Monty Hall Problem (Variations)

Group 7

November 2023

Abstract

In this report we have described the history and mathematics behind the famous Monty Hall problem, describing where the problem came from, how it grew to be so well known, and how the counterintuitive solution can be reached through logic and maths. We then expanded on this original problem to see what would happen with a different number of cars and goats and to try and find a general solution. The result was the same as before, that switching is always the best choice. The next step was adding different prizes or penalties behind the doors, rather than simply cars in each one. This led to a similar result that switching is the best choice when the sum of the prizes and penalties is positive. The final variation we studied was back to the original game but with a second player. This changed things up a bit, but still led to the same conclusion that switching is the best choice. After this, we created a Python code to simulate these variations and test our results. This confirmed what we had previously calculated and provided some interesting statistics relating to the problems and specifically percentage error of the simulation dependency on various parameters of the test.

Contents

Chapter 1

Introduction

1.1 History

The concept behind the Monty Hall problem was originally introduced by Joseph Bertrand in his box paradox in 1889. However, It wasn't until 1990 that the modern version involving the game show host, Monty Hall, was popularised. The problem was posed to Marilyn vos Savant in her weekly column, "Ask Marilyn" in the Parade magazine. Marilyn, who was deemed to have the highest IQ in the world at the time, answered the problem correctly stating that you should always switch doors. This conclusion shocked many educated readers who retaliated with angry letters, calling Marilyn's IQ into question. Since then, this problem has become world-famous and is a common brain teaser used in media and college lectures alike. It is a seemingly obvious problem with an unintuitive solution that humans seem to struggle to solve correctly. In an experiment conducted in 2010, Walter Herbranson and Julia Schroeder from Whitman College conducted an experiment to investigate if playing the Monty Hall problem game multiple times would help refine the player's strategy. In this experiment, the humans were unable to identify and adopt the correct strategy of switching doors. However, when pigeons played the game multiple times using grain as a prize they were able to identify this winning strategy, unlike their human counterparts. This inability of humans to be able to solve this problem without calculation is what makes it so interesting.

1.2 Applications

The Monty Hall problem is a great way to learn about conditional probability, the chance of an event occurring based on another event. It shows how, when presented with new information your position may differ. The fact that the host must choose to show you a door containing a goat is what sparks the change in probability between your options. This problem has been used in movies and television shows to provide entertainment and insight into how humans think.

In Brooklyn Nine-Nine the problem showed how two people can be certain that their respective choice is correct, showing the difference between someone who thinks using their intuition versus someone who thinks rationally. The problem is a great way to show the importance of strategic and rational thinking. It is very prevalent in game theory and is often taught in schools and colleges. The problem is engaging and also very informative, teaching students the importance of critical thinking in a fun environment. The problem can also be used to evaluate an AI's ability to make decisions and use Bayesian reasoning. The research done on squirrels can similarly be applied to AI, having an AI simulate the problem multiple times will allow it to realize the correct strategy using machine learning.

1.3 Aims of this report

Since the original problem has been explored extensively, this report investigates how changing different aspects of the problem affects the player's strategy. To do this, different variations have been introduced in this report such as changing the number of doors, the values behind the doors, and the number of players. Generalizations of the solutions to these variations are discussed along with a simulation written in Python to prove them. For Python simulations dependencies of percentage error fluctuations on different parameters are researched.

Chapter 2

Original problem

The Monty Hall Problem has fairly simple rules. The player is introduced to three doors, behind two of which are goats, and behind one there is a car. The goal of a player is to win a car.

The player chooses a door, then the host opens a door with a goat behind it and the player has two options: to switch to another available door or to stick with their choice.

At first, it seems that the player's decision does not affect the result, because the player doesn't know where the car is. Contrary to this hypothesis, the truth is that switching always increases the player's chances to win a car. This becomes more apparent after some calculations.

2.1 Solution

When the player first picks a door there is a certain possibility that the player will pick the door with a goat behind it as well as the door with a car behind it. Let's call these possibilities P(G) for the probability of picking a door with a goat behind it and P(C) for picking a door with a car behind it. For the original problem:

$$P(G) = \frac{2}{3}$$

$$P(C) = \frac{1}{3}$$

Now, the host opens one of the doors that the player hasn't picked, this door has a goat behind it. Then, the player is offered a choice to stick with the door they already picked or to switch to another door.

