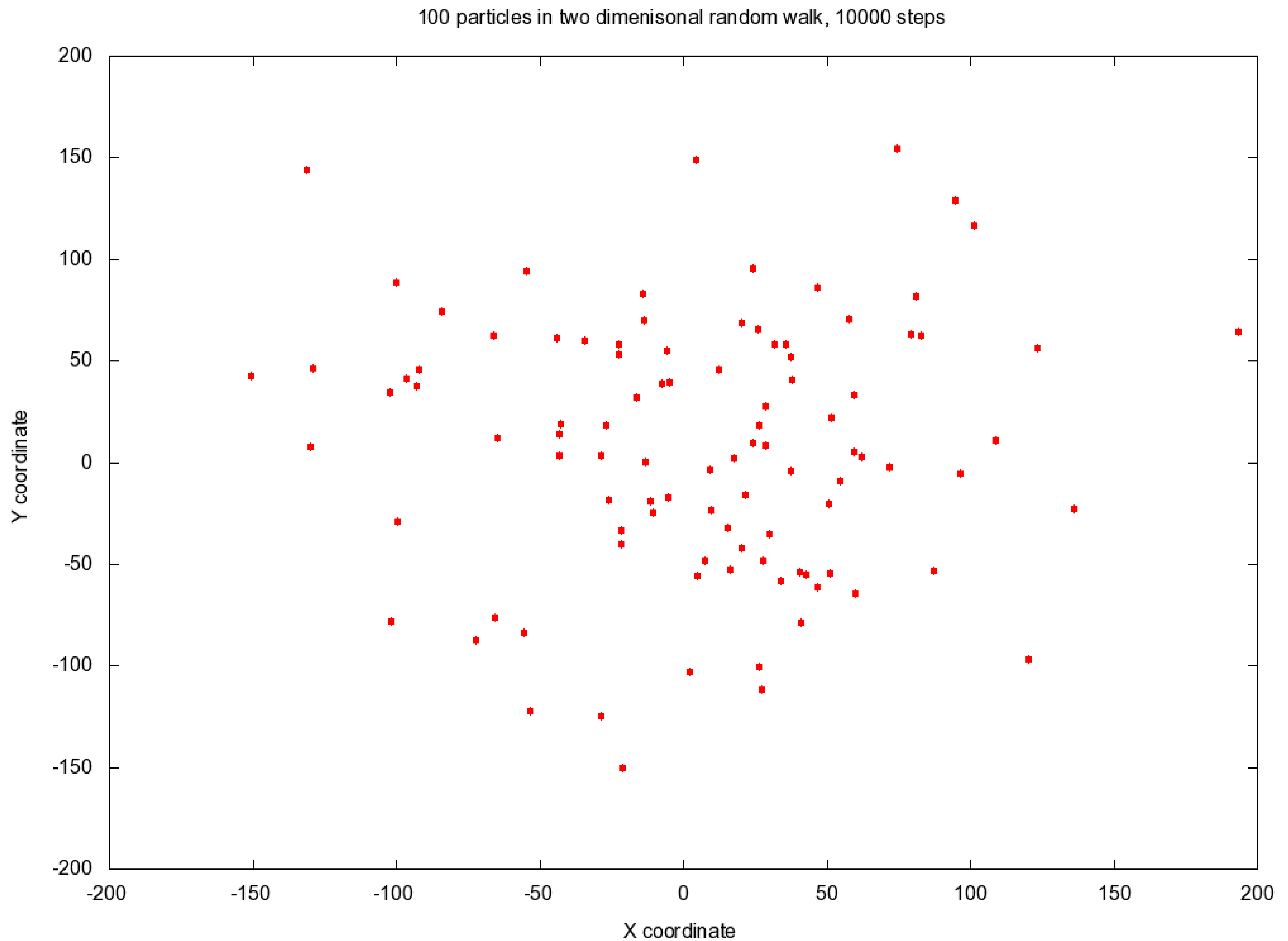
100 Particles in Random Walks



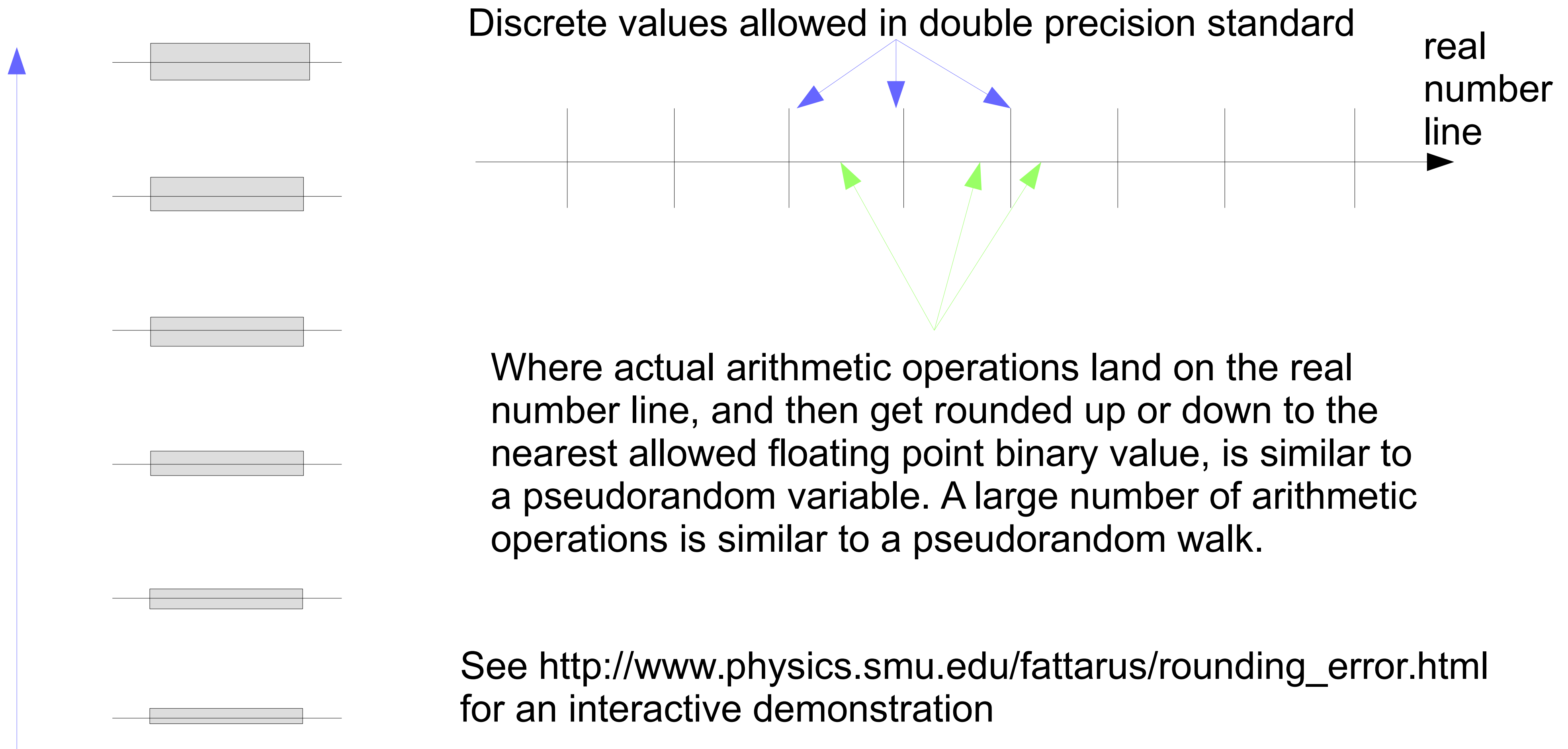
100 Particles in Random Walks



100 Particles in Random Walks



Using Floating Point Variables with Comparison Operators



Sequence of real values intended for an iterative loop, roundoff error bands subject to random walk

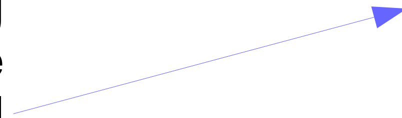
Example Numerical Random Walks

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  double x;

  x = 0.0;
  while (x <= 1.0) {
    printf("%.16f\n",x);
    x = x + 0.01;
  }
  exit(0);
}
```

Output:

```
0.0000000000000000
0.0100000000000000
0.0200000000000000
  . . .
0.3500000000000001
0.3600000000000002
0.3700000000000002
  . . .
0.7500000000000004
0.7600000000000005
0.7700000000000005
  . . .
0.9700000000000006
0.9800000000000006
0.9900000000000007
```

This walk with rounding errors ended up on the high side, so no final iteration at the 1.00 value coder intended!



Example Numerical Random Walks

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  double x;

  x = 0.0;
  while (x <= 1.0) {
    printf("%.16f\n",x);
    x = x + 0.0025;
  }
  exit(0);
}
```

Output:

```
0.0000000000000000
0.0025000000000000
0.0050000000000000
   . . .
0.2300000000000001
0.2325000000000002
0.2350000000000002
   . . .
0.4800000000000004
   . . .
0.5025000000000003
0.5050000000000002
0.5075000000000002
0.5100000000000001
0.5125000000000001
0.5150000000000000
0.5175000000000000
0.5199999999999999
0.5224999999999999
   . . .
0.9949999999999898
0.9974999999999897
0.9999999999999897
```

This walk starts off running over the expected value, but then staggers back to well under value at the end

Radioactive Decay

Start with a sample of material with N_0 radioactive nuclei at time t = 0.
At a given time in the future, the population of radioactive nuclei is $N(t)$.

Assume the probability of radioactive decay is such that the decay rate is

$$-\frac{dN}{dt} = \lambda N$$

$$\frac{dN}{N} = -\lambda\, dt$$

$$\int_{N_0}^{N} \frac{dN}{N} = -\lambda \int_0^t dt$$

$$\ln(N) - \ln(N_0) = -\lambda t$$

$$N(t) = N_0 e^{-\lambda t}$$

Radioactive Decay

But the measured quantity in a lab, for example with a Geiger counter, is the decay rate *R(t)*.

$$R(t) = -\frac{dN(t)}{dt} = \lambda N_0 e^{-\lambda t}$$

$$R(t) = R_0 e^{-\lambda t}$$

where R_0 is the measured decay rate at time t=0. We can experimentally find λ by taking regular measurements of the decay rate $R(t)$ and performing a best linear fit with the logarithmic relationship

$$\ln R(t) = \ln R_0 - \lambda t$$

Usually a decay rate is specified as the “half-life” of a species, that is, the time for half the population to decay.

$$T_{1/2} = \frac{\ln(2)}{\lambda}$$

Discrete Approximation to Radioactive Decay

Interpret
$$-\frac{dN}{N} = \lambda\, dt$$

as meaning "the probability of a given nucleus decaying in a time interval *dt* is λ"
Pseudocode for a discrete approximation algorithm:

```
integer time,left,decay,i; real test,lambda;

time ← 0
left ← initial population;
while left > 0 do
  decay ← 0;
  for i = 1 to left do
    test ← random number between 0.0 and 1.0
    if test <= lambda then
      decay ← decay + 1
    end if
  end for
  output time,decay,left
  left ← left - decay
  time ← time + 1
end while
```

Why is the Night Sky Dark?

