Simulation Modeling

Goal: Use probabilistic methods to analyze deterministic and probabilistic models.

Example. Determine the best elevator delivery scheme.

- The wait is too long, too many stops along the way.
- Inconvenient to experiment with alternate delivery schemes.
    - Disrupt normal service
    - Take surveys of customers
    - Confuse regular customers
- Alternatively, run a computer **simulation**. Write a computer program that models the system of elevators, including:
    - Time of arrival of passengers (a random event)
    - Passenger destination (a random event)
    - Capacity of elevator (fixed by system)
    - Speed of elevator (fixed by system)
    - Current delivery scheme

Simulation Modeling

Once you have written the computer program,

Verify that the simulation models the current real-world situation

- Run the model many times.
- Have the computer keep track of data, such as average wait time, number of stops it takes, longest queue, etc.

Then, modify various parameters in order to simulate a new delivery scheme.

- How do the data change?
- Is the alternate scheme better or worse?
- Determine how to implement to cause minimal disruption.

Monte Carlo Simulations

Definition: A simulation that incorporates an element of randomness is called a **Monte Carlo** simulation.

PROS:

- It is a relatively easy method to approximate complex systems.
- Once built, it allows for tinkering—easy to do sensitivity analysis.
- It can model systems over difficult-to-measure time frames.

CONS:

- You have to build it. (Expensive to develop!)
- Requires computing power and time.
- Makes you over-confident in the results.
- Dealing with probability, so results will always be of the form: "With 95% probability, the wait time will be less than 2 minutes."

Simulating flipping a coin

Example. Get a computer to simulate flipping a fair coin 20 times.

To simulate a random event, use one of the *Mathematica* commands:

- `RandomInteger` gives a pseudo-random *integer*.
    - `RandomInteger[]` (no input) gives either 0 or 1.
    - `RandomInteger[5]` gives an integer from 0 to 5.
    - `RandomInteger[{1, 10}]` gives an integer from 1 to 10.
    - `RandomInteger[{1, 10},20]` gives a list of 20 such integers.
- `RandomReal` gives a pseudo-random *real number*.
    - `RandomReal[]` (no input) gives a real number between 0 or 1.
    - `RandomReal[{0.1, 0.2}]` gives a real number from 0.1 to 0.2.
    - `RandomReal[{0.1, 0.2},15]` gives a list of 15 such numbers.

The first input gives the range; a second input tells how many to make.

The numbers produced by a random number generator are never truly random because they are produced by an algorithm on a deterministic machine.

Simulating flipping a coin

Example. Get a computer to simulate flipping a fair coin 20 times.

Let’s choose a convention: 1=‘Head’ and 0=‘Tail’.
Evaluating `RandomInteger[1,20]` generates a list of 20 coin tosses.

```
 In[1]: CoinFlips = RandomInteger[1,20]
Out[1]: {1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1}
```

The sum of this list is the total number of heads tossed.

```
 In[2]: Total[CoinFlips]
Out[2]: 13
```

Running the commands again will simulate another trial of 20 flips.

`If` statements and `For` loops

In order to incorporate more complex aspects into the model, use `If` statements and `For` loops.

`If[condition,t,f]`

- First, *Mathematica* evaluates the ‘`condition`’.
- If ‘`condition`’ is true, the statement evaluates the ‘`t`’ part.
- If ‘`condition`’ is false, the statement evaluates the ‘`f`’ part.

Examples of conditions:

`x<0` `(x==0) && (y!=1)` `RandomInteger[]==1`

Note the **double equals** sign == and **not equals** `!=`.

Examples.

- `If[x<0, -x, x]` is ______________________________.
- `If[RandomInteger[] == 1, "Head", "Tail"]:`

Using `If` statements in `Table` commands

Goal: Model something that happens 7.5% of the time.

Call `RandomReal[]` to output: ______________________________.

The output satisfies ______________________ 7.5% of the time.

Anything to the left of the split will be taken as success.

