Monte Carlo Algorithms

Dmytro Kedyk

Some Applications

- Concurrent skip list
- Option pricing with binomial tree
- World cup winner prediction
- Quantifying and comparing system performance
- Fun interview questions

Learning Outcomes

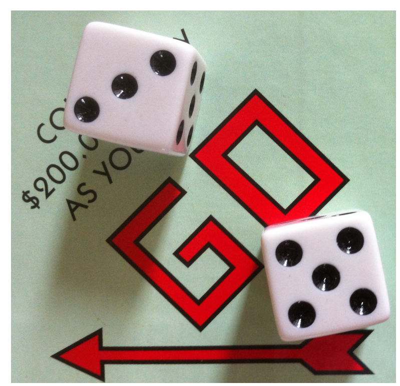
- Understand advantages of randomization
- Be able to generate random numbers and other objects
- Add simulation to your generic solution method toolbox
- Easy perform traditionally complex statistical analysis
- Correctly evaluate and compare system performance

Topics

- Quick review of probability
- Theory of randomization
- Generation of random objects
- Monte Carlo method
- Bootstrap method

What is Probability

- Measures likelihood of **events** E_i
- E_i are subsets of a sample space S
  - E.g. S can be real numbers and E_i intervals
  - Samples $x \in S$ aren't events!
- Defined by axioms:



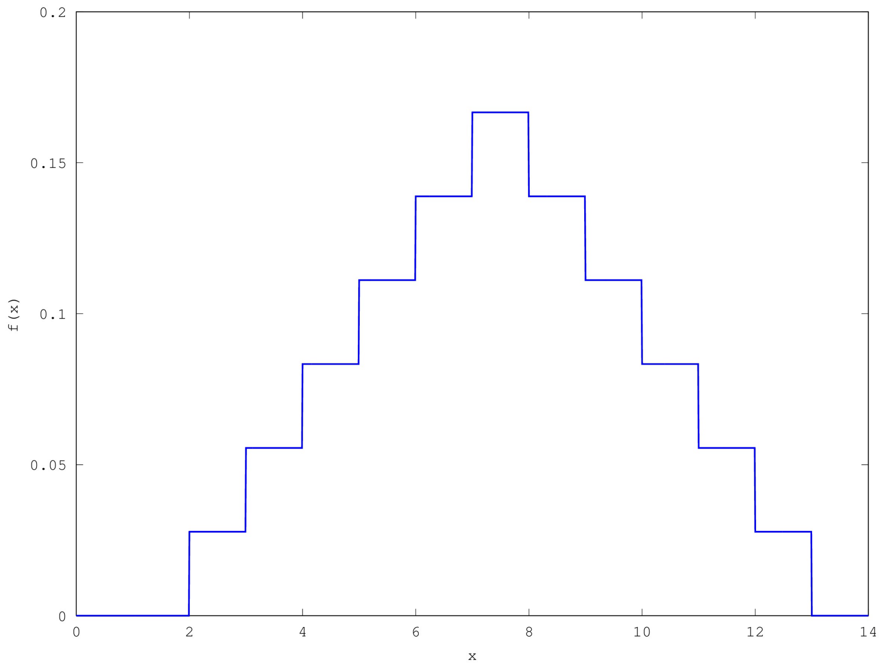
  - $\text{Prob}(E_i) \geq 0$
  - $\text{Prob}(S) = 1$
  - For disjoint events $\text{Prob}(\cup E_i) = \sum E_i$