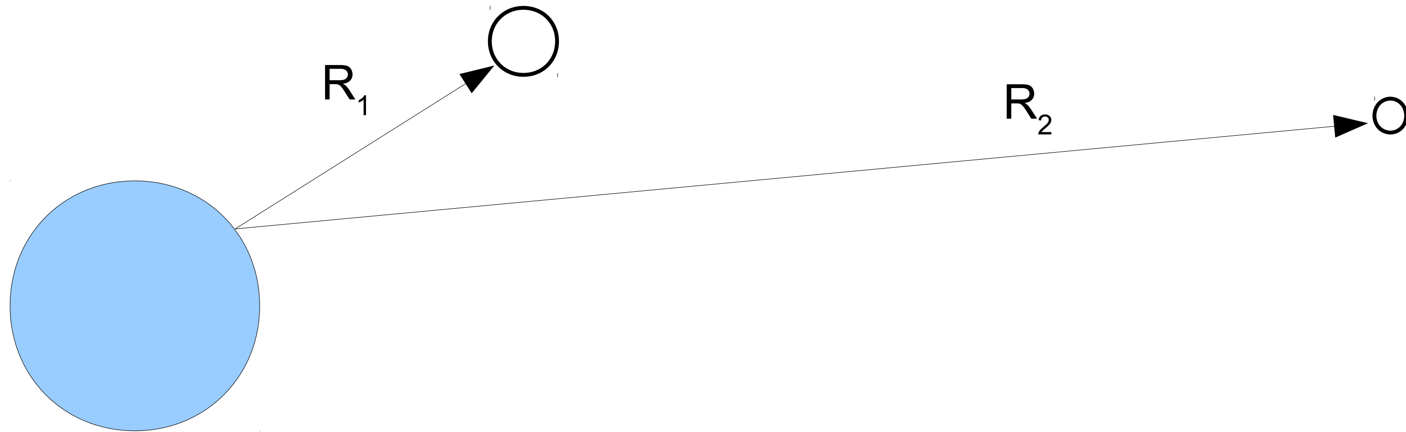
Perseids meteor shower, August 2016

Aurora Borealis, January 2016

Olbers' Paradox

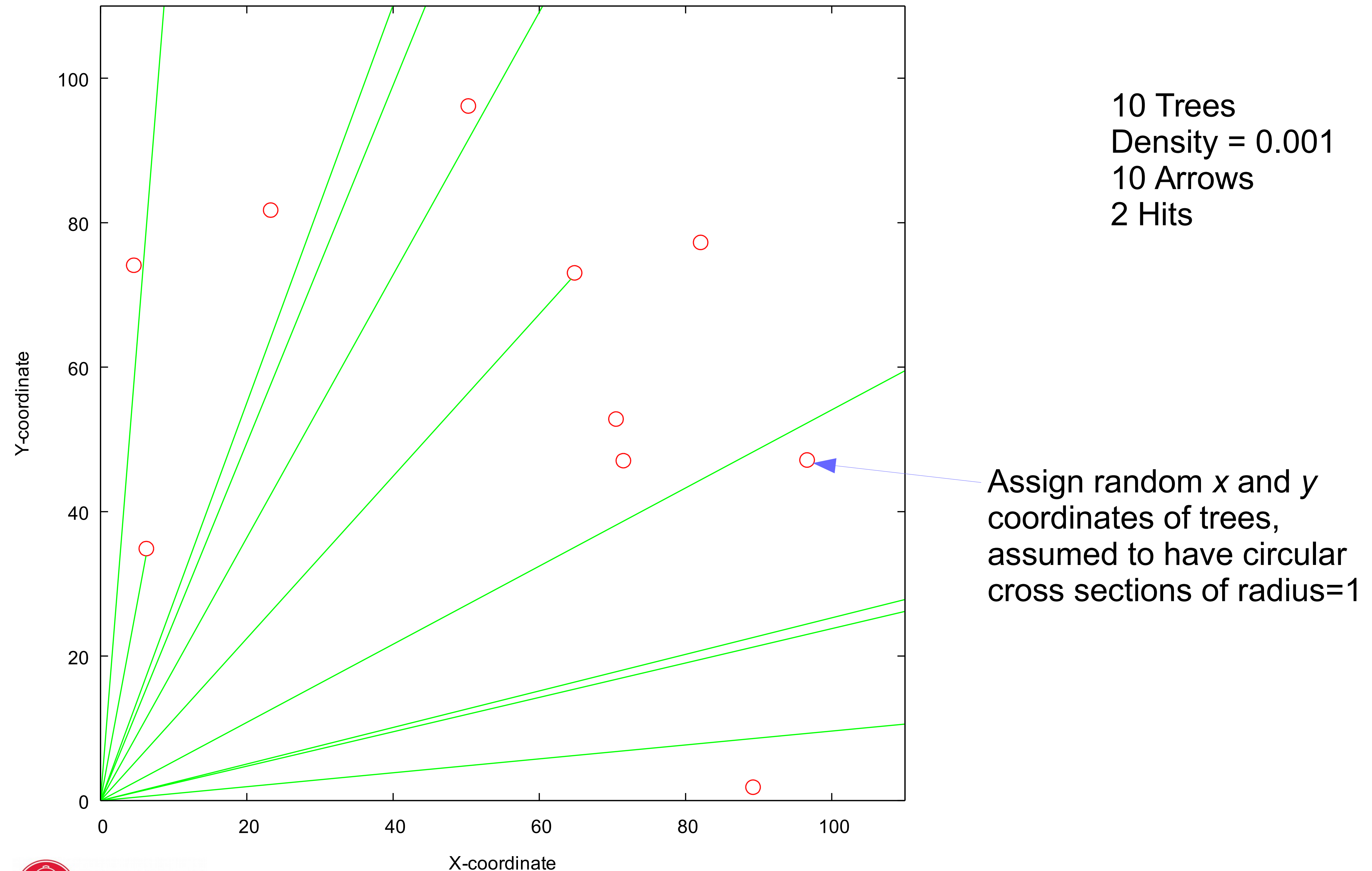
If the universe were of static size and infinitely old, why is the night sky dark, instead of filled with a uniform brightness equivalent to the average star?

The light intensity from a star a distance R from earth decreases as $1/R^2$, but the circular disc of area obscured by the star also decreases as $1/R^2$

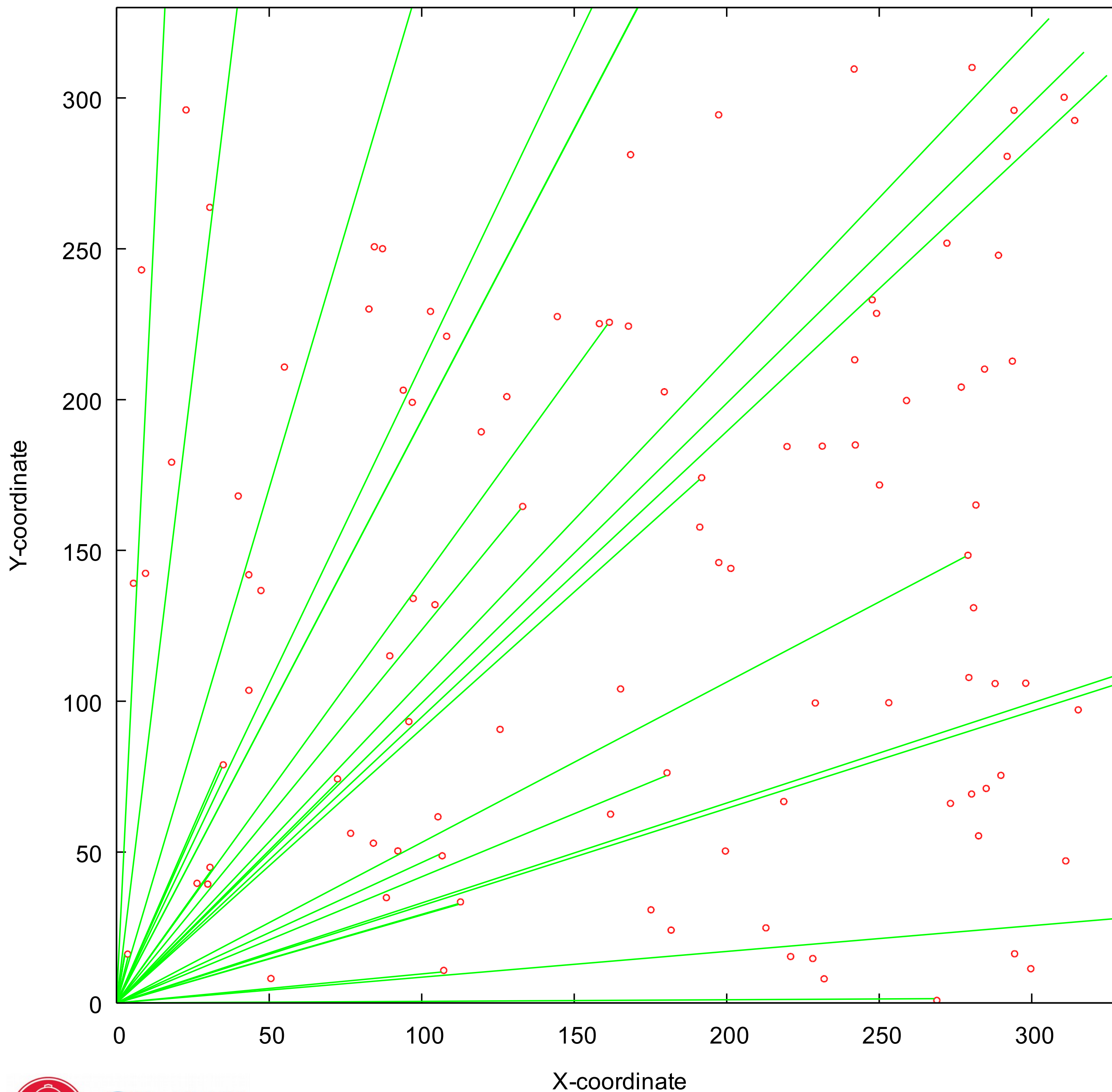


Do the observed discs from the stars distributed with uniform average density throughout the volume of a large universe cover all the observed sky area? This is equivalent to the problem of shooting arrows in Sherwood Forest. Is it possible for any arrows to emerge from the forest without getting stuck in a tree?

