BEE 4750 Lab 2: Uncertainty and Monte Carlo

Name: Maya Yu

ID: mzy3

> **Due Date**
>
> Friday, 9/22/23, 9:00pm

Setup

The following code should go at the top of most Julia scripts; it will load the local package environment and install any needed packages. You will see this often and shouldn't need to touch it.

In [ ]:
```
import Pkg
Pkg.activate(".")
Pkg.instantiate()
```

```
  Activating project at `~/BEE 4750/lab-02-mayazoeyu`
```

In [ ]:
```
using Random # random number generation
using Distributions # probability distributions and interface
using Statistics # basic statistical functions, including mean
using Plots # plotting
```

Introduction

In this lab, you will use Monte Carlo analysis to estimate the expected winnings for a couple of different games of chance.

Monte Carlo methods involve the simulation of random numbers from probability distributions. In an environmental context, we often propagate these random numbers through some more complicated model and then compute a resulting statistic which is relevant for assessing performance or risk, such as an average outcome or a particular quantile.

Julia provides a common interface for probability distributions with the `Distributions.jl` package (https://juliastats.org/Distributions.jl/stable/). The basic workflow for sampling from a distribution is:

1. Set up the distribution. The specific syntax depends on the distribution and what parameters are required, but the general call is the similar. For a normal distribution or a uniform distribution, the syntax is

   ```
   # you don't have to name this "normal_distribution"
   # μ is the mean and σ is the standard deviation
   normal_distribution = Normal(μ, σ)
   # a is the upper bound and b is the lower bound; these can be set to +Inf
   or -Inf for an unbounded distribution in one or both directions.
   uniform_distribution = Uniform(a, b)
   ```

   There are lots of both univariate (https://juliastats.org/Distributions.jl/stable/univariate/#Index) and multivariate (https://juliastats.org/Distributions.jl/stable/multivariate/) distributions, as well as the ability to create your own, but we won't do anything too exotic here.
2. Draw samples. This uses the `rand()` command (which, when used without a distribution, just samples uniformly from the interval $[0, 1]$.) For example, to sample from our normal distribution above:

   ```
   # draw n samples
   rand(normal_distribution, n)
   ```

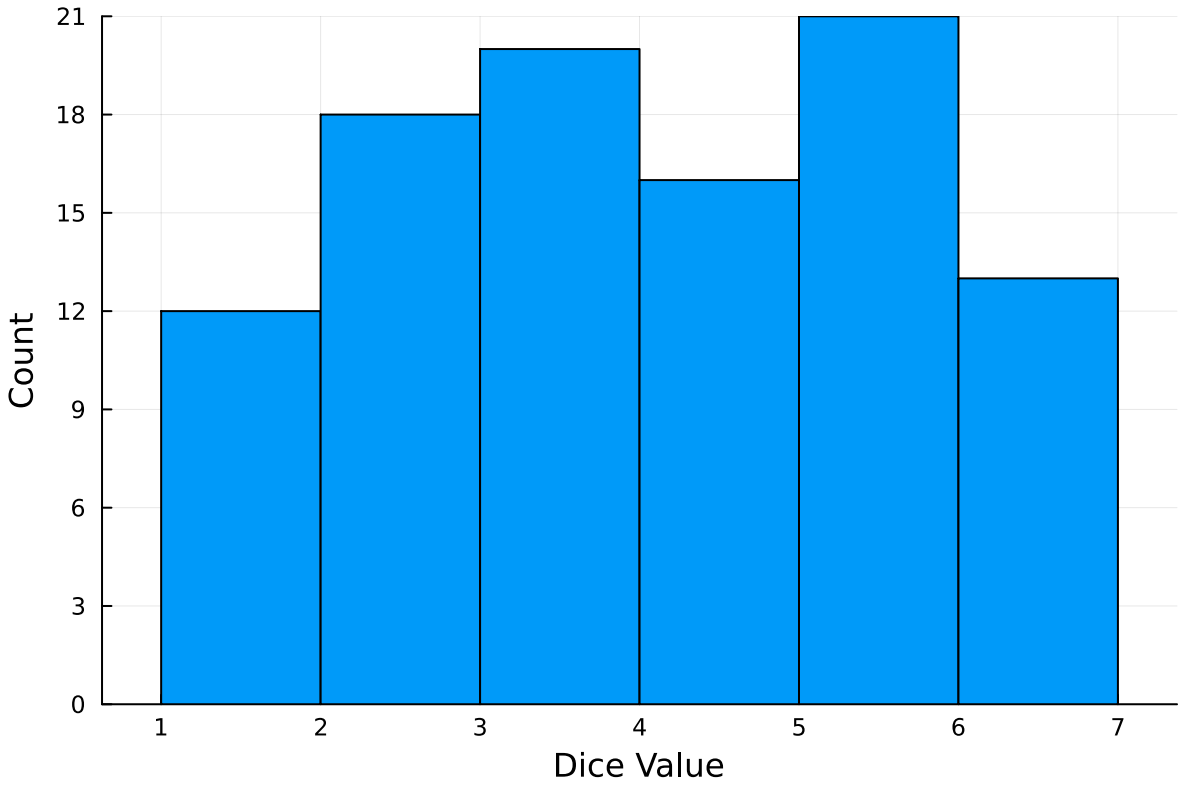
Putting this together, let's say that we wanted to simulate 100 six-sided dice rolls. We could use a Discrete Uniform distribution (https://juliastats.org/Distributions.jl/stable/univariate/#Distributions.DiscreteUniform).