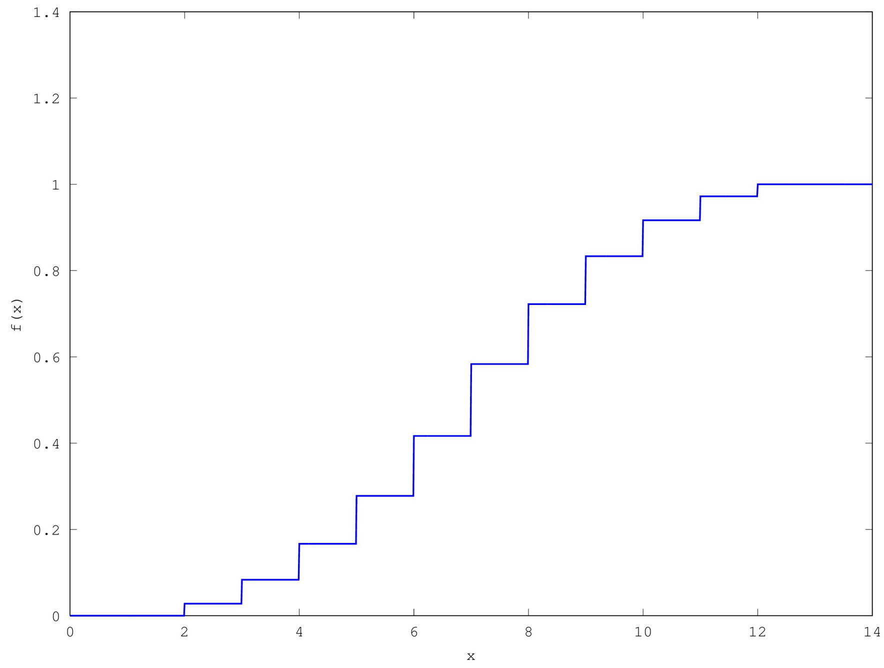
What is a Distribution

- Uses **probability density function** (pdf) $f(x)$ to assign values to $x \in S$
  - $\forall E$ Prob(E) = $\int_{x \in E} f(x)$
  - Values $f(x)$ need not be probabilities!
- For connected range events **cumulative distribution function** (cdf) $F(x) = \int_{\infty < t \leq x} f(t)$ directly gives probabilities

Die Sum Distribution

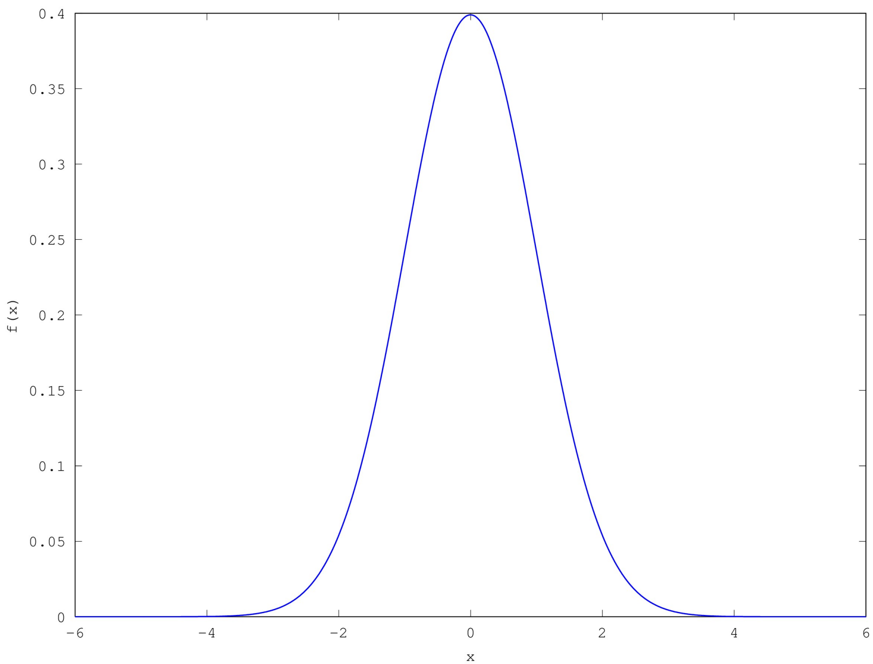
- $f(x) = \max(0, (6 - |7 - x|)/36)$
- $F(8) - F(5)$ = Prob(sum is 6, 7, or 8)

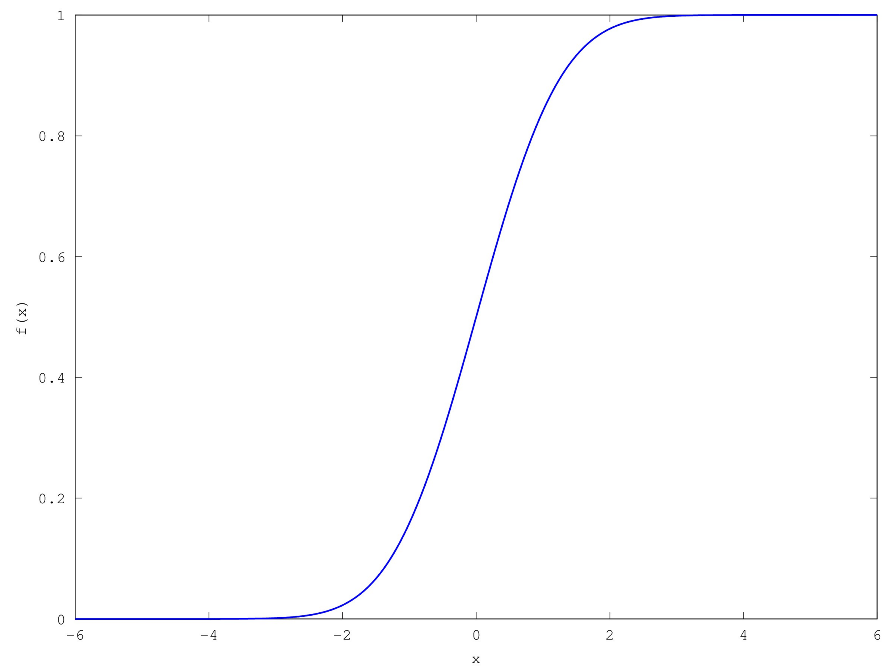




Normal(m, q) Distribution

- m is mean, q standard deviation
- Models many events, heavily used in statistics
- $f(x) = \exp(-((x - m)/q)^2/2)/(q\sqrt{2\pi})$



