Monte Carlo Methods

Named by Stanislaw Ulam and Nicholas Metropolis in 1940s for the famous gambling casino in Monaco.

Used in numerous applications.

Any method that uses random numbers to solve a problem

Major idea: simulate a system behavior using random sampling.

Typically a perfectly deterministic method but with random input. Examine the statistics of the output. Translating uncertainty in input to the output.

Simulation with Monte Carlo

Simple case: Ball toss at the carnival

Throw 10 balls, 20% chance of hitting target. How many hits?

Can generate many realizations of 10 random numbers (0-1).
Count how many numbers 0.2 or less for each realization.
Average the realizations.

Hurricanes:

Develop probability distributions for central pressure, forward speed for example, then using different probabilities pick parameters and then simulate.

Monte Carlo

1. **Identify the Transfer Equation**
quantitative model of the or process you wish to explore.

2. **Define the Input Parameters**
For each factor in your transfer equation, determine how its data are distributed. Some inputs may follow the normal distribution, while others follow a triangular or uniform distribution.

3. **Create Random Data**
To do valid simulation, you must create a very large, random data set for each input —something on the order of 100,000 instances. These random data points simulate the values that would be seen over a long period for each input.

4. **Simulate and Analyze Process Output**
With the simulated data in place, you can use your transfer equation to calculate simulated outcomes. Running a large enough quantity of simulated input data through your model will give you a reliable indication of what the process will output over time, given the anticipated variation in the inputs.

Pump

$$Q = \pi \frac{D^2}{2} LR$$

where *Q* is flow rate, D is piston diameter, L is stoke of pump, and R is rpm of motor.

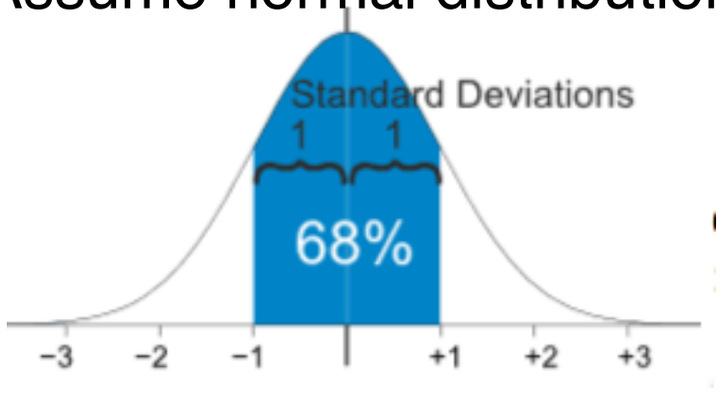
Manufacturer wants 12 ml/min with 0.12 ml/min std deviation:

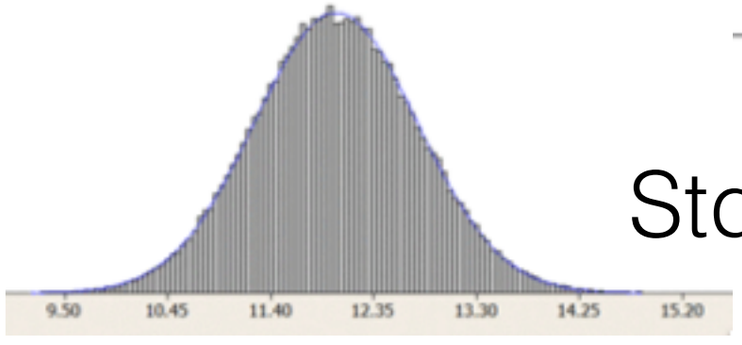
D = 0.8 cm, L=2.5 cm, R = 9.549 rpm.

Data shows D has std dev. of 0.003 cm, L has std dev of 0.15 cm, and R std dev 0.17 rpm

What is the distribution of pump performance? Assume normal distribution.