In [ ]:
```
dice_dist = DiscreteUniform(1, 6) # can generate any integer between 1 a
nd 6
dice_rolls = rand(dice_dist, 100) # simulate rolls
```

```
100-element Vector{Int64}:
 4
 3
 2
 2
 5
 3
 1
 6
 1
 3
 ⋮
 5
 2
 5
 1
 2
 6
 5
 1
 6
```

And then we can plot a histogram of these rolls:

In [ ]:
```
histogram(dice_rolls, legend=:false, bins=6)
ylabel!("Count")
xlabel!("Dice Value")
```



Remember to:

- Evaluate all of your code cells, in order (using a `Run All` command). This will make sure all output is visible and that the code cells were evaluated in the correct order.
- Tag each of the problems when you submit to Gradescope; a 10% penalty will be deducted if this is not done.

Exercises (10 points)

In Problem 1, you will compute the probability of getting a specific combination of multiple dice rolls. The focus will be on understanding how the Monte Carlo estimate changes based on the number of simulations.

In Problem 2, we will implement the culmination of every episode of the long-running game show The Price Is Right (https://en.wikipedia.org/wiki/The_Price_Is_Right): the Showcase (https://en.wikipedia.org/wiki/The_Price_Is_Right#Showcase). You will be asked to make a plot of expected winnings by bid for a particular distribution of prize values.

You should always start any computing with random numbers by setting a "seed," which controls the sequence of numbers which are generated (since these are not *really* random, just "pseudorandom"). In Julia, we do this with the `Random.seed!()` function.

```
Random.seed!(1)
```

```
TaskLocalRNG()
```

It doesn't matter what seed you set, though different seeds might result in slightly different values. But setting a seed means every time your notebook is run, the answer will be the same.

> **Seeds and Reproducing Solutions**
>
> If you don't re-run your code in the same order or if you re-run the same cell repeatedly, you will not get the same solution. If you're working on a specific problem, you might want to re-use `Random.seed()` near any block of code you want to re-evaluate repeatedly.

Problem 1 (5 points)

We want to know the probability of getting at least an 11 from rolling three fair, six-sided dice (this is actually an old Italian game called *passadieci*, which was analyzed by Galileo as one of the first examples of a rigorous study of probability).

Problem 1.1 (1 point)

Write a function called `passadieci()` to simulate this game, which will take as an input the number of realizations and output a vector of the sum of the three dice rolls for each realization.