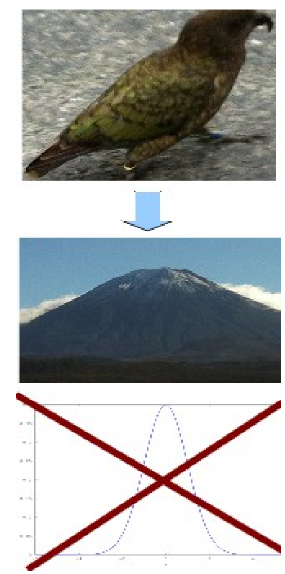


Evaluating Normal CDF

- E.g. two-sided confidence interval is $2F(x) - 1$
- $F(x) = 0.5 + erf(x/\sqrt{2})/2$
- Can't express *erf* in terms of elementary functions, must approximate:
  - $erf(x) \approx 1 - (1 + \sum_{0 \leq i < 6} a_i x^i)^{-16}$
  - a_i are *0.0705230784, 0.0422820123, 0.0092705272, 0.0001520143, 0.0002765672, 0.0000430638*
  - approximation has $\leq 10^{-7}$ error

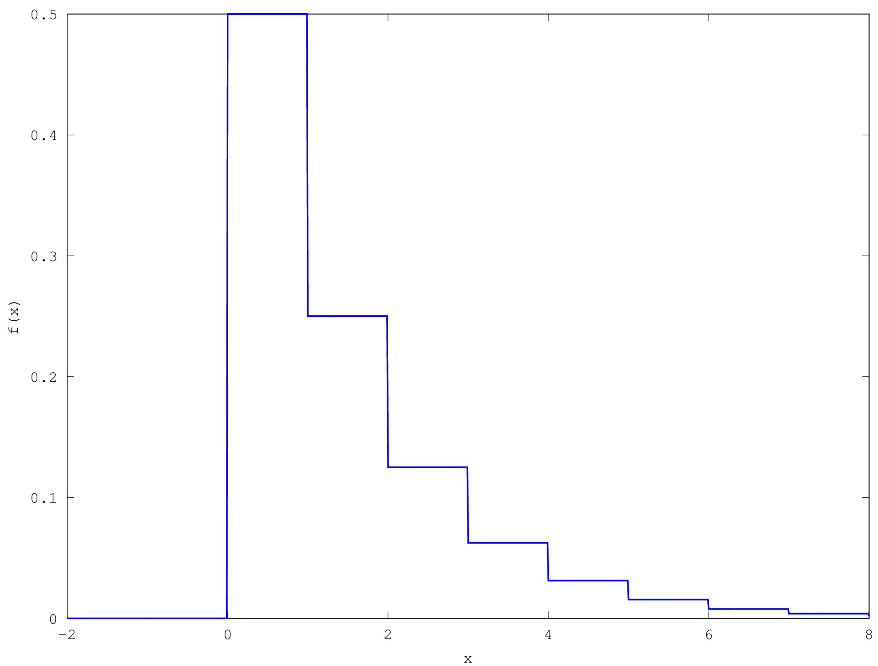
Using Distributions

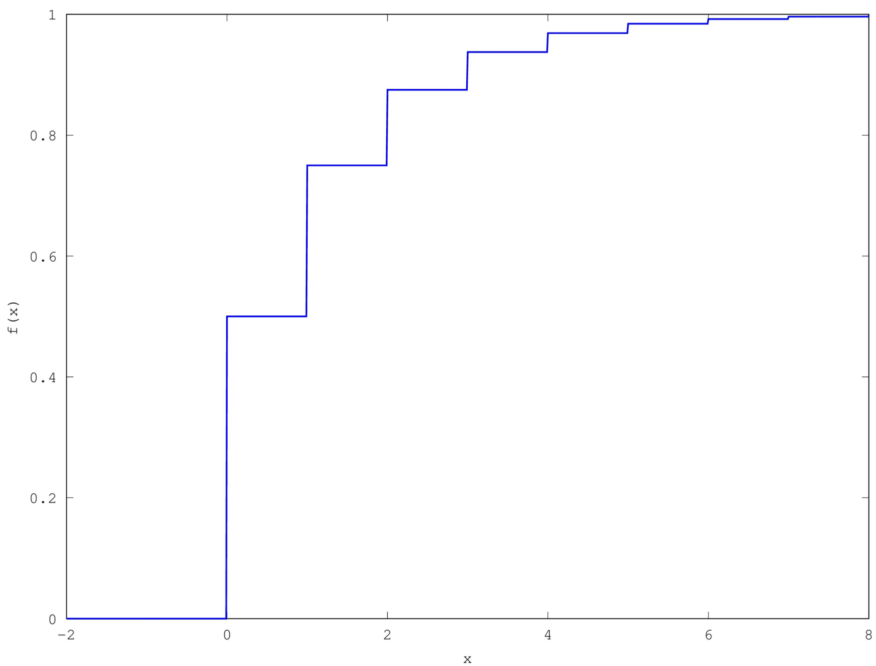
- Often don't know the right distribution!
  - Every distribution is a good or bad model for some events
  - E.g. normal is a bad model for flying bird's avoiding a pole
- In many cases need only a summary
  - **Expected value** is $M = \int_{x \in S} x f(x)$
  - **Variance** is $E[(x - M)^2]$
- Summaries may lose information
  - E.g. for multi-peak data M doesn't mean much