To model this is *Mathematica*, use an `If` statement.

```
trial = RandomReal[]
success = If[trial <= 0.075, 1, 0]
```

Alternatively, do this is one step:

```
If[RandomReal[] <= 0.075, 1, 0]
```

Using `If` statements in `Table` commands

That was: `If[RandomReal[] <= 0.075, 1, 0]`

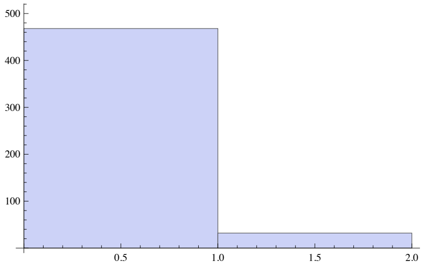
Let's run this command many times and visualize the results: Remember that `Table` will repeat a command multiple times:

```
trials = Table[If[RandomReal[] <= 0.075, 1, 0], {500}];
```

Output: 500-entry list, where each entry is 0 (failure) or 1 (success).

Question: How many successes? (Expected value: $500 \cdot 0.075 = 37.5$)

- If we add the entries `Total[trials]`, we get $\#$ successes. One time I ran it had 32 successes.
- Alternatively, `Tally[trials]` gives how many times distinct entries appear. Output: `{{0, 468}, {1, 32}}`
- Last, we might want a visualization; Use `Histogram[trials]` to get:



`If` statements and `For` loops

`For[start,test,incr,body]`

- First, *Mathematica* evaluates the code in `start`.
- As long as `test` is true, (Can happen many times!)
- Continue to evaluate `body` and do the increment `incr`.

Example. `For[i = 0, i < 4, i++, Print[i]]`

- First, *Mathematica* defines `i` to be equal to 0.
- Next, it checks to see if `i` is less than 4.
- It is, so it evaluates `Print[i]`, and increments `i` by 1 (`i++`).
- Now `i` $= 1$, which is still < 4. So '`Print[i]`' is evaluated and `i` is incremented. Similarly for `i` $= 2$ and `i` $= 3$. Now `i` is incremented to 4, which is NOT < 4, and the loop terminates.

This variable `i` is called a **counter**.

Be careful to name counters wisely! They are defined as variables.

Simulating flipping a coin

Example. Simulate flipping a fair coin 20 times using a `for` loop.

We'll write some **pseudocode**—words that explain what we want the computer to do, but won't actually work if we typed them in.

- Run the loop 20 times.
  (Keep track using a counter: let `loopCount` vary from 1 to 20.)
- Each time the loop evaluates,
  - Generate a random integer between 0 and 1.
  - If '1' output 'Head', if '0', output 'Tail'.

```
For[loopCount = 1, loopCount <= 20, loopCount++,
    flip = RandomInteger[];
    If[flip == 1, Print["Head"], Print["Tail"]]]
```

- Notice the == and also the ; that separates the commands.
- `loopCount` is ONLY a counter; it does not change each step's evaluation.

Simulating flipping a coin

Pimp my code! Let's keep track of # heads and tails with counters.

`headCount`: # of heads so far. `tailCount`: # of tails so far.

- Zero out the counters: '`headCount`=0' and '`tailCount`=0'.
- Run the loop 20 times by having `loopCount` vary from 1 to 20.
- Each time the loop evaluates,
  - Generate a random integer between 0 and 1.
  - If '1', output 'Head' **AND** increase '`headCount`',
  - If '0', output 'Tail' **AND** increase '`tailCount`'.
- After 20 iterations, display '`headCount`' and '`tailCount`'.

```
headCount=0; tailCount=0;
For[loopCount = 1, loopCount <= 20, loopCount++,
  If[RandomInteger[]==1,
     Print["Head"]; headCount++,          ← Notice the ';'
     Print["Tail"]; tailCount++]]         ← Notice the '++'
{headCount, tailCount}
```

Simulating rolling a biased die

Suppose you have a four-sided die, where the four sides (A, B, C, and D) come up with probabilities 1/2, 1/4, 1/8, and 1/8, respectively.