In [ ]:
```
#### Problem 1.1 Model ####

### Create random seed
Random.seed!(1)

### Create a function to simulate the passadieci game
function passadieci(realizations)
    sum_of_rolls = zeros(realizations)
    for i in 1:realizations
        distribution = DiscreteUniform(1, 6) # Create the distribution o
f the dice rolls
        rolls = rand(dice_dist, 3) # Simulate 3 rolls
        sum_of_rolls[i] = sum(rolls) # Evaulate the sum
    end
    return sum_of_rolls
end
```

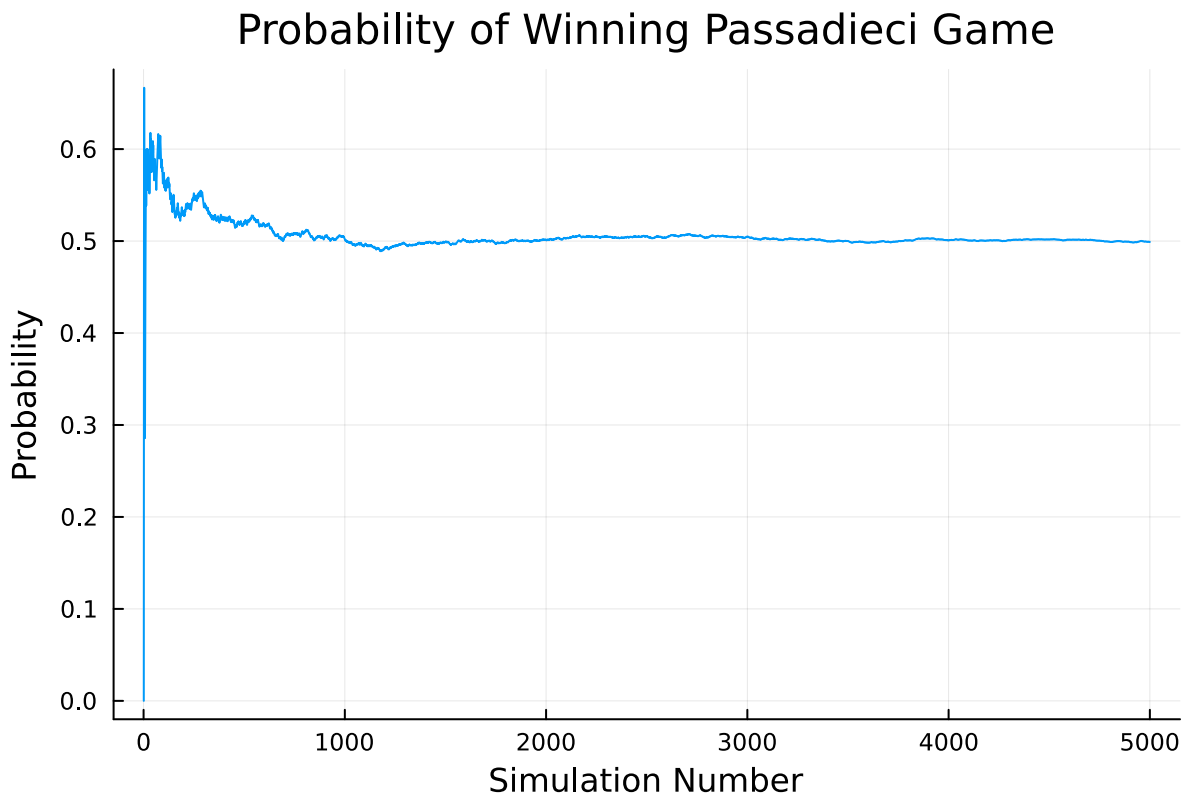
```
passadieci (generic function with 1 method)
```

Problem 1.2 (2 points)

Generate 5,000 simulations of the game using your `passadieci()` function. Plot how the computed probability of winning the game changes as the number of simulations increases (you can do this by computing the frequency of wins for each additional simulation).