Shooting Random Arrows in a Random Forest

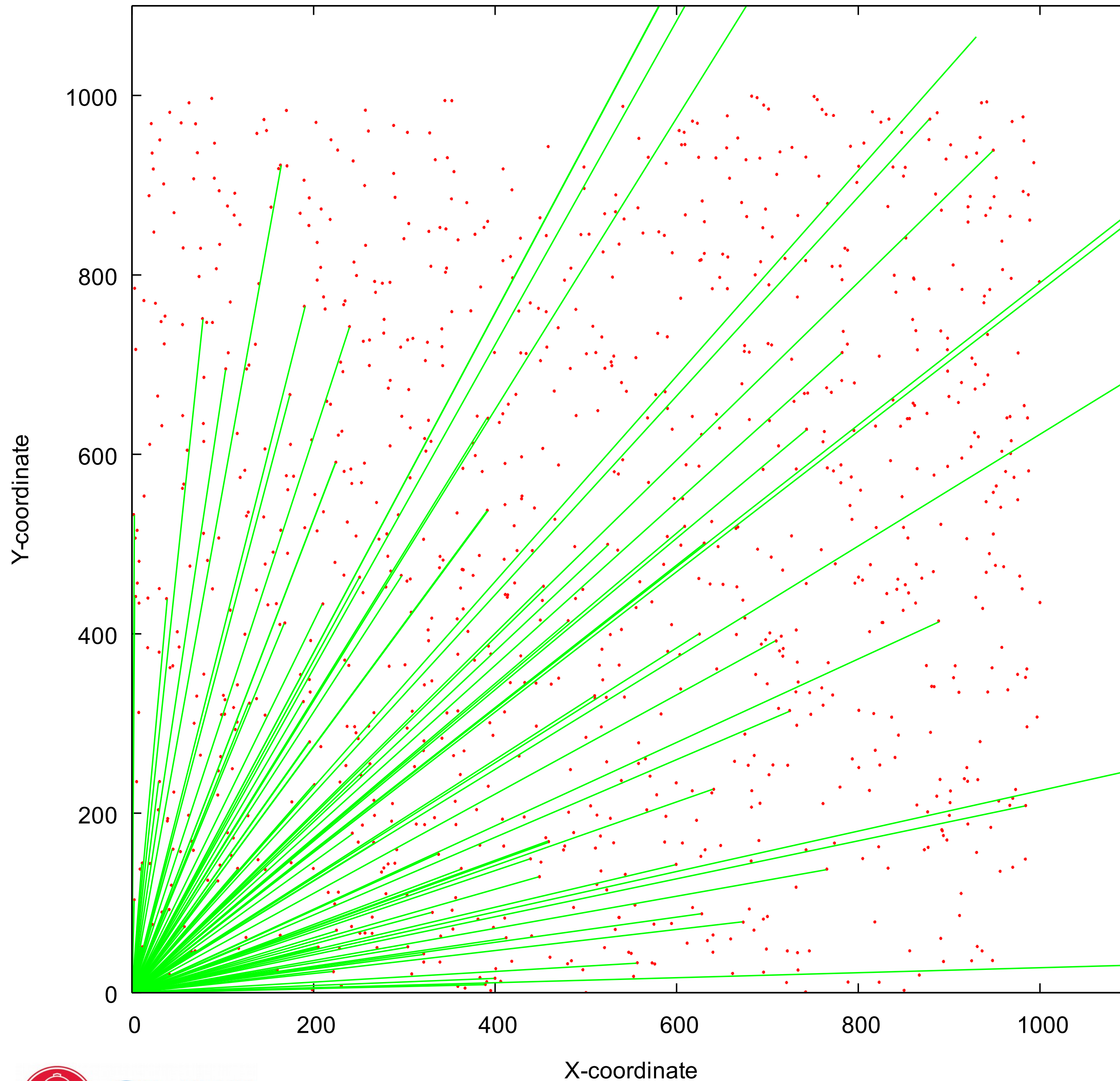


Shooting Random Arrows in a Random Forest



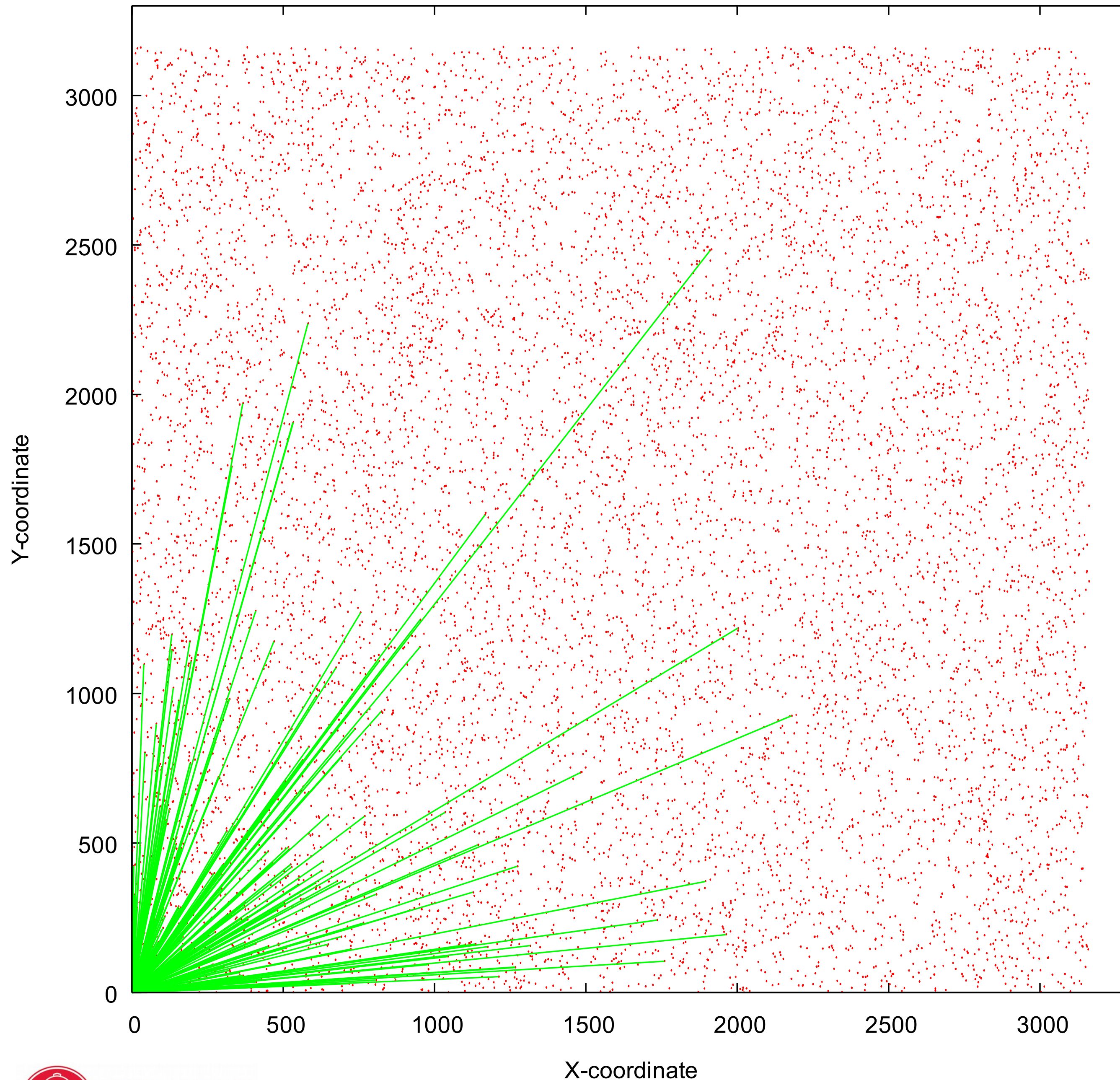
100 Trees
Density = 0.001
30 Arrows
18 Hits

Shooting Random Arrows in a Random Forest



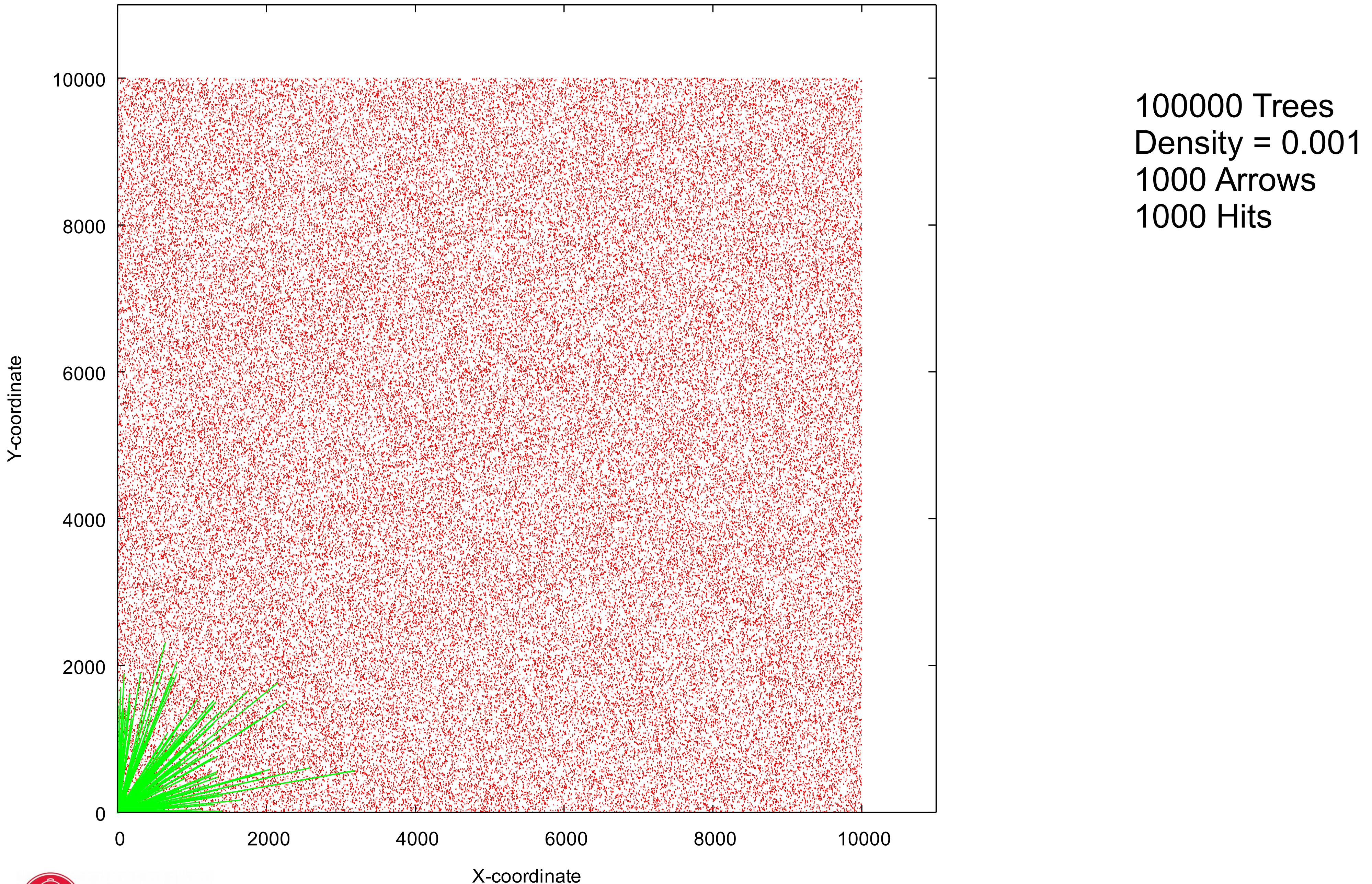
1000 Trees
Density = 0.001
100 Arrows
90 Hits