$$f(x; \mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$





Std dev: 0.757 ml/min

Monte Carlo Calculation of Pi

Inscribe a circle of radius r inside a square with sides $2r$

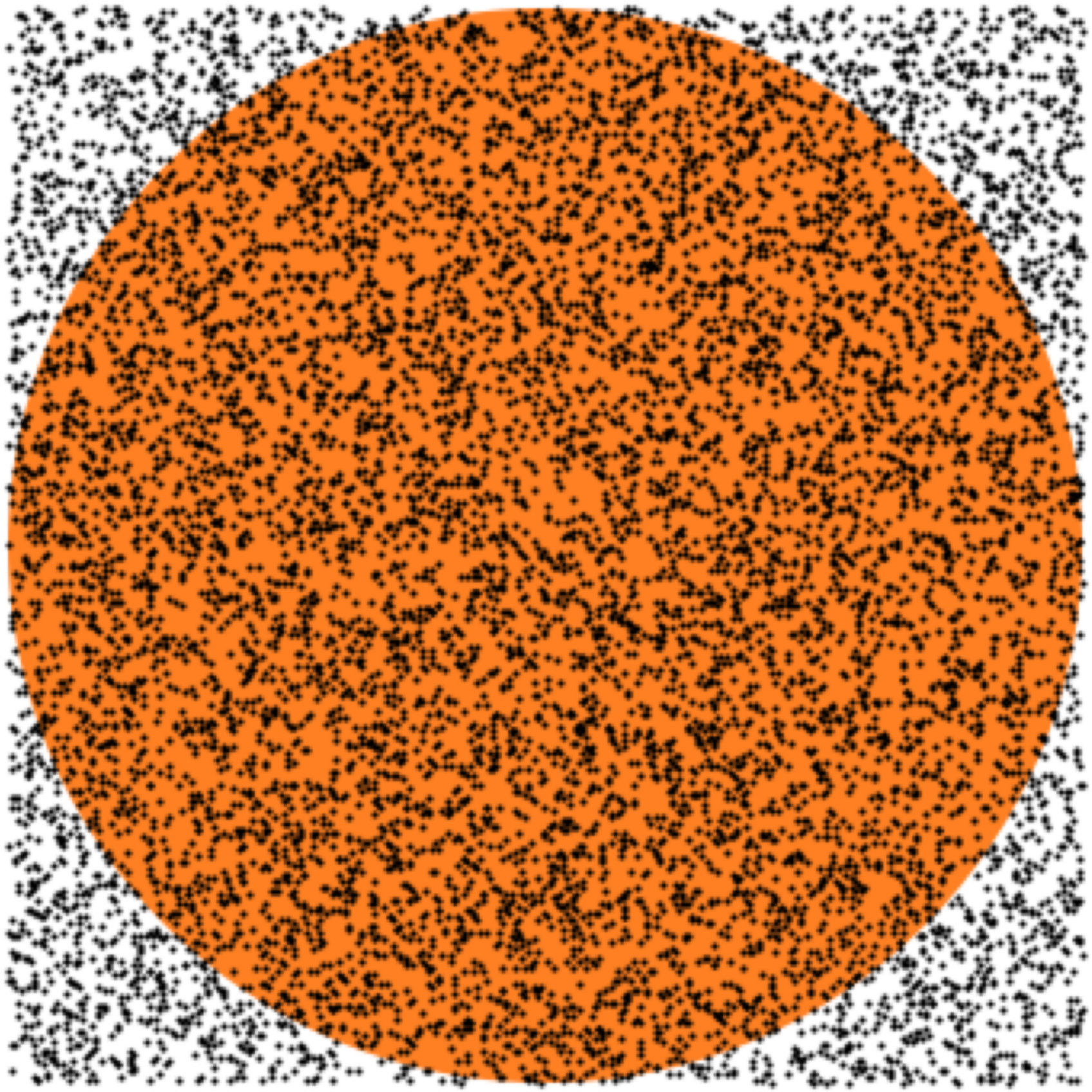
Area of Square: $(2r)^2$

Area of circle: πr^2

Ratio of Areas: Circle/Square $= \pi/4$

Therefore consider N random positions (x_i, y_i) between 0 - r in both x and y

Determine the fraction of points that lie inside the circle; multiply by 4



inside: 7861 outside: 2139 total: 10000

$\pi \approx 4 \times 7861/10000 = 3.1444$

CuRAND Library

Generate ‘random’ numbers on host or device.

Host side: /include/curand.h
Device side: /include/curand_kernel.h

Advantage of the device side is that the random numbers are created on device and no memcpy calls are needed to load global memory.

cuRAND

All random numbers are developed by generators: RNG

To use:

1) Create a new generator of desired type (9 types)
2) Set generator options (such as seed)
3) Allocate memory on device (cudaMalloc)
4) Generate random numbers with curandGenerate() or other function
5) Use results
6) Can generate more numbers with curandGenerate()
7 Clean up with curandDestroyGenerator()

CUDA Toolkit Documentation

Distributions

curand_uniform, curand_normal, curand_log_normal, curand_poisson (single)

curand_uniform_double, curand_normal_double, curand_log_normal_double (double)

100 Random Numbers

```
/*  This program uses the host CURAND API to generate 100 * pseudorandom floats. */
#include <stdio.h>
#include <curand_kernel.h>

#define CURAND_CALL(x) do { if((x)!=CURAND_STATUS_SUCCESS) { \
  printf("Error at %s:%d\n",__FILE__,__LINE__);\
  return EXIT_FAILURE;}} while(0)
```

```
int main(int argc, char *argv[])
{  int n = 100;
   int i;
   curandGenerator_t gen;
   float *devData, *hostData;
   /* Allocate n floats on host */
   hostData = (float *)calloc(n, sizeof(float));
   /* Allocate n floats on device */
   cudaMalloc(&devData, n*sizeof(float));
```

```
/* Create pseudo-random number generator */
  CURAND_CALL(curandCreateGenerator(&gen,
                                    CURAND_RNG_PSEUDO_DEFAULT));
  /* Set seed */
  CURAND_CALL(curandSetPseudoRandomGeneratorSeed(gen, 1234567));

  /* Generate n floats on device */
  CURAND_CALL(curandGenerateUniform(gen, devData, n));

  /* Copy device memory to host */
  cudaMemcpy(hostData, devData, n * sizeof(float),
                                    cudaMemcpyDeviceToHost);

   /* Show result */
    for(i = 0; i < n; i++)
        { printf("%1.4f ", hostData[i]); }
     printf("\n");
     /* Cleanup */
     CURAND_CALL(curandDestroyGenerator(gen));
     cudaFree(devData);
     free(hostData);
     return 0;
}
```

