Class 08

Random Number Generation &

An Intro to Monte Carlo Methods

Outline

- Random Number Generators
- Engines, Distributions, and Seeds
- Intro to Monte Carlo Methods

Random Number Generation

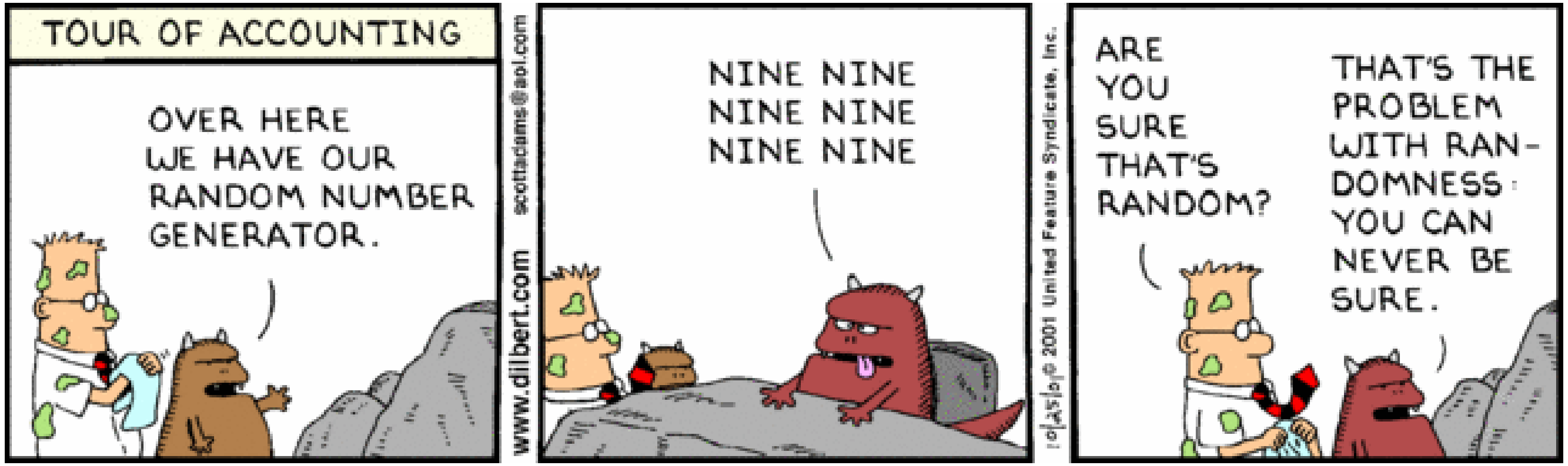
- A random number generator (RNG) is one that produces a stream of numbers in an undeterminable pattern.

- Many applications require the use of a random number.
  - Statistics & analysis
  - Simulations
  - Cryptography
  - Games

- Random Number Generation (RNG) can be based on hardware or on software.

Random Number Generation

- The quality of a random number generator that we are concerned about is its *period*, or how long it goes before repeating a number.



RNG – Reproducibility & Seeds

- When using random numbers we need to consider the reproducibility of those random numbers.

- We want systems to be reproducible to help facilitate debugging and for improving quality control.
  - Imagine your system crashes only *some* of the time. If you have unpredictable random inputs, tracing the root cause becomes immensely difficult!

- Engines use *seeds* to control the pattern that it creates. Use the same seed and the same exact pattern emerges from the engine. Not so random...

RNG – Deterministic vs. Indeterminism

- Engines are initialized with a seed; this seed may be set by the user or set with some form of *entropy*.
- Allowing the user to set the seed allows the system to be run with that same seed again to produce the same results. This is called *deterministic RNG*.
- Entropy is some characteristic about the (computing) environment at that moment in time. E.g. available RAM, system clock, etc. or some combination that is (nearly) impossible to replicate. Using entropy is called *indeterministic RNG*.

