BEE 4750 Lab 2: Uncertainty and Monte Carlo

Name: Anthony Nicolaides

ID: ajn68

> **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 `~/Documents/BEE4750/labs/lab-02-anthonynic28`
```

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. 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 and 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.

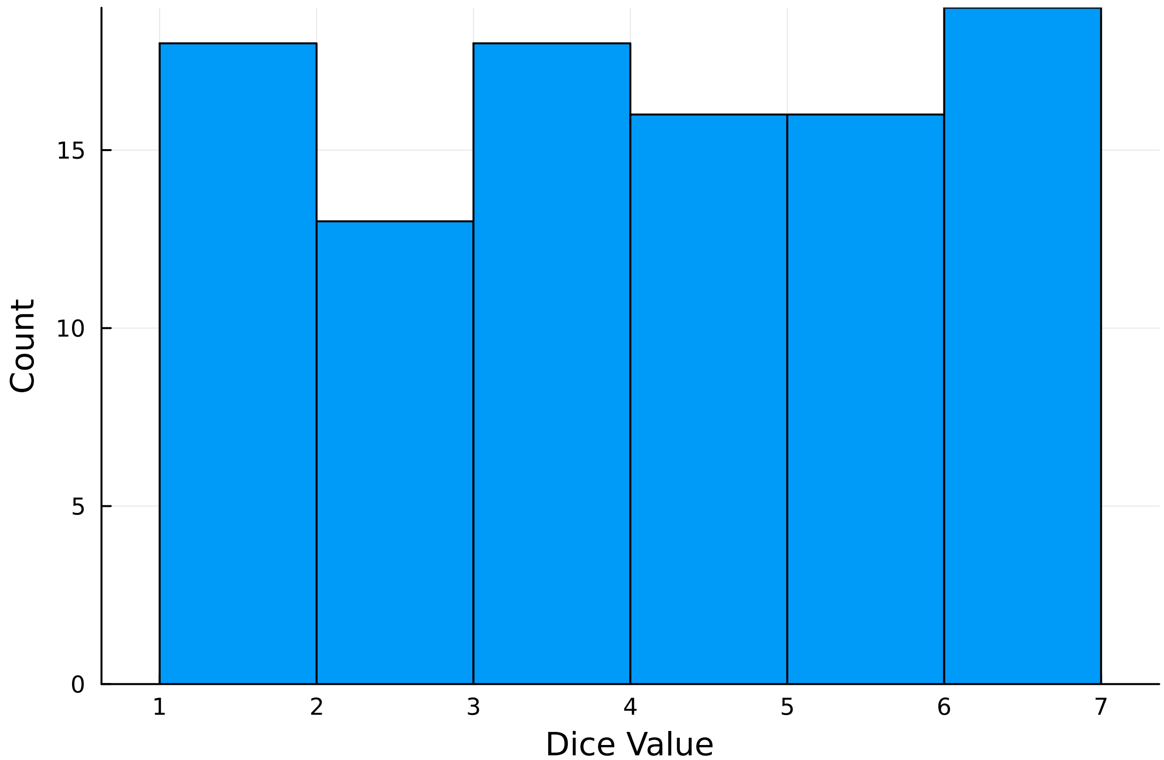
In [ ]:
```
dice_dist = DiscreteUniform(1, 6) # can generate any integer between 1 and 6
dice_rolls = rand(dice_dist, 100) # simulate rolls
```

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

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: the 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 [ ]:
```
function passadieci(num_realizations)