In [ ]:

```
#### Problem 1.2 Model ###

### Create random seed
Random.seed!(1)

# Initialize for loop
simulation = passadieci(5000)
counter = 0;
probability = zeros(5000)

# Create for loop for simulating 5000 rolls and count if the sum is grea
ter than 11
for j in 1:5000
    if simulation[j] >= 11
        counter = counter + 1
    end
    probability[j] = counter/j
end
return probability

plot(probability, xlabel = "Simulation Number", ylabel = "Probability",
legend = false,
title = "Probability of Winning Passadieci Game")
```



Problem 1.3 (2 point)

Based on your plot from Problem 1.2, how many simulations were needed for the win probability estimate to converge? What did you notice from your plot about the estimates prior to convergence?

Problem 1.3 Answer

Based on the graph, it looks like the plot begins to converge around 900 trials. Prior to the convergence, the probability varies significantly. This is because there is a element of randomness when running the trials – it will not converge and it is highly dependent on the current rolls. However, as the iterations increase, they will converge more rapidly.

Problem 2 (5 points)

The Showcase is the final round of every episode of The Price is Right, matching the two big winners from the episode. Each contestant is shown a "showcase" of prizes, which are usually some combination of a trip, a motor vehicle, some furniture, and maybe some other stuff. They then each have to make a bid on the retail price of the showcase. The rules are:

- an overbid is an automatic loss;
- the contest who gets closest to the retail price wins their showcase;
- if a contestant gets within $250 of the retail price and is closer than their opponent, they win both showcases.

Your goal is to find a wager which maximizes your expected winnings, which we may as well call utility, based on your assessment of the probability of your showcase retail price. We'll assume that the distribution of all showcases offered by the show is given as truncated normal distribution, which means a normal distribution which has an upper and/or lower bound. `Distributions.jl` makes it easy to specify truncations on any distribution, not just normal distributions. For example, we'll use this distribution for the showcase values:

```
showcase_dist = truncated(Normal(31000, 4500), lower=5000, upper=42000)
```

```
Truncated(Normal{Float64}(μ=31000.0, σ=4500.0); lower=5000.0, upper=42000.0)
```

Problem 2.1 (3 points)

Write a function `showcase()` which takes in a bid value and uses Monte Carlo simulation to compute the expected value of the winnings. Make the following assumptions about your expected winnings if you don't overbid:

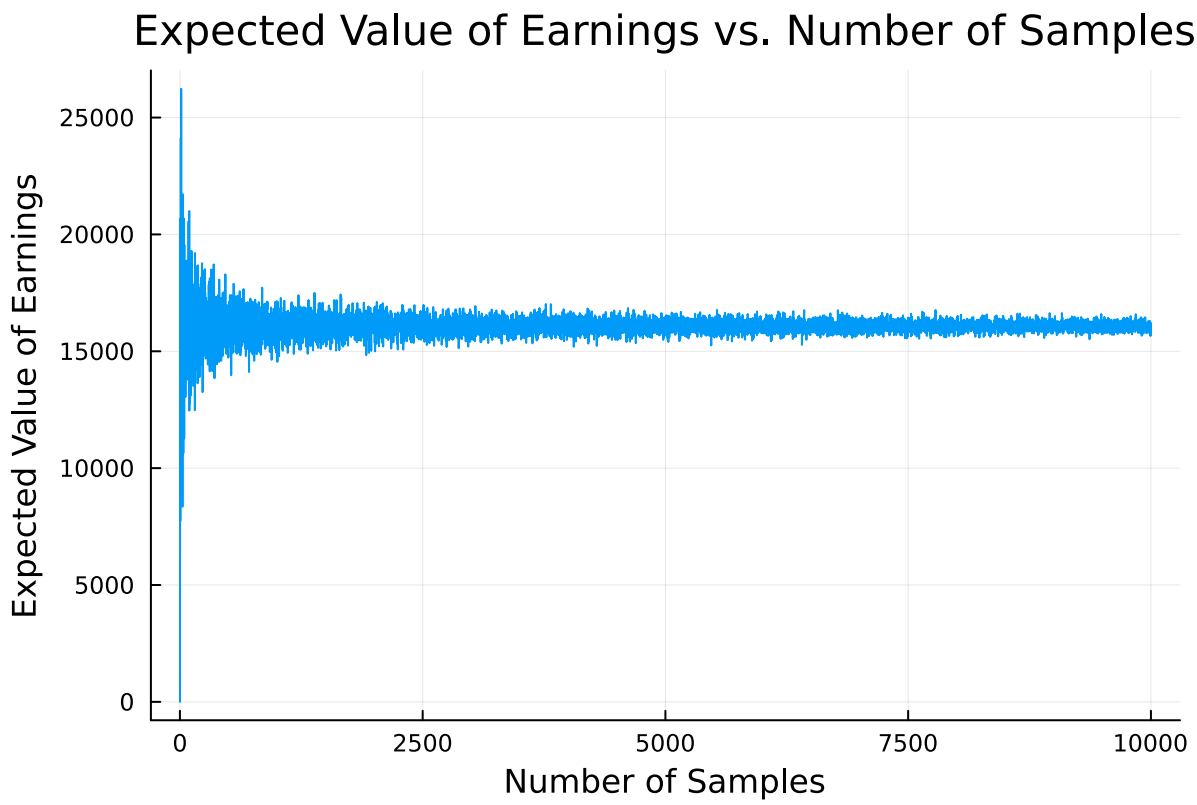
- If you win both showcases, the value is the double of the single showcase value.
- If you did not win both showcases but bid under the showcase value, the probability of being outbid increases linearly as the distance between your bid and the value increases (in other words, if you bid the exact value, you win with probability 1, and if you bid $0, you win with probability 0).