RNG – Engines & Distributions

- We can produce random and pseudo random numbers for our programs using a combination of generators (engines) and distributions.
  - All are provided by the C++ header *random*.
- Engines are tasked with producing a random number.
- Distributions are tasked with taking the number generated by the engine and transforming it according to some set of rules.
- C++ provides a handful of engines and many different distributions.
  - Common engine: std::mt19937
  - Common distributions: std::uniform_real_distribution, std::uniform_int_distribution, std::normal_distribution
  - These are all data types!

RNG – Engines & Distributions

- The engines and distributions in C++ are called *functors*.
  - They are variables that look and act like functions.

- Engines are called like functions without parameters:

```
auto r = eng();
```

- Distributions are called like functions, taking in an engine as a parameter:

```
auto r = dis(eng);
```

Engines – Random Device

- A random device is an indeterministic engine that usually uses hardware entropy as its seed.
  - Sensors on chip. e.g. temperature, sound, etc.
  - User inputs. e.g. mouse movement, key presses, etc.
  - CPU cycling/throttling, memory available, etc.
- Not consistent across platforms, and heavily dependent on the type of hardware entropy in use.
  - Poor entropy availability will result in poor randomness.
  - Not all systems can provide hardware entropy.
- Typically used to seed another engine.

Engines – Random Device

- Example

```
#include <random>
#include <fmt/format.h>

auto main() -> int
{
    auto rd = std::random_device{};
    for (int i = 0; i < 10; ++i)
    {
        fmt::print("{}\n", rd());  // produce a random number and print it
    }
}
```

Engines – Mersenne Twister

- This method generates a random number stream that has a period length equal to a *Mersenne Prime*.
  - Is considered a pseudorandom number generator
- A very common implementation uses the *Mersenne Prime* $2^{19937} - 1$ as its period.
  - The period of a random number generator is the number of samples that occur before a number is repeated.
- While this method is not cryptographically secure, it is perfectly well suited for many scientific methods.
  - Incredibly long period is useful
  - It is memory intensive compared to other engines, which may be more problematic on small devices with limited memory.