Shooting Random Arrows in a Random Forest



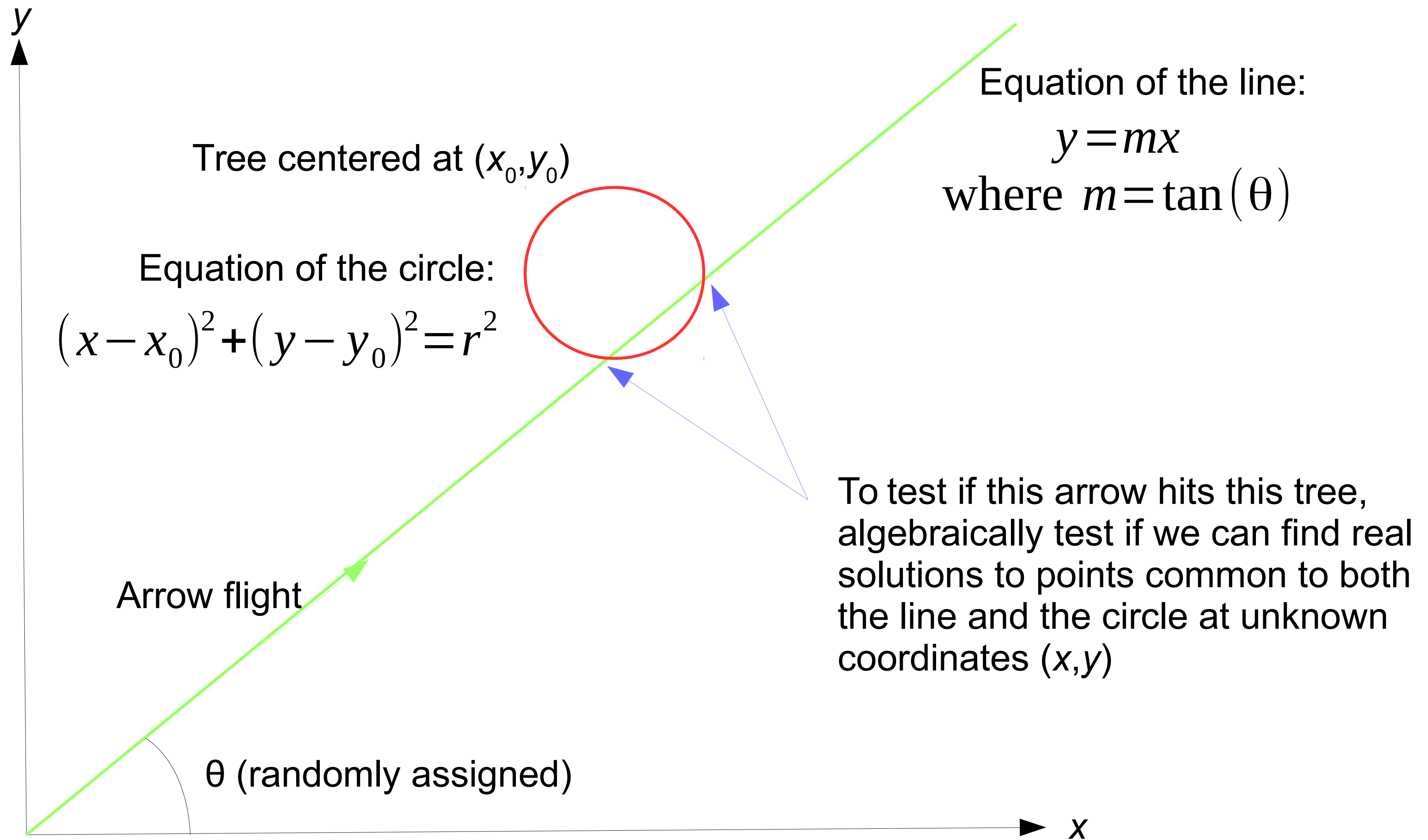
10000 Trees
Density = 0.001
300 Arrows
300 Hits

Shooting Random Arrows in a Random Forest

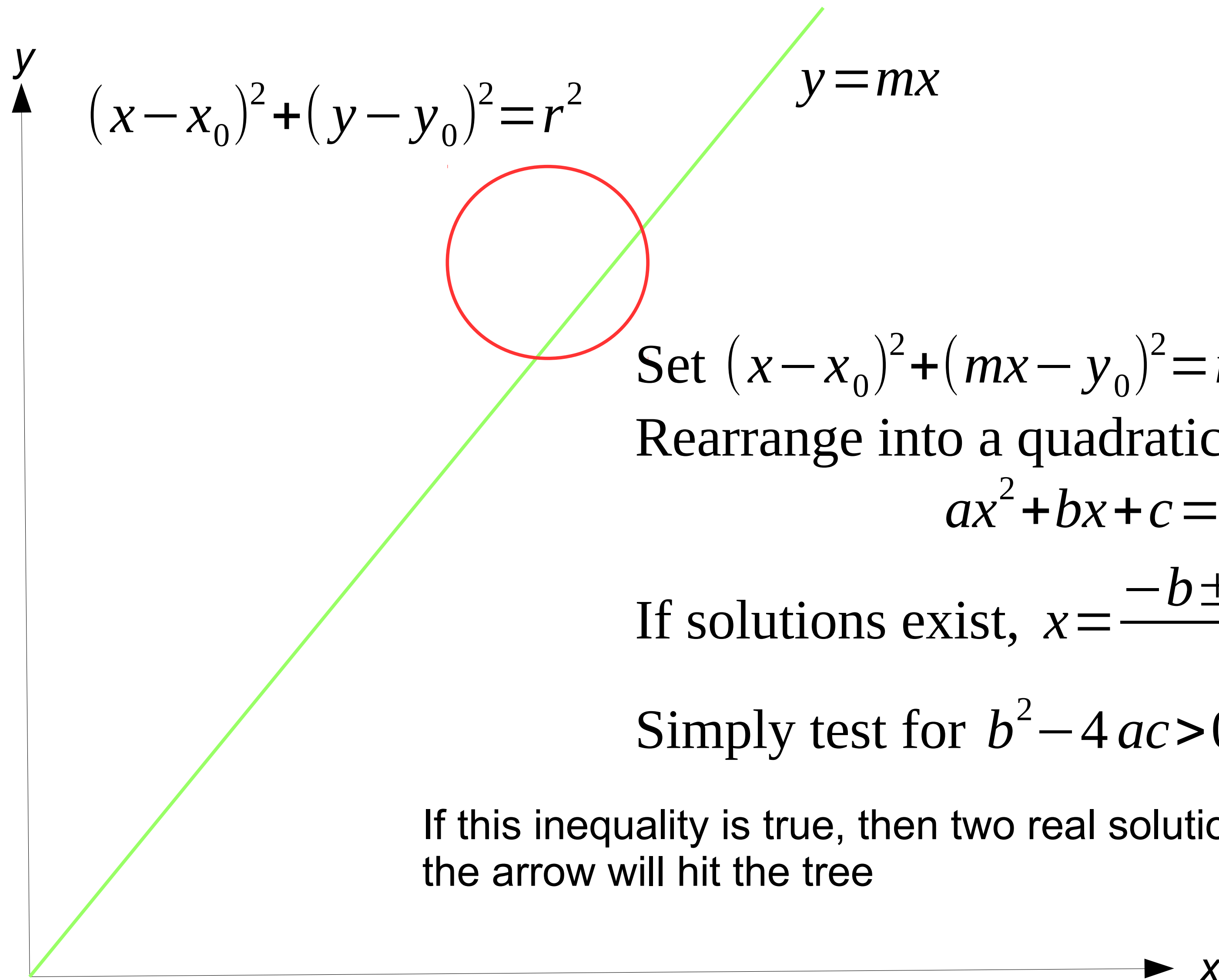


100000 Trees
Density = 0.001
1000 Arrows
1000 Hits

Testing for an Arrow Hit



Testing for an Arrow Hit



Set $(x-x_0)^2+(mx-y_0)^2=r^2$

Rearrange into a quadratic equation in x:

$$ax^2+bx+c=0$$

If solutions exist, $x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$

Simply test for $b^2-4ac>0$

If this inequality is true, then two real solutions exist for *x*, and the arrow will hit the tree