- Reset the counters: ‘aCount=bCount=cCount=dCount=0’.
- For loopCount from 1 to 20,
  - Generate a random real number between 0 and 1.
  - If between 0 and 1/2, then output ‘A’ and aCount++
    if between 1/2 and 3/4, then output ‘B’ and bCount++
    if between 3/4 and 7/8, then output ‘C’ and cCount++
    if between 7/8 and 1, then output ’D’ and dCount++
- Display ‘aCount’, ‘bCount’, ‘cCount’, and ‘dCount’.

Simulating rolling a biased die

```
aCount = 0; bCount = 0; cCount = 0; dCount = 0;
For[loopCount = 1, loopCount <= 20, loopCount++,
 roll=RandomReal[];
 If[ 0  <= roll < 1/2, Print["a"]; aCount++];
 If[1/2 <= roll < 3/4, Print["b"]; bCount++];
 If[3/4 <= roll < 7/8, Print["c"]; cCount++];
 If[7/8 <= roll <= 1 , Print["d"]; dCount++];]
distribution = {aCount, bCount, cCount, dCount}
```

- Sample output: (each on its own line)

  a, a, a, d, d, b, a, a, d, a, a, a, a, d, b, a, a, c, a, b $\{12, 3, 1, 4\}$
- These `If` statements all have no "False" part. (; vs ,)
- Important: You MUST set a variable for the roll. Otherwise, calling `RandomInteger` four times will have you comparing different random numbers in each `If` statement.
- If you are feeling fancy, you can use one `Which` command instead of four `If` commands.

Using Simulation to Calculate Area

Suppose you have a region whose area you don't know. You can approximate the area using a Monte Carlo simulation.

Idea: Surround the region by a rectangle. Randomly chosen points in the rectangle will fall in the region with probability

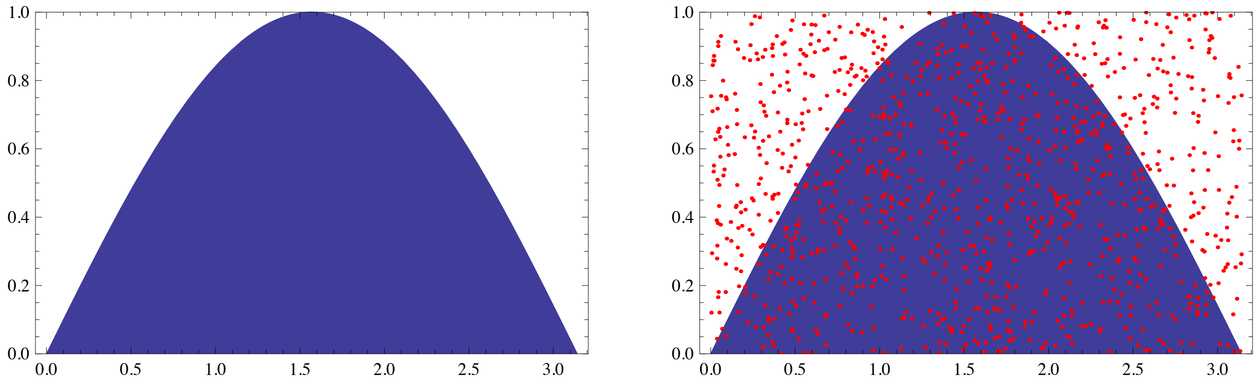
(area of region)/(area of rectangle)

We can approximate this probability by calculating

(points falling in region)/(total points chosen).

Using Simulation to Calculate Area

Example. What is the area under the curve $\sin(x)$ from 0 to π?



Randomly select 100 points from the rectangle $[0, \pi] \times [0, 1]$.

[*Choose a random real between* 0 *and* π *for the* x*-coordinate and a random real between* 0 *and* 1 *for the* y*-coordinate...*]

$$\text{Then, } \frac{\text{Area of region}}{\underline{\qquad\qquad}} \approx \frac{\text{Number of points in region}}{100}.$$

Here, 63 points fell in the region; we estimate the area to be $\underline{\qquad\qquad}$.

Compare this to the actual value, $\int_{x=0}^{x=\pi} \sin(x)\, dx = [-\cos(x)]_{x=0}^{x=\pi} = 2$