Engines – Mersenne Twister

43154247973881626480552355163379198390539350432267115051652505414033306801376580911304513629318584665545269938257648835317902217334584413909528269154609168019007875343741
39629680192011448648090266141431844327698030006672810498409545158817607713296984376213462179039639134128520562761960051310664637664861599423667548653748024196435029593516
86623639090479483476923139783013778207857124190544743328445291831729732423108882650813216264694510777078122828294447750226804880578200287646593991647662652009005614958003
44054353690389862894061792872011120833614808447482913547328367277879565648307846909116945866230169702401260240187028746650033445774570315431292996025187780790119375902863
17108414964247337898626750330896137490576634090528957229001603800057163087519137397955504746815433325347499104624813250451634179655147057548145920085947261483621387555711
68644457897508862779964873043084504842234206292665185560243393391908443689210184248446770427276646018529149252772809226975384267702573339289544012054658956103476588553866
33902546289962132643282425748035786233580608154696546932563833327670769899439774888526687278527451002963059146963875715425735534475979734463100678367393327402149930968778
29674139151459960237421362989872061143141040214723899809096281891589064569393448333099416963229587799584899336674701487176349480554999616305154122540346529700772114623135
57040814930986630657336771911728539870957481678162560842128233801686253345864312540346708061352735432707144788768618619833207772806448066911257131972625817631513135964295
47763576367837019349835178462144294960757190918054625114143666384189433852576452289347652454631535740468786228945885654608562058042468987372436921445092315377698407168198
37653823774861419620704154810637936512319281799900662176646716711347163271548179587700538269439340040306170045769113534918787488892342934934014517057171618112579588888927
74954269771499145496239163940148229850253316515114312788020090568084565068188772666098316368838849056218222629339865486456690806721917047404088913498356856624280632311985
20436826329415290752972798343429446509992206368781367154091702655772727391329424277529349082600585884766523150957417077831910016168475685658673192860882070179760307269849
98735483604237173466025769434723550630174411887414129243895814154910060975221688223088761143199647233084238013711092744948355781503758684964458574991777286992674421836962
11376751010832785437940817490940910430840967741447084363242794768920562004272279616386691498054898311212446763999319553714840128863607487064795686690485747828552170547401
13945929622177502575565811067452201448981991968635965361551681273982740760138899638820318776303668762730157584640042798880691862640268612686180883874939573818125022279689
93026744625577395954246983163786300017127922715140603412990218157065965053260077582367739818212908739444985918274999900722359242333456785067118656883918674770496001627754
06253314406190191299837899147125153652003360579935086016788076875685623778570952555413049029271922201841725023571244499118702106426945650613849193734743245039662677990384
02386781686809962015879090586549423504699190743519551043722544515740967829084336025938225780730880273855261551972044075620326780624448803490998232161231687794715613405793
24954550952805251801012308725877897411581704824558897143859675440808131343837550298872673952337529664161550140609160798322923982724061478325289247971651993698951918780868
12211916417477109024806334910917048274412282811866324459071457871383512348422613800746219140048181523866660431333448750679035828382835626880832365754820684796395463838195
32174522502682372441363275765875609119783653298312066708217149316773564340379289724393986744139891855416612295739356668612658271234696438377122838998040199739078061443675
41567107846340467370240377765347817336708484473470205686663615813800369225338220990946646959193016162609792050874217567030650513954286075080615983535754103214709508427846
10567013677397949320242029987077310176925820462107022125141204293225304317896162670477761151235979354041470848709854654265027720573009003338479053342506041195030300017040
02887892941404603345869926367501355094942750552591581639980523190679610784993580896683299297681262442314008657033421868094551740506448829039207316711307695131892296593509
01862309481055751956030524078716380921916443375451486330100091591698585624217656362477132898167854824629737624953025136036341276836645617507703197745753491280643317653999
59943433081184701471587128161493944212766142282629099500557469810532066100015602957846566161932522694120268311595089496715138451958832171479827488792618514178199790344172
85598607727220866677680426090308754823803345446566305619241308374452754668143015487710877728011086004325892262259413968285283497045571062757701421761565262725153407407625
40514993198949445910641466053430537857670986252004986488096114486925860347371436365919401396270636685138929969286949180517255681850829882495495481579606316951765874142015
97987542734280267234524812635691573072131537397810416276537150785985041547972876631229467113481585294188164328250444666927811374744948983850643757875073764963451486253063
8339155514569008789195531599446294449323524881759990711913575593382121706191477185054936632211157222920331148502487563303118018805685073569841580518118710778653953571296
01437294086527040702192438316729032323156791228941948624059403907445232167801938187121909215546076844457357855951361330424220615135645751393727093900970723782710124585383
76783381610233975868548942306960915402499879074534613119239638529507547580582056259566008177430071917468126559550217476709224608667477445208756078590623347506270983285934
80067789456169602494392813763495657599847485773553990957557313200809040830036446492219409934096948730547494301216165686750735749555882340303989874672975455060957736921559
19548081551403591570712993005702711728625284319741331230761788679750678426019543676030599034070848146460727895549548774214075357062121719825219297886978691673462561843017
5454903864111585429504569920905636741539030968041471

This is 6002 digits long!

Engines – Mersenne Twister

- Here we are creating a *std::mt19937* engine. Whenever we want to generate a new random number, we *call* the engine like it is a function.

```
#include <random>
#include <fmt/format.h>

auto main() -> int
{
    auto engine = std::mt19937{};
    fmt::print("{}\n", engine());
}
```

Engines – Mersenne Twister

- This sample below uses a new seed to change the pattern produced by the engine. Here our seed is 111.

```cpp
#include <random>
#include <fmt/format.h>

auto main() -> int
{
    auto engine = std::mt19937{111};
    fmt::print("{}\n", engine());
}
```

- You will commonly see the seed *1337* in code for this course. It has absolutely zero scientifically significant meaning. It is just a good number.