Generating Random Points on a Sphere

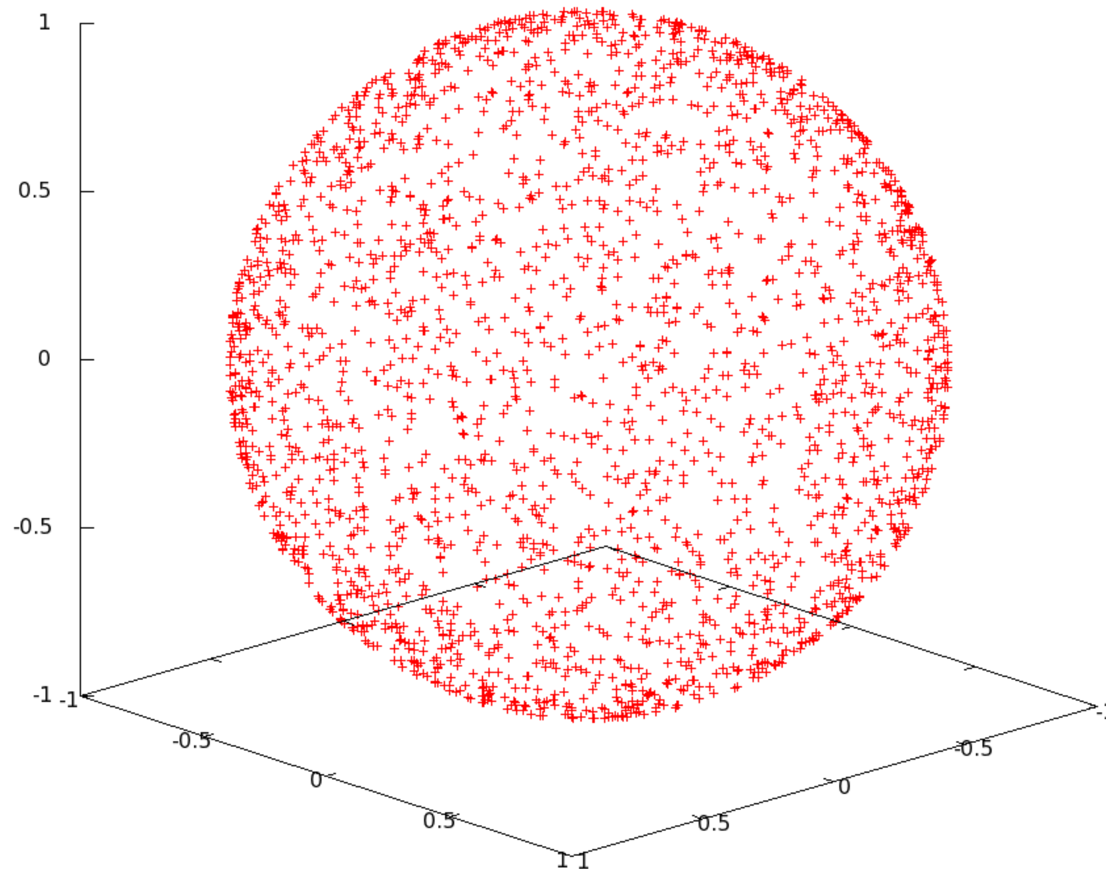
set x to a uniform random variable between -1 and 1
set y to a uniform random variable between -1 and 1
set z to a uniform random variable between -1 and 1
this sample is a point (x, y, z) somewhere in the unit cube
calculate $R_{sample} = \sqrt{x^2 + y^2 + z^2}$
if $R_{sample} > 1$ discard this sample and start over with a different sample
otherwise extend sample onto unit sphere with

$$x \leftarrow \frac{x}{R_{sample}}$$

$$y \leftarrow \frac{y}{R_{sample}}$$

$$z \leftarrow \frac{z}{R_{sample}}$$

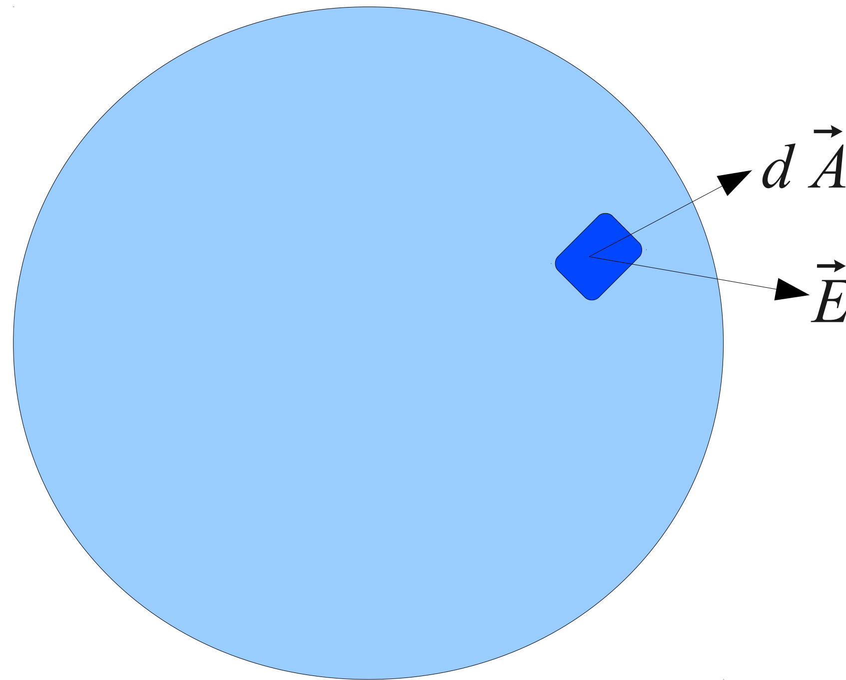
Generating Random Points on a Unit Sphere



Example: Gauss' Law

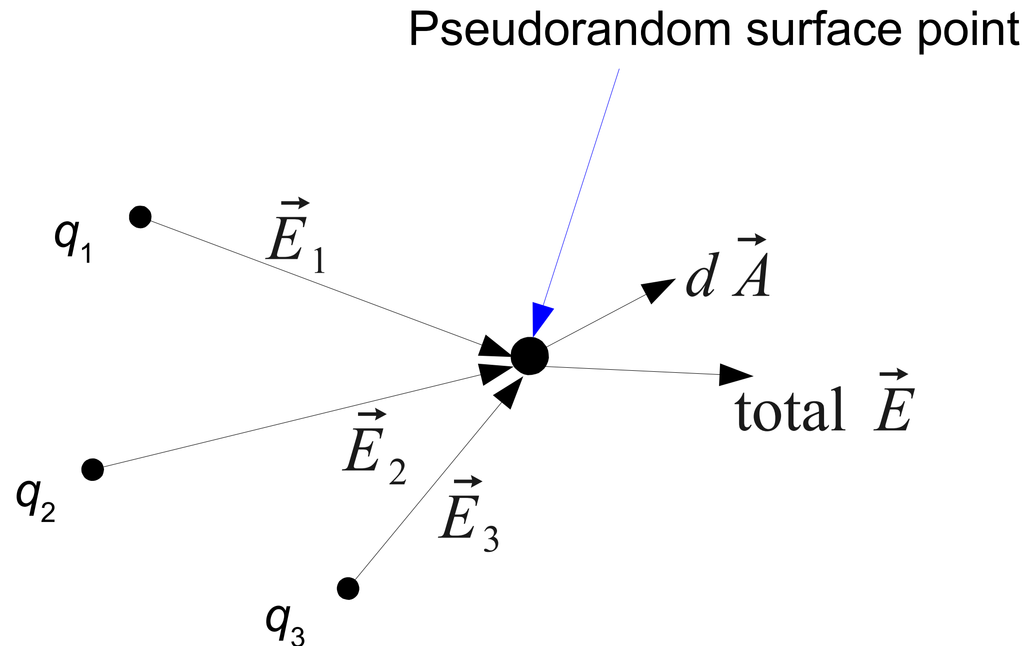
$$q = \epsilon_0 \oint \vec{E} \cdot d\vec{A}$$

q is the charge enclosed in a Gaussian surface,
$\vec{E}$ is the electric field vector at a surface element, and
$d\vec{A}$ is the outward unit vector from the surface element



With a single charge at the center of a sphere, Gauss' Law is equivalent to Coulomb's Law, but with multiple charges off center, Monte Carlo sampling can give a surface integral without the complexity of spherical coordinates

Example: Gauss' Law



Pick n pseudorandom points on unit sphere. At each surface point, sum the electric field vector from m point charges to get a total electric field vector. Form the dot product of the total E field vector with the surface area element vector as the sum of the products of the x,y and z vector components. The surface area associated with each pseudorandom point is $4\pi / n$. Approximate the surface integral as the sum of all the dot products. The total turns out to be equal to the sum of the charges inside the sphere.

Example: Multidimensional Integration

Suppose you have a 12 element integration to perform over 3 dimensions for each element, resulting in a 36 dimensional integral to evaluate, as in:

$$\int dx_1 \int dx_2 \int dx_3 \cdots \int dx_{36} \, f\left(x_{1,} \, x_{2,} \, x_3 \cdots x_{36}\right)$$

Suppose you try Simpson's rule or some other discrete interpolation method. Say for example, use 64 points for each dimension of integration. This would require 64^{36} evaluations of the integrand function. This is $64^{36} = 2^{216} \sim 10^{65}$ evaluations. Suppose on a supercomputer you could evaluate the function in 1 nanosecond = 10^{-9} seconds. Evaluation of the integral would take $10^{65} * 10^{-9} = 10^{56}$ seconds. The age of the universe is about 10^{17} seconds.

A practical, but still quite time consuming, method is sampling the 36-dimensional space with pseudorandom sampling, but is beyond the scope of this course.

Generating Normal Distributions

The Central Limit Theorem from statistics:

Let X_1, X_2, ... be identically distributed random variables with mean μ and standard deviation σ^2.

Form the sum $S_n = X_1 + X_2 + X_3 + ... + X_n$

Then with large *n* the distribution of

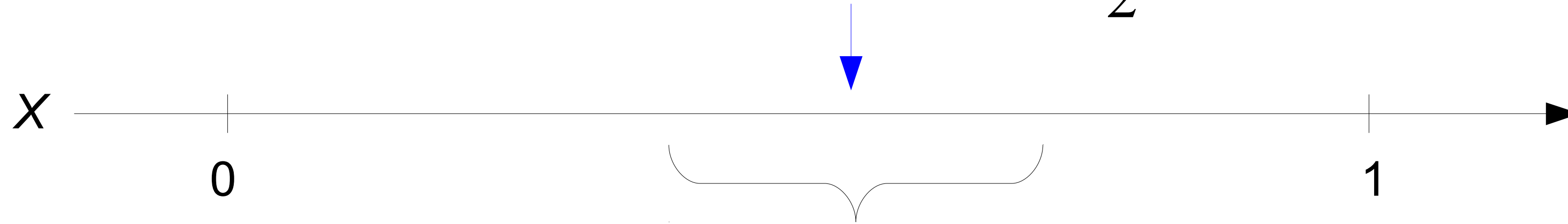
$$\frac{S_n - n\mu}{\sigma\sqrt{n}}$$

approaches a standard normal distribution

Generating Normal Distributions

The class pseudorandom number generator gives floating point numbers uniformly distributed between 0 and 1:

$$\text{Mean } \mu = E[X] = \frac{1}{2}$$



$$\text{Variance } \sigma^2 = E[X^2] - \mu^2 = \frac{1}{3} - \frac{1}{4} = \frac{1}{12}$$