Geometric(p) Distribution

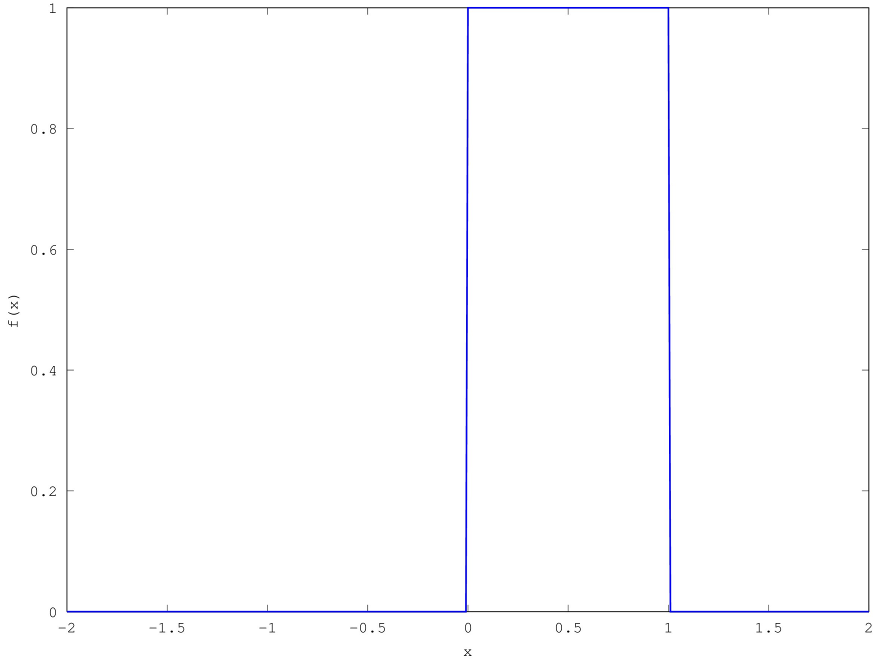
- How long it takes for p-coin to give “tails“
- x discrete, $f(x) = (1 - p)^{x-1}p$
- Models waiting for specific event, $E[x] = 1/p$

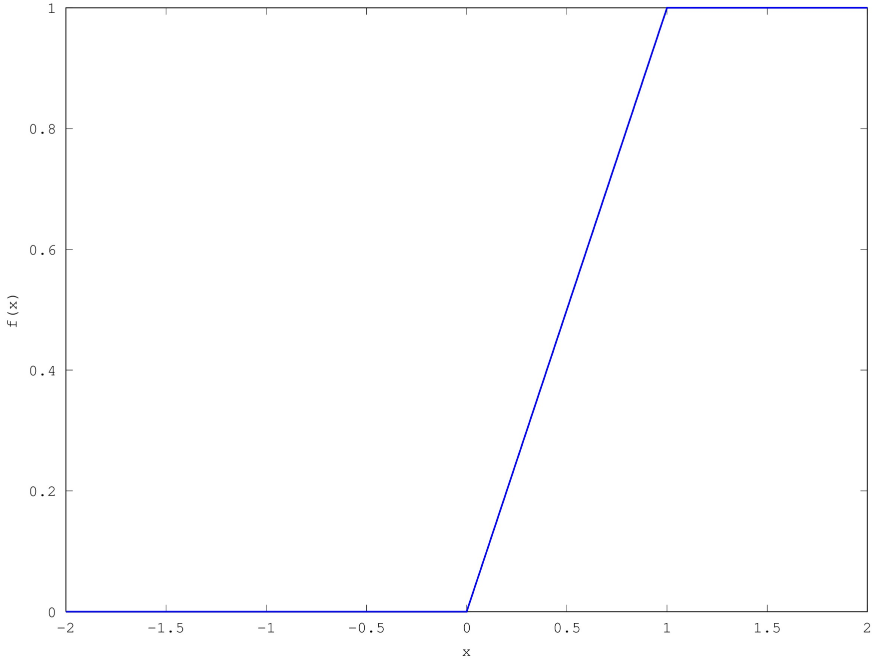




Uniform(a, b) Distribution

- Easy to sample from, basic building block
- $f(x) = 1/(b - a)$ if $a \leq x \leq b$ and 0 otherwise
- Used for no information models





Topics

- Quick review of probability
- Theory of randomization
- Generation of random objects
- Monte Carlo method
- Bootstrap method

Randomization Helps

- In every game there is an optimal **randomized** strategy
- But may not be optimal **deterministic** one
- E.g. rock-paper-scissors – random choices draw on average
- But can learn to defeat a clever deterministic strategy, e.g. using machine learning