    Random.seed!(1) # set random seed to have consistent output
    dice_dist = DiscreteUniform(1, 6) # can generate any integer b/w 1 & 6
    sum_of_roll = zeros(num_realizations)
    for i = 1:num_realizations
        dice_rolls = rand(dice_dist, 3) # simulate rolls of 3 die
        sum_of_roll[i] = sum(dice_rolls) # record sum of those 3 rolls
    end
    return sum_of_roll
end
```

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

Problem 1.2 (2 points)

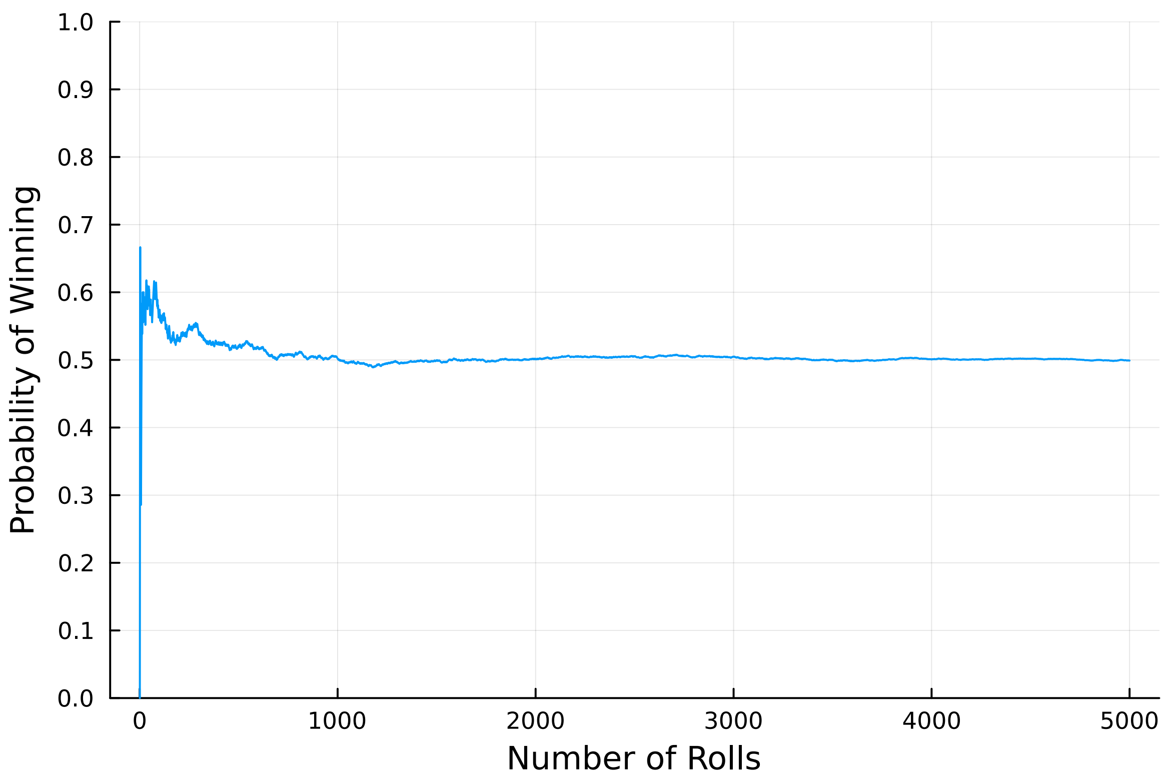
Generate 5,000 simulations of the game using your `passadieci()` function. Plot how the computed probability of winning the game changes as the number of simulations increases (you can do this by computing the frequency of wins for each additional simulation).

In [ ]:

```
array_of_sims = zeros(5000)
for i = 1:5000
    # generate i amount of rolls and assign the output to sim_of_rolls
    sim_of_rolls = passadieci(i)

    # calculate probability of getting at least 11
    prob_of_win = (sum([x >= 11 for x in sim_of_rolls])) / i

    array_of_sims[i] = prob_of_win
end
plot(array_of_sims, legend=:false)
ylabel!("Probability of Winning")
xlabel!("Number of Rolls")
ylims!(0, 1)
yticks!(0:0.1:1)
```



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?

It takes about 1500 simulations of rolls for the win probability to converge to 0.5. Prior to the convergence, the estimates of the win probability started off high and slowly started to decrease to 0.5 as the number of rolls increased. Also, the estimates seem to oscillate up and down before converging as well.

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 [ ]:
```
function showcase(bid_value, sample_size)
    Random.seed!(1) # set random seed to have consistent output
    showcase_dist = truncated(Normal(31000, 4500), lower=5000, upper=42000)
    expected_winnings = 0
    for sample = 1:sample_size
```

```julia
        win_prob = 0
        # generate showcase value for both showcases
        #   (both are the same value)
        showcase_value = rand(showcase_dist)

        if (bid_value < showcase_value) # if overbid -> automatically lose
            if (showcase_value - bid_value) <= 250
                win_prob = 1 # auto win if within 250 of showcase value
            else
                # linear relationship b/w bid and showcase value
                win_prob = (bid_value / showcase_value)
            end
        end
        expected_winnings = ((win_prob * showcase_value) +
                             (win_prob * showcase_value)) +
                            expected_winnings
    end
    return expected_winnings / sample_size
end
```

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

In [ ]:

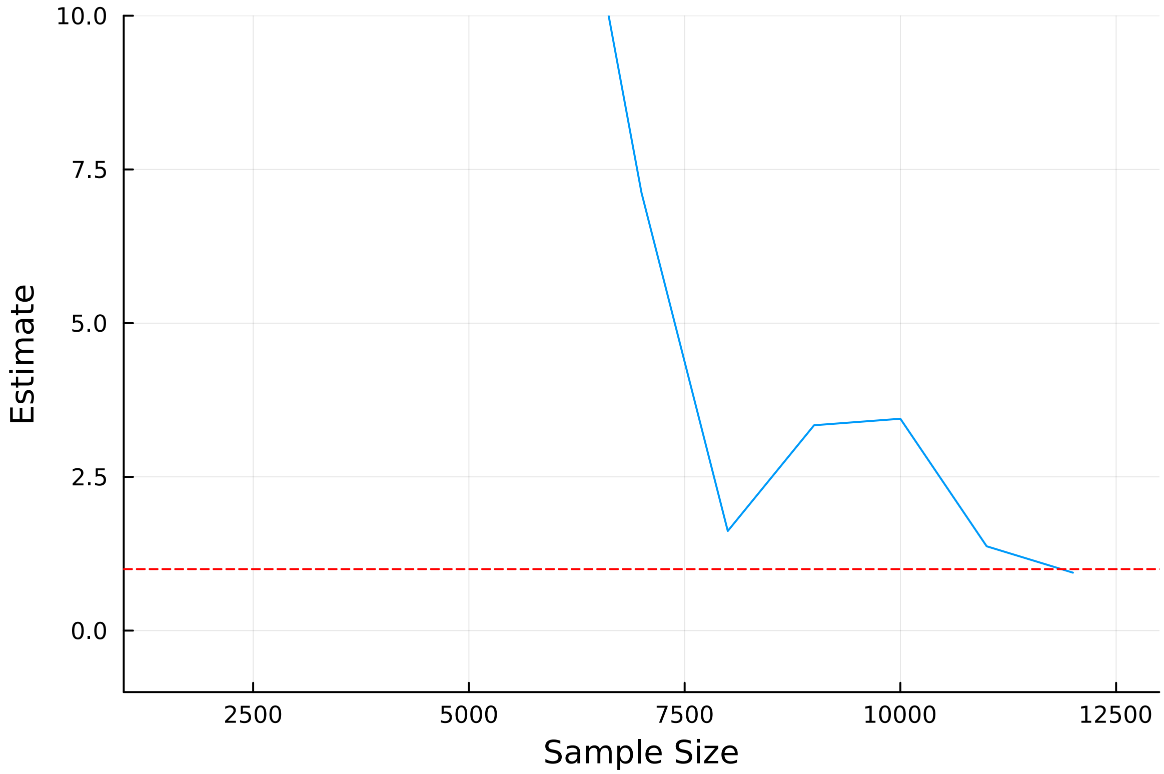
```julia
# delcare variables
desired_accuracy = 1 # using mean-squared error (MSE)
n0 = 1000 # sample size, start with large sample size for faster output
n_step = 1000 # jump around for faster output
bid_range = 0:42000
estimates = Float64[]
n = n0

# initiate values to enter while loop
prev_estimate = (bid -> showcase(bid, n)).(bid_range)
current_estimate = (bid -> showcase(bid, n + 1)).(bid_range)
accuracy = mean((current_estimate .- prev_estimate) .^ 2) # MSE
push!(estimates, accuracy)
while accuracy > desired_accuracy
    n = n + n_step
    prev_estimate = (bid -> showcase(bid, n)).(bid_range)
    current_estimate = (bid -> showcase(bid, n + 1)).(bid_range)
    accuracy = mean((current_estimate .- prev_estimate) .^ 2) # MSE
    push!(estimates, accuracy)
end
```

In [ ]:

```julia
println("Using a sample size of ", n, " gives a mean-squared error of ",
    round(accuracy, digits=3))
println("This is a very low mean-squared error for this function. The low
error suggests that the function has converged and minimal noise is in
the graph.")
plot(n0:n_step:n, estimates,
    legend=:false,
    xlabel="Sample Size",
    ylabel="Estimate")
hline!([desired_accuracy], color=:red, linestyle=:dash)
xlims!(n0, n + n_step)
ylims!(desired_accuracy - 2, desired_accuracy * 10)
```

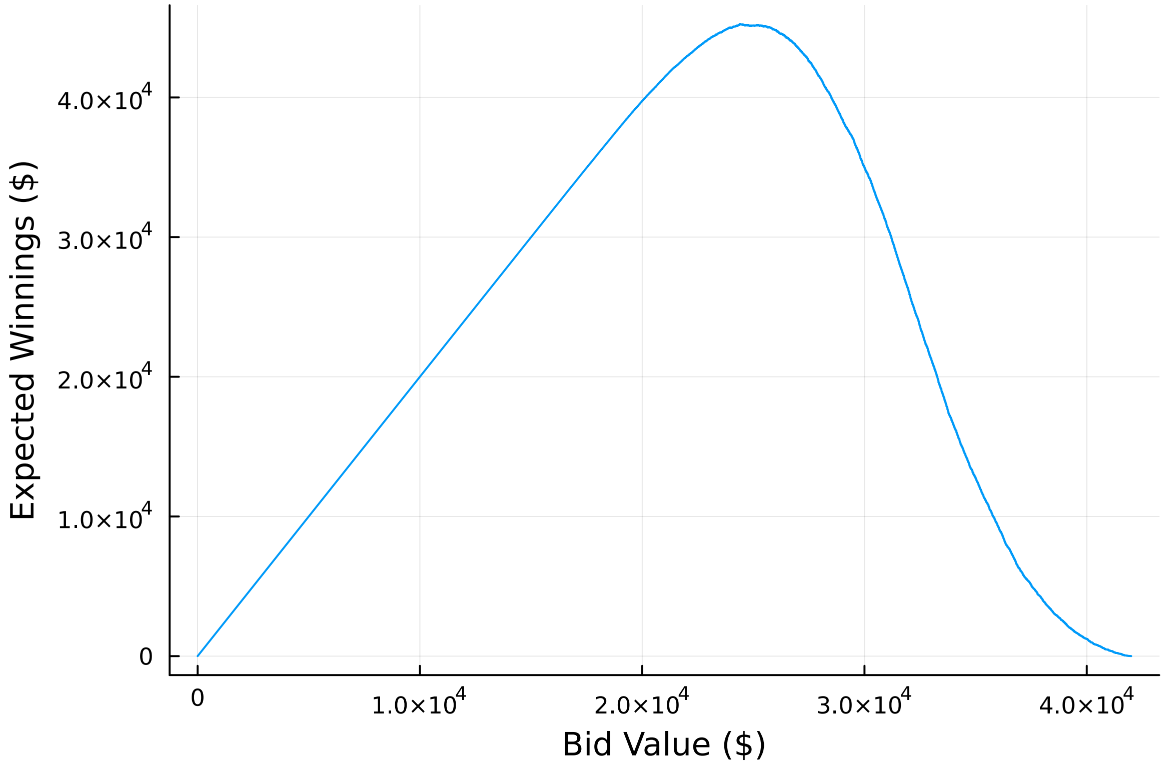
```
Using a sample size of 12000 gives a mean-squared error of 0.942
This is a very low mean-squared error for this function. The low
error suggests that the function has converged and minimal noise is in
the graph.
```



Problem 2.2 (2 points)

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

In [ ]:
```
bid_range = 0:42000
winnings = (bid -> showcase(bid, n)).(bid_range)
plot(bid_range, winnings,
    legend=:false,
    xlabel="Bid Value (\$)",
    ylabel="Expected Winnings (\$)")
```



I notice that for low bid values, the expected winnings has a linear relationship with the bid values. The expected winnings peak at approximately 25000 USD, which is relatively close to the mean value of the showcases (31000 USD). Interestingly, expected winnings at larger bid values diminish non-linearly. This makes sense because the larger the bid, the more likely overbidding will occur.

References

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

BEE 4750 9/15 Lecture "Probability and Monte Carlo Simulation" Slides

BEE 4750 9/20 Lecture "Monte Carlo, Formally" Slides