So just summing n numbers returned from the generator as S_n and calculating:

$$Y = \left(S_n - \frac{n}{2}\right) \cdot \sqrt{\frac{12}{n}}$$

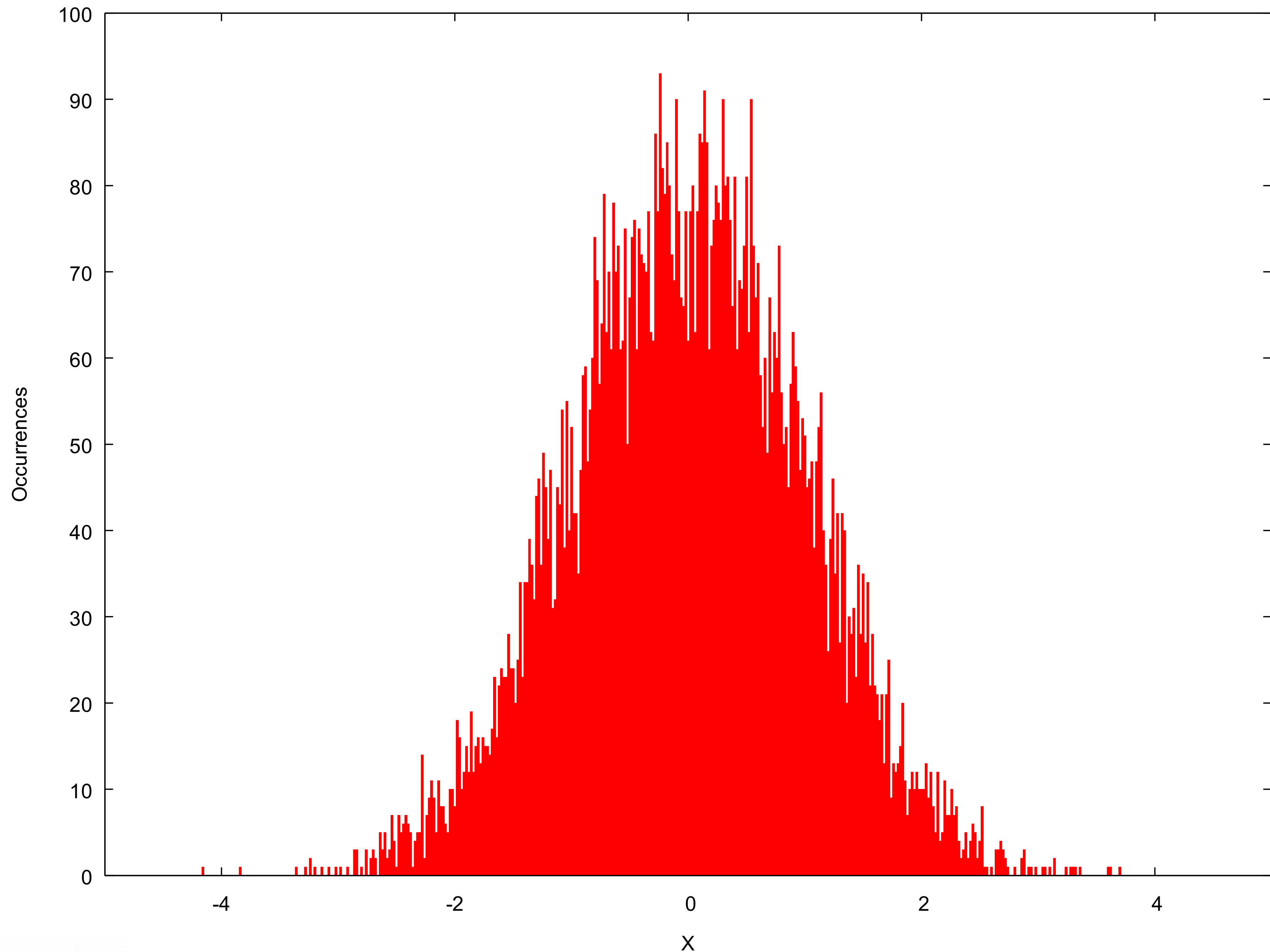
for a modest value of n will give a very good approximation to a normally distributed variable. For simplicity, choose n = 12:

$$Y = S_{12} - 6$$

Additional C Function for Normal Distribution

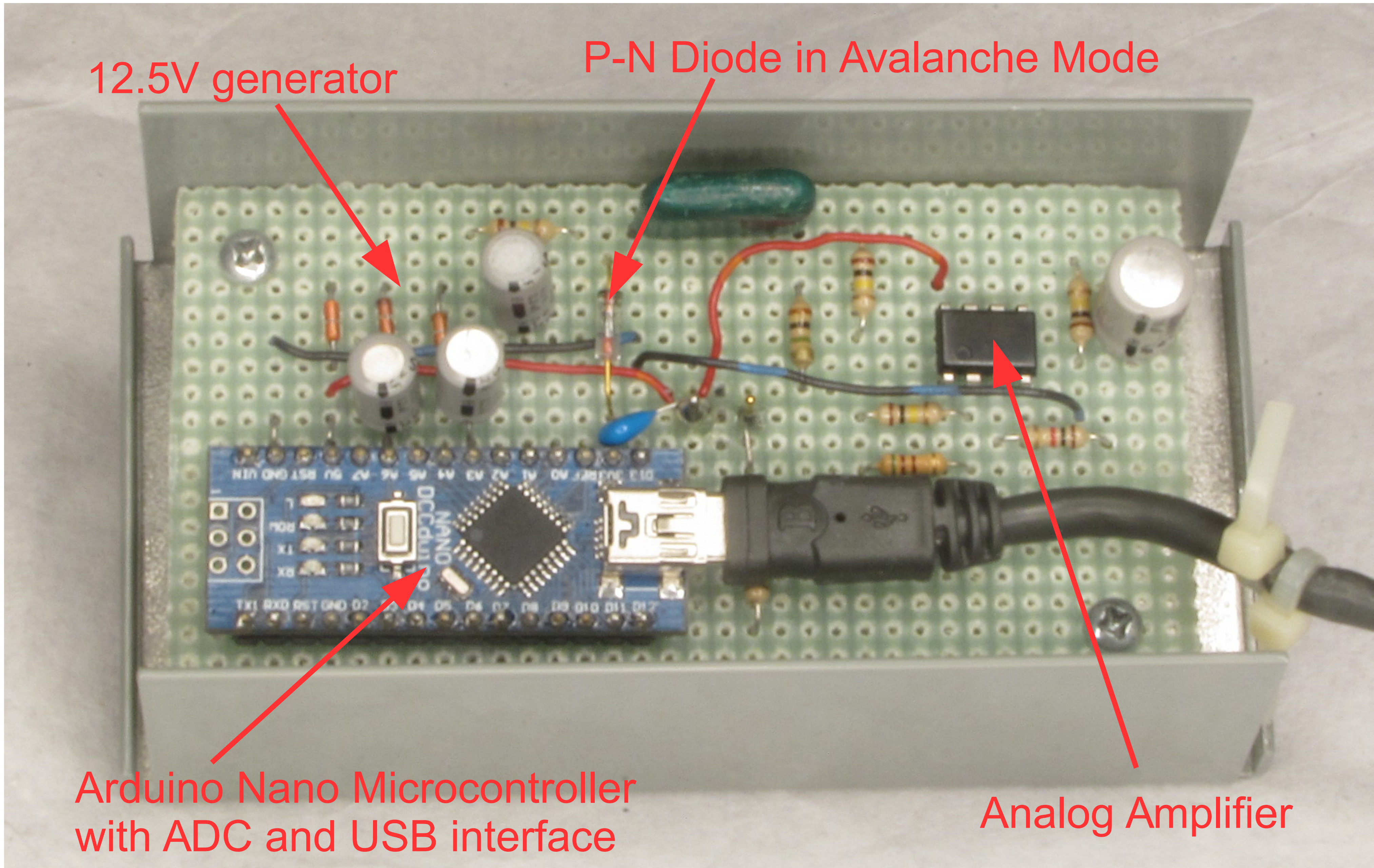
```
double random_normal_gen(double mu,double sigma) {
  int i;
  double sum;

  sum = random_gen();
  for (i = 1; i < 12; i++) sum += random_gen();
  sum -= 6.0;
  return (mu + (sigma * sum));
}
```

Test With 10,000 Samples



Generating True Random Numbers

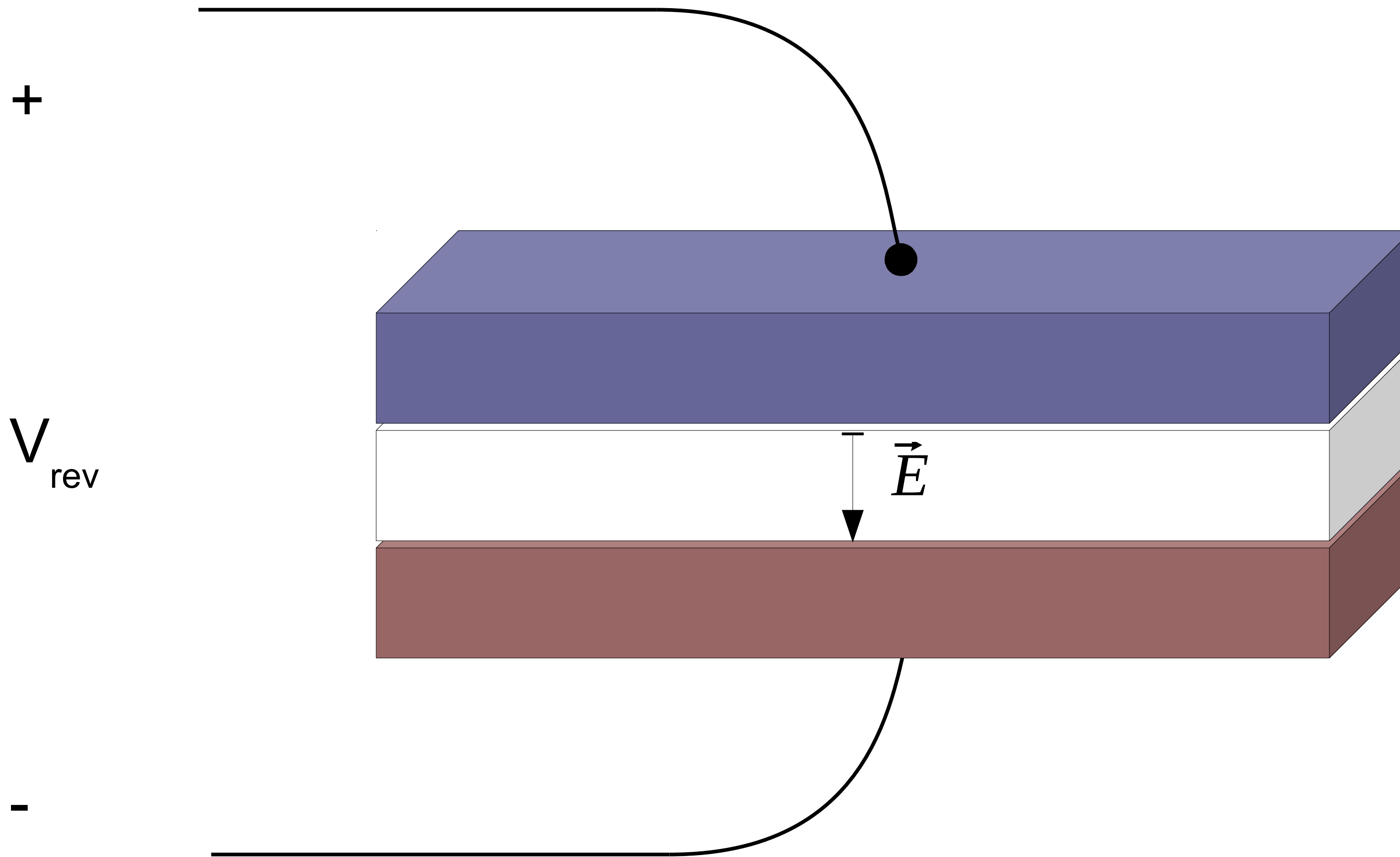
What's Inside?