Randomized Algorithms

- Game between algorithm and input
  - Using **rand()** may avoid bad worst cases on average
- **Randomness in data** ≠ rand()
- But may help too
  - Worst case input may be unlikely
  - Thus many algorithms are fast in practice and slow in theory
  - E.g. unbalanced binary tree

Quicksort

- Pick a pivot, split array into < pivot and ≥ pivot, and recurse on each half

M	I	S	S	I	S	S	I	P	P	I

⇩

M	I	I	I	I	S	S	S	S	P	P

- How to pick? – first, last, median of 3 may give $O(n^2)$ runtime
- Random!
  - $O(n\ln(n))$ runtime on average
  - Also **tail inequality** – Prob(runtime > $O(n\ln(n))$) is exponentially small!

Types of Randomized Algorithms

- **Las Vegas** – expected performance
  - E.g. random pivot quicksort – E[runtime] = $O(n\ln(n))$
- **Monte Carlo** – expected correctness
  - E.g. Miller-Rabin primality test – tiny chance of error
  - Repeat to reduce error
- Often randomized algorithms are the fastest known

Topics

- Quick review of probability
- Theory of randomization
- Generation of random objects
- Monte Carlo method
- Bootstrap method

What is a Random Number Sequence?

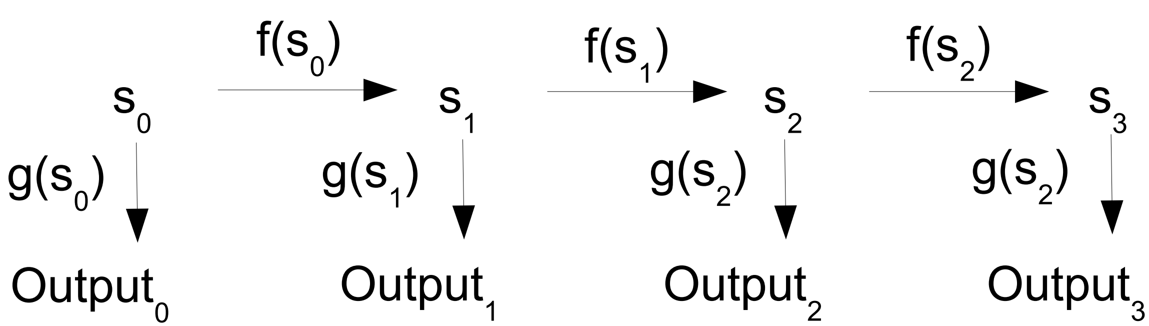
- Hard to define exactly, but:
- A sequence is random if can't predict next value with probability > that of a guess
- Is a sequence containing 10^9 consecutive 0's random?

Getting Random Numbers

- Hire a coin flipper
- Measure physical phenomena, e.g. atmospheric noise or radioactive decay
- Record CPU or hard drive activity, mouse movement, keyboard actions, etc
- Slow and unportable!

Pseudorandom Numbers

- Generated deterministically and indistinguishable from random
- General algorithm:
  - Start from some initial state
  - Output and the next state are functions of the current state



Generator Quality

- Produce random-looking outputs
  - Period ≥ 2^{64} – after how long the sequence repeats itself
  - High equidistribution – largest *k* for which consecutive *k* values can be any values
  - Pass statistical tests – can't reject hypothesis that the sequence is random

Generator Requirements

- Return random *double* $u \in (0, 1)$, not $[0, 1]$, to make $\log(u)$ well-defined
- Simple, fast, and portable
- Optionally generate independent streams for parallelization

Linear Congruential Generator

- Single word state s
- Transition $s = (as + c) \% m$
  - a, c, m are picked constants
- $u = (s + 1)/(m + 1)$
- E.g. $a = 2685821657736338717$, $m = 2^{64}$, $c = 0$, $s_0 = 123456789$
  - $s_1 = 8624929095735532502$
  - $u_1 = 0.46755834315649508$

Linear Congruential Generator

- Don't use as is!
  - Period of the lower k bits = 2^k
  - Fails many tests
  - Multiplication needs double precision to avoid overflow

Picking the Initial State

- Function of system time and a password – fast, simple, portable, and very random
- Operating system random source – for cryptography
- Restored generator state from the last run – complete independence of generator runs

Xorshift

- Much better generator, transition
  - Interprets state as Boolean vector
  - And multiplies it by a Boolean matrix
- The matrix has special form – can implement using shift and xor

Xorshift Code

```
class Xorshift
{
    unsigned int state;
    enum{PASSWORD = 19870804};
public:
    Xorshift(unsigned int seed = time(0) ^ PASSWORD)
    {
        assert(numeric_limits<unsigned int>::digits == 32);
        state = seed ? seed : PASSWORD;
    }
    static unsigned int transform(unsigned int x)
    {
        x ^= x << 13;
        x ^= x >> 17;
        x ^= x << 5;
        return x;
    }
    unsigned int next(){return state = transform(state);}
    double uniform01(){return 2.32830643653869629E-10 * next();}
};
```