Examine MonteCPi.cu

```
#include <curand_kernel.h>

#define TRIALS_PER_THREAD 4096
#define BLOCKS 256
#define THREADS 256
#define PI 3.1415926535898  // known value of pi

 __global__ void gpu_monte_carlo(float *estimate, curandState *states)
{
        unsigned int tid = threadIdx.x + blockDim.x * blockIdx.x;
        int points_in_circle = 0;
        float x, y;
        curand_init(1234, tid, 0, &states[tid]); // Initialize CURAND
        for (int i = 0; i < TRIALS_PER_THREAD; i++) {
            x = curand_uniform (&states[tid]);
            y = curand_uniform (&states[tid]);
            points_in_circle += (x*x + y*y <= 1.0f); // count if x & y is in the circle.
            }
        estimate[tid] = 4.0f * points_in_circle / (float) TRIALS_PER_THREAD;
}
```

Main

```
int main ()
{
	float *dev;
	curandState *devStates;
	printf("# of trials per thread = %d, # of blocks = %d, # of threads/block = %d.\n",
					TRIALS_PER_THREAD, BLOCKS, THREADS);
	 printf("---> total number of points = %d\n", TRIALS_PER_THREAD*BLOCKS*THREADS);

	cudaMalloc((void **) &dev, BLOCKS * THREADS * sizeof(float));
	cudaMalloc( (void **)&devStates, THREADS * BLOCKS * sizeof(curandState) );

	gpu_monte_carlo<<<BLOCKS, THREADS>>>(dev, devStates);
	cudaMemcpy(host, dev, BLOCKS * THREADS * sizeof(float),
						cudaMemcpyDeviceToHost);

	float pi_gpu;
	for(int i = 0; i < BLOCKS * THREADS; i++)
		{
			pi_gpu += host[i];}			//average the values from the threads
	pi_gpu /= (BLOCKS * THREADS);
```

Try MonteCPi

Use different probability distributions in MonteCPi.cu

Crude Monte Carlo Integration

Numerical integration:

$$\int_a^b f(x)dx$$

Newton-Cotes formulas: evaluate integral at fixed points

Rectangular rule:

$$\int_a^b f(x)dx = \frac{(b-a)}{N}\sum_{n=0}^{N-1} f(x_n) \text{ where } x_n = a + n(b-a)/N$$

Crude Monte Carlo Integration

$$\int_a^b f(x)dx = \frac{(b-a)}{N}\sum_{n=0}^{N-1} f(x_n) \text{ where } x_n \text{ are random numbers}$$

Converges with factor of 1/Sqrt(N), which is slow.

For two dimensions:

$$\int_a^b \left(\int_c^d f(x,y)dy \right) dx = \frac{(b-a)}{N_x}\frac{(d-c)}{N_y}\sum_{n=0}^{N_x-1}\sum_{m=0}^{N_y-1} f(x_n, y_m)$$

$$\text{where } x_n, y_m \text{ are random numbers}$$

Monte Carlo integration for 5 dimensions or higher usually faster than Newton-Cotes formulas. Consider 10 dimensions: 20 pts per dimension. Conventional methods: 20^{10} points

Monte Carlo

Law of Large Numbers

Chose *N* numbers x_i randomly with a uniform probability from a to b:

$$\frac{1}{N}\sum_{i=0}^{N-1} f(x_i) \rightarrow \frac{1}{(b-a)}\int_a^b f(x)dx$$

as *N* becomes large. Conditions on *f(x)*: integrable, finite, and piecewise continuous.

Acceptance-Rejection Monte Carlo

Same integral. $$\int_a^b f(x)dx$$

Find max of f(x) in interval a to b: f_{max}; Find min: f_{min}
Develop a box of length (b-a) and height f_{max}-f_{min}.

Now generate N random pairs (x_i,y_i) with $a < x_i < b$ and $f_{min} < y_i < f_{max}$

If yi > f(xi), then this pair falls outside of integrand area.

Finally, sum all the pairs inside integrand area, divide by N, and multiply by area of rectangle, (f_{max}-f_{min}) * (b - a) to get the solution.

Works but not as accurate as crude Monte Carlo.

Class Problem

Write a program to generate a sequence of 1024 random numbers that are Gaussian (normally) distributed.