Snow Avalanche Triggered by Skier

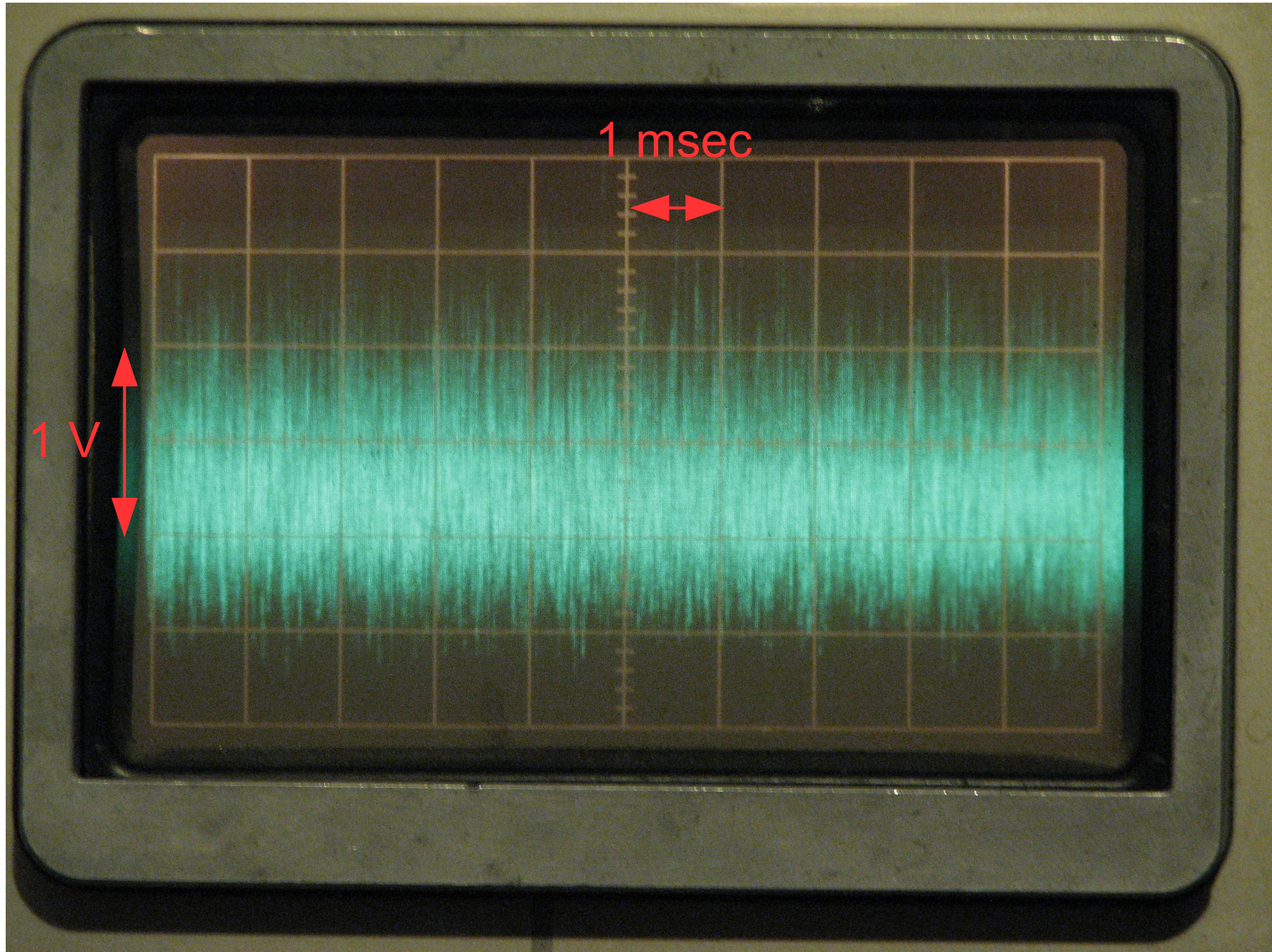
Avalanche Ionization in P-N Diodes

Apply a large reverse-bias voltage, and a large internal electric field will be established. If the field reaches a critical level, impact ionization will occur. The current can reach large levels with ionization avalanching through the junction.