Xorshift Properties

- 13, 17, and 5 are picked by theory, exhaustive search, and testing
- Minor quality problems
  - The period is $2^{32} - 1$ (state is never 0) – too small
  - Matrix multiplication is a linear operation – bits of successive numbers are correlated
- To remove correlation and increase period
  - Use 64-bit state, this changes constants
  - Combine with a simple LCG

Improved Xorshift

```
class QualityXorshift64
{
    unsigned long long state;
    enum{PASSWORD = 19870804};
public:
    QualityXorshift64(unsigned long long seed =
        time(0) ^ PASSWORD)
    {
        assert(numeric_limits<unsigned long long>::digits == 64);
        state = seed ? seed : PASSWORD;
    }
    static unsigned long long transform(unsigned long long x)
    {
        x ^= x << 21;
        x ^= x >> 35;
        x ^= x << 4;
        return x * 2685821657736338717ull;
    }
    unsigned long long next(){return state = transform(state);}
    double uniform01(){return 5.42101086242752217E-20 * next();}
};
```

Improved Xorshift Propeties

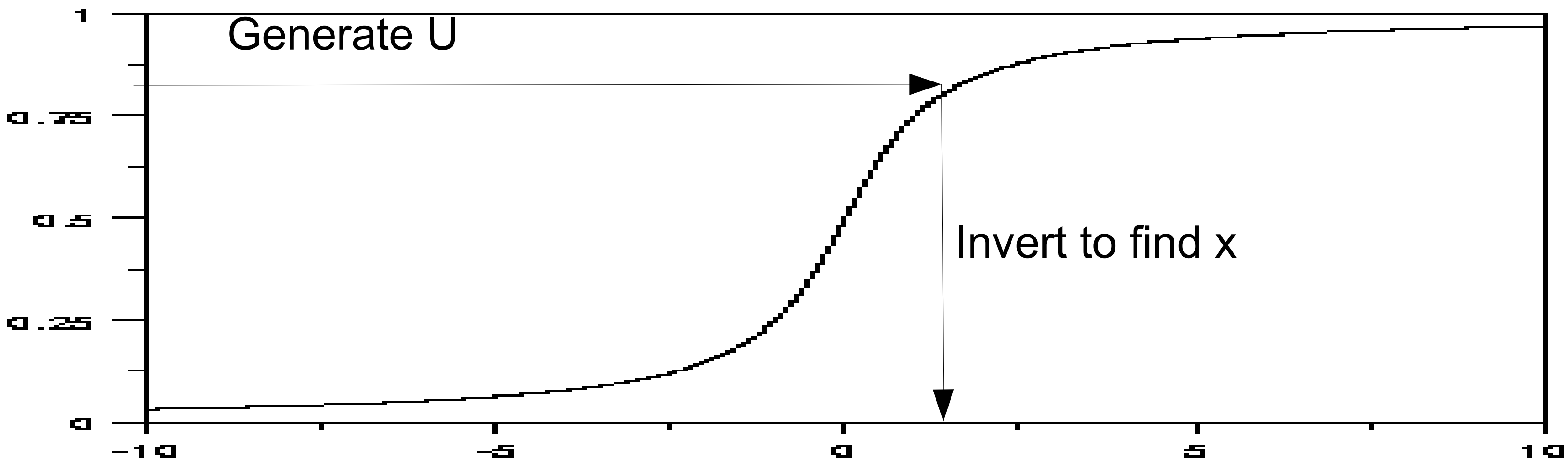
- Doesn't generate 0, so $u \in (0, 1)$
- Very fast, passes most tests
- Period 2^{64} is long enough for all practical uses
- Use as default generator unless have another good one from some API
- Can use transition function for hashing

Other Good Generators

- MRG32k3a
  - Slower, but passes all tests and supports independent streams
- RC4
  - Much slower, but cryptographically secure
- Mersenne Twister
  - Same speed and test performance, and huge state
  - But complex implementation and uses 4KB memory

Samples from Probability Distributions

- Inverse method – cumulative distribution function F is a function $x \rightarrow [0,1]$, so $F^{-1}(u)$ is a random variate
- Can calculate F^{-1} numerically



Inverse Method Example

- Exponential distribution with parameter l: $F(x) = 1 - e^{-lx}$
  - Solve $F(x) = u$ to get $x = -\ln(1 - u)/l$
  - Simplifies to $-\ln(u)/l$ since u and $1 - u$ have the same distribution
- Many other methods for generation, most are distribution-specific
  - Boost has good API

Some Continuous Generators

- Uniform*(a, b)* with *a* < *b*:

```
double uniform(double a, double b)
{return a + (b - a)* uniform01();}
```

- Exponential – memoryless waiting times:

```
double exponential01(){return
-log(uniform01());}
```

- Many more complex ones
  - Normal, gamma, Cauchy, etc.