How did you decide how many samples to use within the function?

In [ ]:

```
#### Problem 2.1 Model ####

### Specify the distribution of showcase value
showcase_dist = truncated(Normal(31000, 4500), lower = 5000, upper = 420
00)

### Add random seed
Random.seed!(2)

### Create pilot function that models the bid and receives an input of n
samples
function showcase_pilot(bid, n)
    # Initialize earnings
    earnings = 0
    # Model the bidding outcomes for n iterations
    for i in 1:n
        # Model showcase value based on distribution
        showcase_value = rand(showcase_dist)
        # Model the 3 possible bidding outcomes
        if bid > showcase_value
            win = 0
        elseif showcase_value - bid <= 250
            win = 2*showcase_value
        else
            # Calculate probability of winning
            probability = bid/showcase_value
            win = probability*showcase_value
        end
        earnings = earnings + win # Calculate total earnings
    end
    # Calculate the expected value
    expected = earnings/n
    return expected
end

### Finalize function based on convergence of plot
function showcase(bid)
    # Initialize earnings
    earnings = 0
    # Model the bidding outcomes for n iterations
    for i in 1:7500
        # Model showcase value based on distribution
        showcase_value = rand(showcase_dist)
        # Model the 3 possible bidding outcomes
        if bid > showcase_value
            win = 0
        elseif showcase_value - bid <= 250
            win = 2*showcase_value
        else
            # Calculate probability of winning
            probability = bid/showcase_value
            win = probability*showcase_value
        end
        earnings = earnings + win # Calculate total earnings
    end
    # Calculate the expected value
```

```
    expected = earnings/7500
    return expected
end

### Plot for convergence point
distribution = zeros(10000)
for x in 1:10000
    distribution[x] = showcase_pilot(31000, x)
end
plot(distribution, yformatter = :plain, xlabel = "Number of Samples", yl
abel = "Expected Value of Earnings",
    legend = false, title = "Expected Value of Earnings vs. Number of Sa
mples")
```