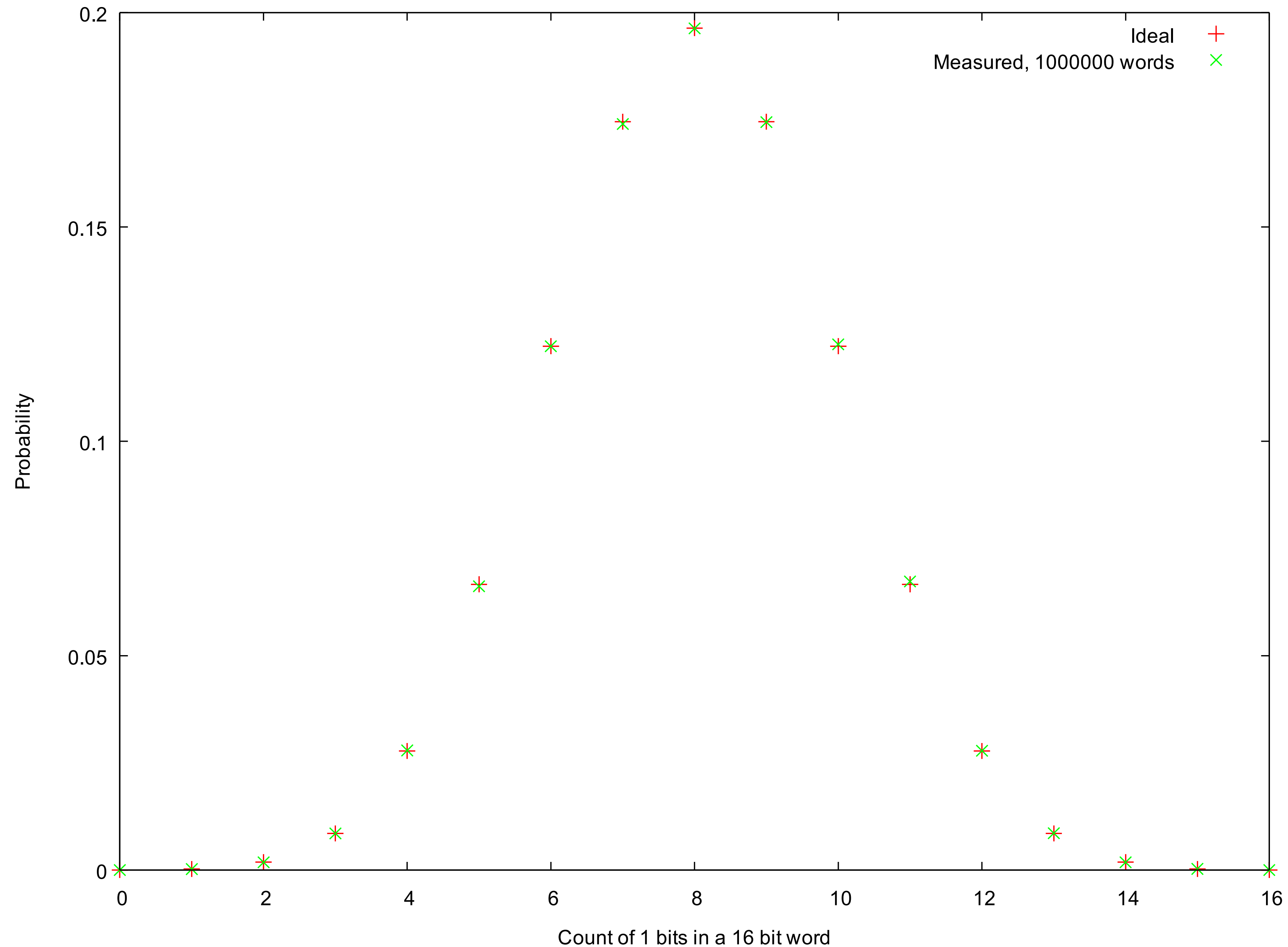


Ionization of Inert Gasses

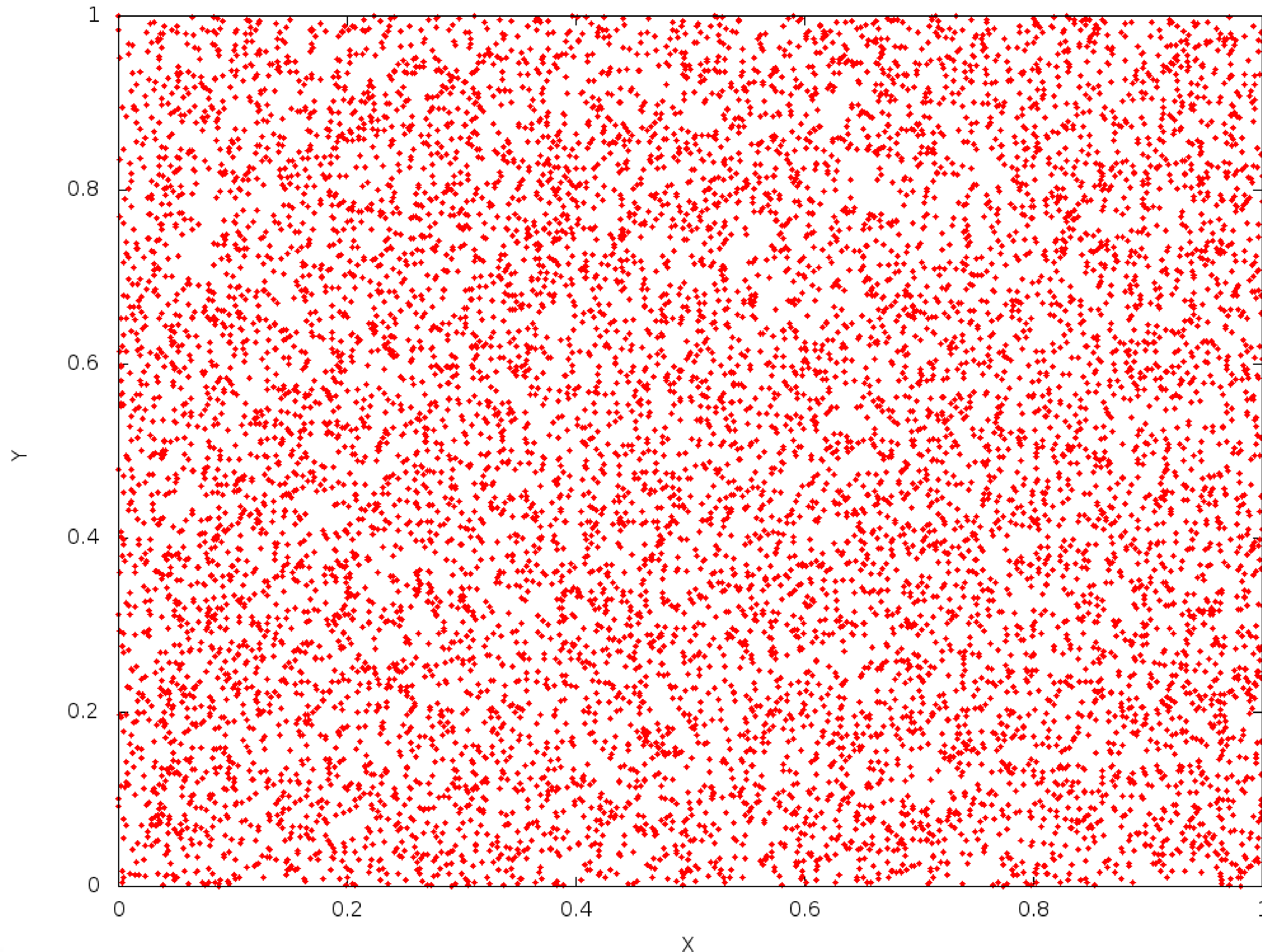
True Random Voltage Waveform



Testing Random Bits vs. Binomial Distribution

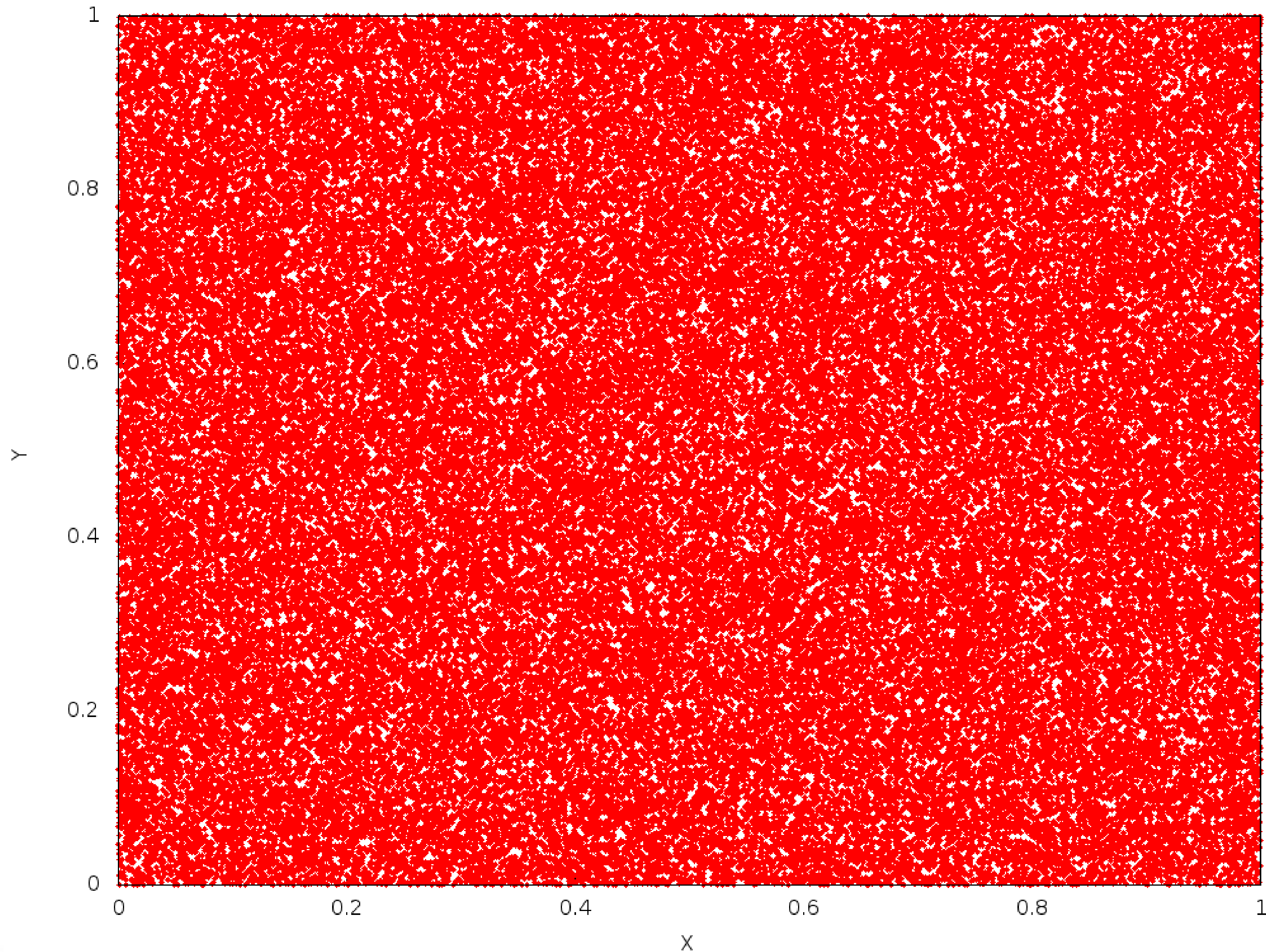


Testing Correlation of Numerical Stream

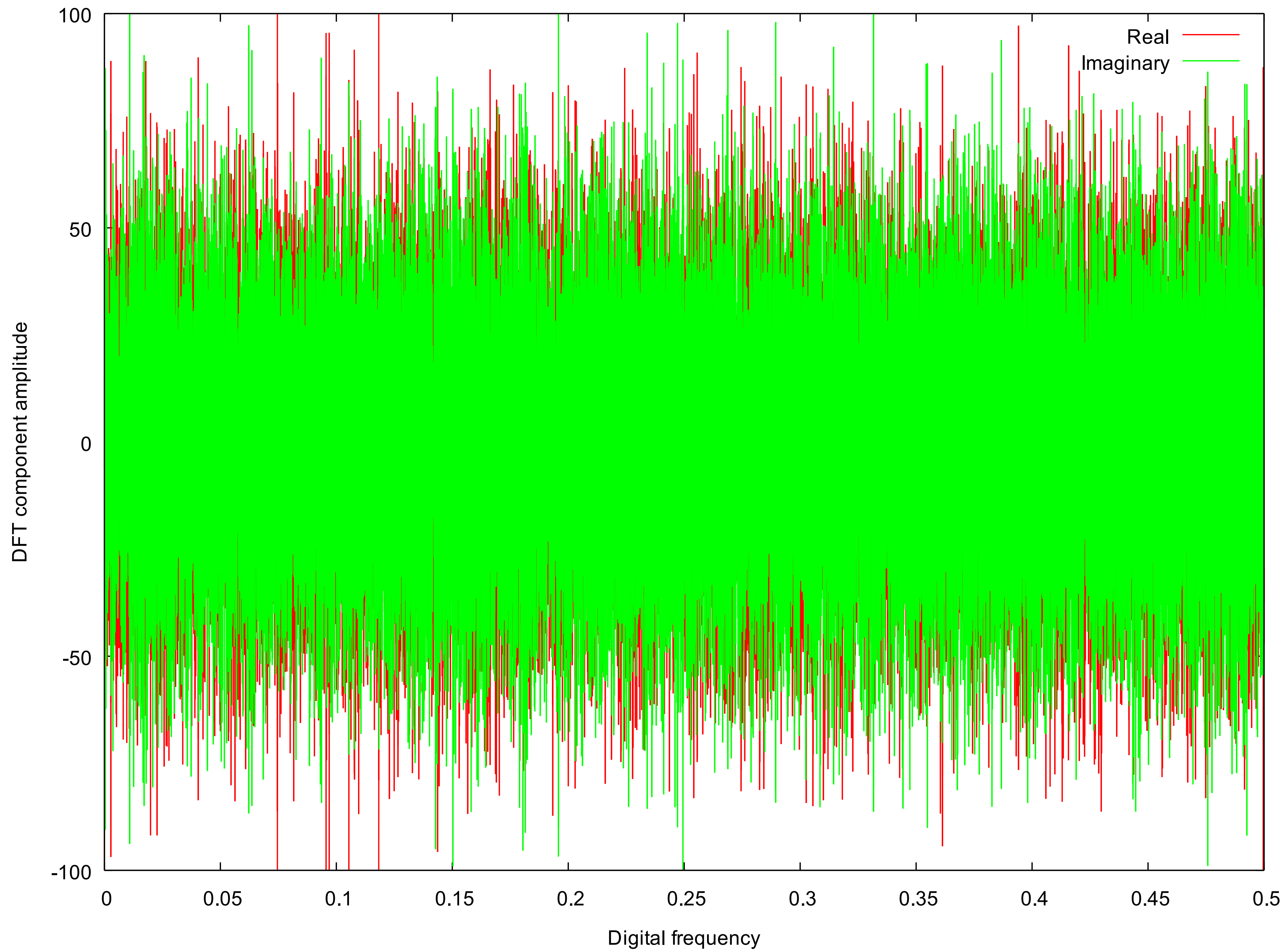


Testing Correlation of Numerical Stream

100,000 points



Fourier Transform of Numerical Stream



Test Results from Linux rngtest Tool

Analyses performed on blocks of 20,000 bits from any random source, after 32 initialization bits...

```
$ Random2_Linux -n2500004 | rngtest

rngtest: starting FIPS tests...
rngtest: entropy source drained
rngtest: bits received from input: 20000032
rngtest: FIPS 140-2 successes: 1000
rngtest: FIPS 140-2 failures: 0
rngtest: FIPS 140-2(2001-10-10) Monobit: 0
rngtest: FIPS 140-2(2001-10-10) Poker: 0
rngtest: FIPS 140-2(2001-10-10) Runs: 0
rngtest: FIPS 140-2(2001-10-10) Long run: 0
rngtest: FIPS 140-2(2001-10-10) Continuous run: 0
rngtest: input channel speed: (min=2.688; avg=4.489;
max=9765625.000)Kibits/s
rngtest: FIPS tests speed: (min=86.305; avg=98.716;
max=143.410)Mibits/s
rngtest: Program run time: 4360659098 microseconds
```