Engines - Mersenne Twister

- This sample below uses a special objected *random_device* to generate a random seed. This allows our program to generate something new every time we run it.

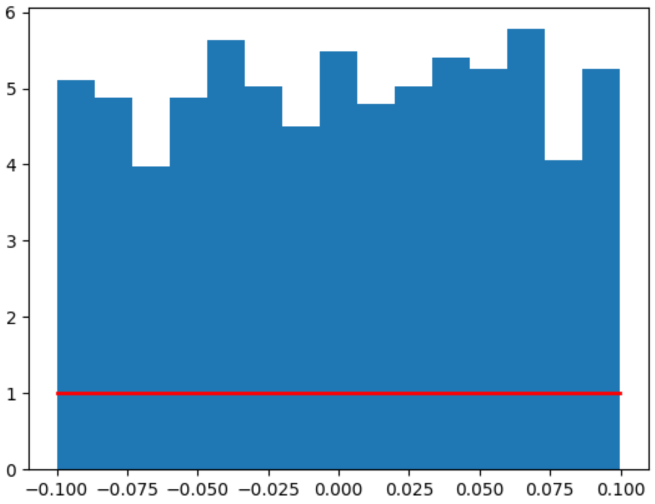
```
#include <random>
#include <fmt/format.h>

auto main() -> int
{
    auto rd = std::random_device{};
    auto engine = std::mt19937{rd()};
    fmt::print("{}\n", engine());
}
```

Distributions - Uniform

- *Uniform Distributions* provide an equal probability of sampling to every value within the specified range.
  - *i.e. all possibilities are equal!* If there are 10 possible outcomes, each of those outcomes would have a 10% chance of occurring (100 / 10 = 10)
  - Useful for representing a random event that shows no bias

- The diagram here shows binned counts of 1000 random samples taken uniformly between -0.1 and 0.1.

- The red line represents the *probability density* function, or the likelihood of selecting a random sample at that value. Here it is a flat line, and so all samples have an equal chance of being selected!



Distributions - Uniform

- C++ distinguishes between uniform distributions for integers and floating-point numbers.
  - *std::uniform_int_distribution* is used **only** for integer types
  - *std::uniform_real_distribution* is used **only** for floating-point types

- C++ however has more than one integer type, and more than one floating-point type, and so we need to explicitly tell these distributions what data type they are going to manage (like *std::vector*!).

```
auto d1 = std::uniform_int_distribution<int>{0, 100};
auto d2 = std::uniform_real_distribution<double>{0.0, 100.0};
```

Uniform Integer Distribution

- Example

```
#include <random>
#include <fmt/format.h>

auto main() -> int
{
    auto dis = std::uniform_int_distribution<>{1, 100};
    auto engine = std::mt19937{};
    fmt::print("{}\n", dis(engine));
}
```

Uniform Real Distribution

- Example

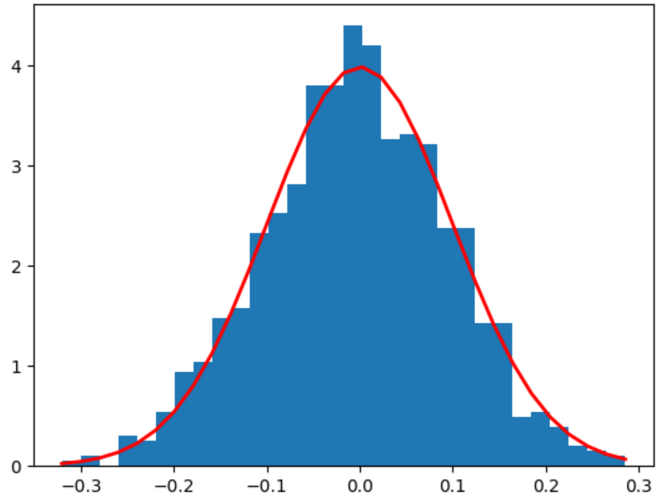
```cpp
#include <random>
#include <fmt/format.h>

auto main() -> int
{
    auto dis = std::uniform_real_distribution<>{1.0, 100.0};
    auto engine = std::mt19937{};
    fmt::print("{}\n", dis(engine));
}
```

Distributions - Normal

- *Normal Distributions* provide a biased probability of sampling values within some range of a center value (mean); values closer to the mean are more likely to occur.
  - More realistic for modeling real events by being able to capture likely and unlikely events.