Some Discrete Generators

- Bernoulli(p) – 1 with prob p and 0 with $1 - p$:

```
bool bernoulli(double p){return
uniform01() <= p;}
```

- Geometric(p) – number of times Bernoulli(p) = 0 before it's 1:

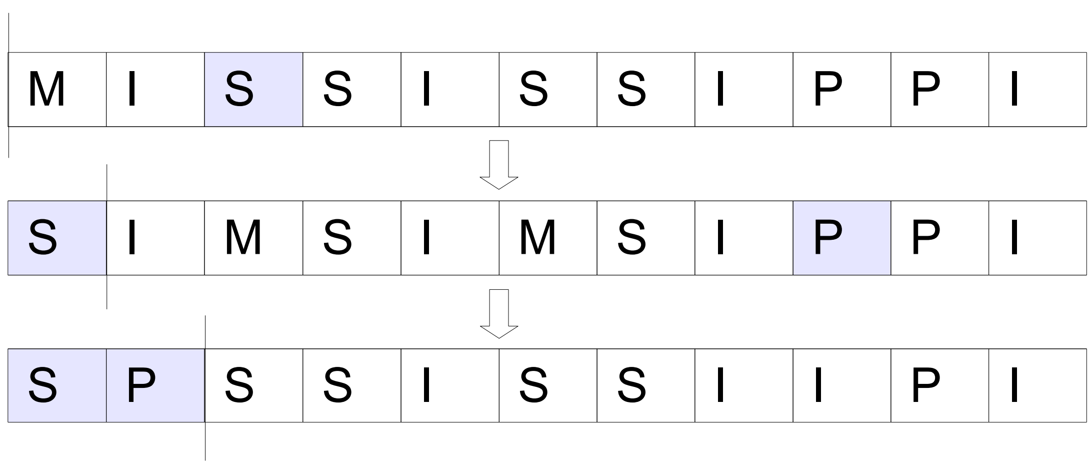
```
int geometric(double p)
{
    int result = 0;
    while(!bernoulli(p)) ++result;
    return result;
}
```

Example Output

```
GlobalRNG.uniform01() 0.904229
GlobalRNG.uniform(10, 20) 10.2389
GlobalRNG.normal01() 0.508248
GlobalRNG.normal(10, 20) -8.47315
GlobalRNG.exponential01() 1.42522
GlobalRNG.gamma1(0.5) 0.00646845
GlobalRNG.gamma1(1.5) 0.355508
GlobalRNG.weibull1(20) 0.894177
GlobalRNG.erlang(10, 2) 13.6015
GlobalRNG.chiSquared(10) 10.7126
GlobalRNG.t(10) 0.492003
GlobalRNG.logNormal(10, 20) 5.51916e+010
GlobalRNG.beta(0.5, 0.5) 0.205863
GlobalRNG.F(10 ,20) 2.49824
GlobalRNG.cauchy01() 0.585426
GlobalRNG.binomial(0.7, 20) 14
GlobalRNG.geometric(0.7) 0
GlobalRNG.poisson(0.7) 2
```

Generating Random Objects

- Permutations – swap the first element with a random one and recursively permute the remaining $n - 1$
- Combinations of k out of n – use above for k steps



Topics

- Quick review of probability
- Theory of randomization
- Generation of random objects
- Monte Carlo method
- Bootstrap method

Law of Large Numbers

- Average of many enough samples is the true average
  - Given n iid samples X_i from distribution T such that $E[X_i] = M$, $(\sum X_i)/n = \overline{x} \rightarrow M$ for $n \rightarrow \infty$
- $\overline{x} \rightarrow$ sample mean of T
- $s = (\sum(X_i - \overline{x})^2)/(n - 1) \rightarrow$ sample variance of T
  - Divisor for s is $n - 1$
  - Because variance $= E[X_j - (\sum_{i \neq j} X_i)/n]$

Central Limit Theorem

- Essentially LLN with error bounds
  - Given n iid samples X_i from a distribution with mean M and finite variance V, for $n \to \infty$, $\overline{x} \sim \text{normal}(M, V/n)$
  - LLN and technical Slutsky's theorem allow using s instead of V
- True mean M is $\overline{x} \pm 2\sqrt{s/n}$ with 95% probability!
  - The multiplier is 1.96 actually, but use 2 for convenience, or others

Monte Carlo Idea

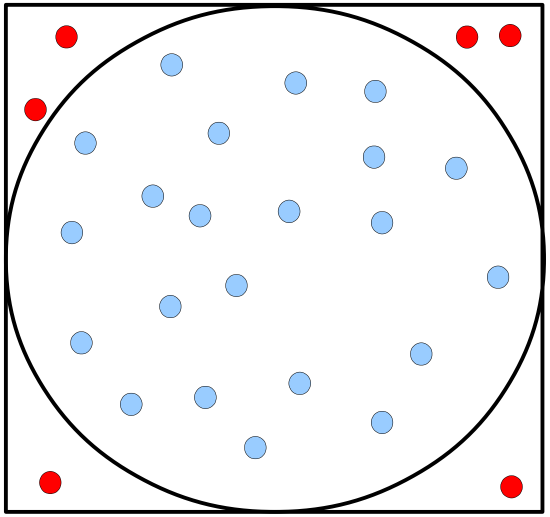
- Define quantity of interest *X* and compute it using CLT
  - Need function *f* producing iid events with value X_i such that $E[X_i] = X$
- Important events must be generated often enough
  - Otherwise need very large *n* for CLT to kick in
  - Rare event problem!