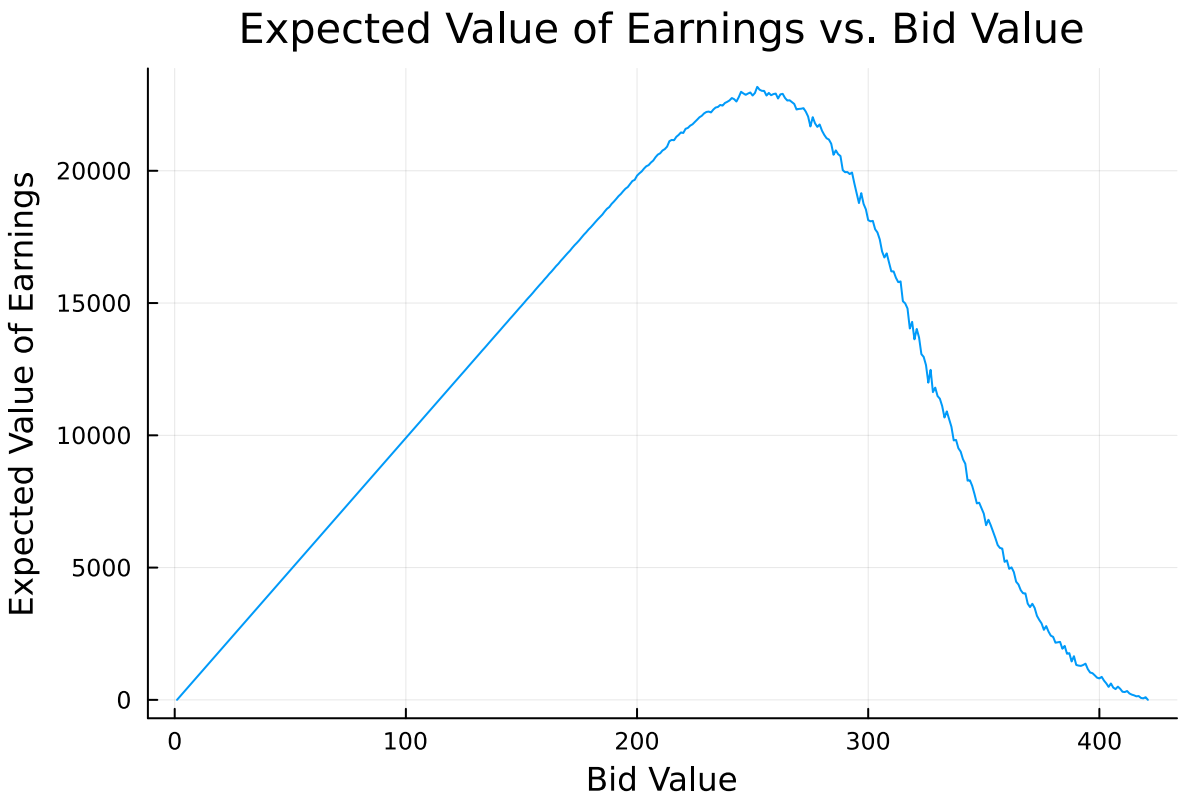
Problem 2.1: Number of Samples

To determine an adequate number of samples, I plotted a pilot function "showcase_pilot" that allowed for the input of an arbitrary number of samples. I then plotted the distribution of showcase_pilot for various sample numbers. As per the above plot, the expected value of "showcase_pilot" converges around 2500. However, there is some noise that still occurs. Increasing the number of samples reduces the variability of the expected value. Therefore, I decided to opt for 7500 samples – as it has a more stable convergence. I then created a final function "showcase" that has the 7500 sample number built into the function.

Problem 2.2 (2 points)

Plot the expected winnings for bids ranging from $0 to $42,000. What do you notice?

In [ ]:
```
#### Problem 2.2 Model ####

### Calculate expected earnings for bids ranging from $0 to $42,000
x = 0:100:42000
expected_winnings = (a -> showcase(a)).(x)

### Plot the expected earnings
plot(expected_winnings, xlabel = "Bid Value", ylabel = "Expected Value o
f Earnings", legend = false, yformatter = :plain,
    xformatter = :plain, title = "Expected Value of Earnings vs. Bid Val
ue")
```



Problem 2.2: Plot Observations

Based on the plot, we see that the bid value that will maximize the average earnings is around $25,000. This bid value will yield an expected earning of around $25,000. In addition, for bid values less than $25,000, we see that the expected value increases nearly linearly with the bid value. This growth is relatively slow compared to the steep drop-off that occurs when bidding over $25,000. As such, the distribution is not symmetric around the bid value of $25,000.

References

Put any consulted sources here, including classmates you worked with/who helped you.

Caleb Julian-Kwong, Kiley Espineira