- The diagram here shows binned counts of 1000 random samples taken from a mean of 0.0 and a standard deviation of ±0.1

- The red line represents the *probability density* function, or the likelihood of selecting a random sample at that value. Here the curve indicates that values closer to the mean are more likely!



Distributions - Normal

- C++ only allows normal distributions to be used with floating-point numbers!
- Like *std::uniform_real_distribution* though we still need to explicitly tell it what sort of floating-point it will manage!

Normal Distribution

- Example

```
#include <random>
#include <fmt/format.h>

auto main() -> int
{
    auto dis = std::normal_distribution<double>{50.0, 25.0};
    auto engine = std::mt19937{};
    fmt::print("{}\n", dis(engine));
}
```

Improper Use of Random Numbers

- What's wrong here?

```
#include <random>
#include <fmt/format.h>

auto main() -> int
{
    auto dis = std::uniform_real_distribution<int>{0, 10};
    auto engine = std::mt19937{};
    fmt::print("{}\n", dis(engine));
}
```

Improper Use of Random Numbers

```
#include <random>
#include <fmt/format.h>

auto main() -> int
{
    auto dis = std::uniform_real_distribution<int>{0, 10};
    auto engine = std::mt19937{};
    fmt::print("{}\n", dis(engine));
}
```

- Uniform real distributions only work with floating point types!

RNG – Common Tasks

- generating random points in 2D or 3D space
  - generate 2 or 3 numbers, and together they represent a point!
- adding noise to data
  - sometimes we want to fuzz data to make it appear more realistic, like when simulating signal processing
- strategy patterns
  - predictive analytics – understand the depth of your system and where it can go
  - AI design – use randomness to represent the choices made by AI

Baking a Cake

- A recipe for a cake may specify the following:
  - Ingredients:
    - 3 cups flour
    - 1 tbsp baking soda
    - 1 tsp salt
    - 1 cup butter
    - 1 cup milk
    - 5 eggs
  - Directions
    - mix
    - bake for 30 min at 350F

Baking a Cake

- A recipe for a cake may specify the following (not this simply...):
  - Ingredients:
    - 3 cups flour
    - 1 tbsp baking soda
    - 1 tsp salt
    - 1 cup butter
    - 1 cup milk
    - 5 eggs
  - Directions
    - mix
    - bake for 30 min at 350F

How perfectly are you measuring each of these quantities?

What if you miscounted 4 eggs?

How well did you mix?
Was the oven really at 350F?
Did it bake for exactly 30 minutes?
How is the heat distributed in the oven?

Monte Carlo Methods

- Monte Carlo Methods are a class of techniques that utilize random number generation to aid in representing uncertainty.
- In theory algorithms and measurements are precise and without error. The simulation is a perfect world.
- In practice apparatus are imperfect, human elements are unpredictable, and details are unaccounted for.
- Monte Carlo Methods allow an algorithm to run with randomly imperfect data so that one execution to the next is different, which over time will paint a better picture.

Monte Carlo Methods

- Various aspects of the system are represented with varying distributions that align reasonably well with their realistic behavior.
  - This is usually dictated by real-world measurements and/or subject matter experts that understand the system exceptionally well.
- We run a Monte Carlo method many times over until the resulting data *converges*.
  - This can take tens, hundreds, and even thousands of executions to observe a convergence in the data.
- When checking for a numerical convergence, we utilize some error threshold (epsilon).
  - This will typically be $\varepsilon = 1e^{-8}$ (this is a really small number!)
  - If our data is off from the expect result by a difference less than ε then we say our result is close enough.

List of Engines & Distributions

- https://en.cppreference.com/w/cpp/header/random