Monte Carlo Algorithm

- Define X, pick f and confidence level
- Until out of patience or (n is large enough and error small enough)
  - $X_i \leftarrow f()$
  - Incrementally update m and s with X_i
- Return $X \leftarrow \bar{x} \pm$ error

Computing π

- Area of a circle with radius r is πr^2 and of its enclosing square $4r^2$
- $X = \pi/4 =$ (area of the circle)/(area of the square)



$\pi \approx 4 \times 22/(22 + 6) \approx 3.142$

Computing π

- f generates random points $p \in (-1, 1) \times (-1, 1)$ and returns $X_i = (\text{distance}(p, (0, 0)) \leq 1)$
- $X_i = 1$ and $X_i = 0$ happen often enough
- After 10^8 2D uniform$(-1, 1)$ variates $\pi = 3.14182 \pm 0.000493$ with 99.73% confidence

Calculating Mean and Error Incrementally

- Store and update $\sum X_i$ and $\sum X_i^2$
- After $n > 1$ simulations
  - $\bar{x} = \sum X_i / n$
  - $s = \max((\sum X_i^2 - \bar{x}^2/n)/(n - 1), 0)$
  - Need *max* for numerical issues!
  - Variance of mean = s/n

Monte Carlo Good and Bad

- $O(n)$ time and $O(1)$ space
- A simulated event can produce *k* values
  - Effectively perform *k* related simulations at cost of one
- $O(1/\sqrt{n})$ convergence is too slow
  - Variance reduction via **common random numbers** – fix everything that isn't simulated
  - E.g. when simulating performance of a randomized algorithm, run all simulations on the same input

Predict World Cup Winner

- Highest rated team may not be most likely winner
  - Team with relatively easiest opponents is
- Team ratings R_i determine game result probabilities
  - Prob(A wins against B) = $1/(1 + 10^D)$, where $D = (R_B - R_A)/400$
  - Use this to simulate the tournament tree
- Expected winner is the most frequent winner

Topics

- Quick review of probability
- Theory of randomization
- Generation of random objects
- Monte Carlo method
- Bootstrap method

Motivation

- Want to estimate $E_S[f(S)]$ where f is some function and S iid sample of size n from distribution T
- Monte Carlo applies if f = mean and can sample from T
- But what if f = median and have small fixed sample?
- Obvious $f(S)$ can be a bad estimator – bias and no error bounds

Bootstrap Idea

- $f(S)$ has some unknown distribution W, e.g. W is normal if f = mean
- S defines an empirical distribution with CDF $F(x) = \sum(x > S_i)/n$
- Dvoretzky-Kiefer-Wolfowitz inequality: $F \rightarrow T$ exponentially fast as $n \rightarrow \infty$
  - Let r_j be iid sample of size n from F
  - Heuristically $f(r_j)$ has about the same distribution as $f(S)$!
- $f(r_j)$ are effectively random samples from W

Algorithm

- b times for some large b
  - $r_i \leftarrow n$ random items from s with replacement
  - $f_i \leftarrow f(r_i)$
  - Return average f_i and its confidence interval
- No CLT – distribution of $f(S)$ may not be normal
  - Compute confidence interval by sorting f_i and finding those that enclose wanted confidence level %

Example

- On 1000 uniform01 values, with *f* = mean with confidence 95%
  - CLT gave 0.509 ± 0.018
  - Bootstrap with *b* = 10000 gave 0.500 –0.018/+0.019
- Bootstrap gave correct confidence without knowledge of the normal distribution!
- With *f* = median and same parameters bootstrap gave 0.480 –0.030/+0.028
- More complicated intervals may be better

Parameters

- Usually b = 10000 or as much as feasible
- No point to have $b \rightarrow \infty$, n limits accuracy
  - Number of distinct resamples ≈ $4^n/(n\sqrt{\pi n})$
  - Don't have something for nothing – bootstrap effectively extracts all info from S

Extentions

- Works with functions of multiple samples
  - E.g. to compute difference of medians of two samples, resample from both and use the difference of the resample medians
  - Can do many statistical tests this way

References

- Kedyk, D. (2018). *Commodity Algorithms and Data Structures in C++: Simple and Useful*. 3rd Edition. CreateSpace.

Lab

- Estimate π correctly to 3 decimal places
  - Use any programming language
  - That has a simple random number generator
  - Beware: accuracy of π ≠ accuracy of π/4
    - Multiplication by 4 magnifies the error