Let's first calculate the probability of the player winning a car if they decide to stick with their choice. If they do, then the probability of winning is the same as the probability of the player picking the door with the car behind it from the beginning.

$$P(win) = P(C) = \frac{1}{3}$$

Now, let's look at the case where the player decides to switch the picket door. This probability will be equal to the sum of the probabilities that the player will win originally picking the door with the goat behind it and the probability that the player will win originally picking the door with the car behind it:

$$P(win) = P(G)P(win|originaldoorG) + P(C)P(win|originaldoorC)$$

The probability of winning if the original door picked had a goat behind it and the player decides to switch the door is 1 as the original door had a goat behind it and one of the doors with the goat behind it was opened. Hence, the player could only switch to the door with the car behind it.

The probability of winning if the original door picked had a car behind it and the player decides to switch the door is 0 as the player did not stick with their choice and, hence, did not pick the door with the car behind it.

Then the probability of winning if switching the door is:

$$P(win) = P(G)P(win|originaldoorG) + P(C)P(win|originaldoorC) =$$

$$= \frac{2}{3} \cdot 1 + \frac{1}{3} \cdot 0 = \frac{2}{3}$$

2.2 Answer

The probability of winning when switching the door is higher than the probability of winning when not switching the door. It is always the right decision to switch the door in the original problem.

Chapter 3

Variations

3.1 Generalization for $c+g$ doors and c cars behind them

This variation is a generalization of the original problem. In the original problem, there were 3 doors with 2 goats and 1 car behind them. In this variation, there are $c+g$ doors with $g(g \geq 2)$ goats and $c(c \geq 1)$ cars behind them.

3.1.1 Solution

To solve this problem we will use the same framework as for the original problem. We will calculate the probability of originally picking the door with the car behind it and the probability of originally picking the door with the goat behind it:

$$P(G) = \frac{g}{c+g}$$
$$P(C) = \frac{c}{c+g}$$

If the player decides not to switch, then the same way as in the original problem the probability of them winning stays:

$$P(win) = P(C) = \frac{c}{c+g}$$

If the player decides to switch then the probability of winning is calculated the same way as in the original problem:

$$P(win) = P(G)P(win|originaldoorG) + P(C)P(win|originaldoorC)$$

Taking into consideration that now there are c cars and $c+g$ doors:

$$P(win) = \frac{g}{c+g} \cdot \frac{c}{c+g-2} + \frac{c}{c+g} \cdot \frac{c-1}{c+g-2} =$$
$$= \frac{c^2+cg-c}{(c+g)(c+g-2)}$$

3.1.2 Answer

To find out whether the player should switch the door or not in this condition we need to compare the probabilities of winning.

$$P(win) - P(C) = \frac{c^2 + cg - c}{(c+g)(c+g-2)} - \frac{c}{c+g} =$$

$$= \frac{c^2 + cg - c}{(c+g)(c+g-2)} - \frac{c^2 + cg - 2c}{(c+g)(c+g-2)} =$$

$$= \frac{c}{(c+g)(c+g-2)}$$

And $\frac{c}{(c+g)(c+g-2)} > 0$ as $c \geq 1$ and $g \geq 2$.

This means that $P(win) > P(C)$ and, hence, it is always the right decision to switch. This variation shows that no matter how many doors there are, as long as the host opens the door with the goat, it is always the right decision to switch.

3.2 Inhomogeneous prizes

This variation is to discover how the outcomes of the problems will change depending on different prizes. Let's change the prize set to 0 euros behind two doors, 1 euro behind one door, and x euros behind another door. Altogether there are 4 doors.

3.2.1 Solution

To solve this problem we cannot use the exact template that we used to solve previous variations as in this template we used the main value that determined whether we should switch or not was the probability to win. But in this variation, it is; not that clear what is it to 'win' – It is possible to win 1 euro or x euros. Instead of calculating the probability of winning we will calculate expected winnings.

Let's calculate expected winnings if the player decides not to switch. With the same logic as in the previous variations, it will be just probabilities of picking the door with the prize behind it multiplied by the value of the prize:

$$Ex.W. = P(0) \cdot 0 + P(1) \cdot 1 + P(x) \cdot x =$$

$$= \frac{1}{2} \cdot 0 + \frac{1}{4} \cdot 1 + \frac{1}{4} \cdot x = \frac{1}{4}x + \frac{1}{4}$$

Now we will calculate the expected winning in the case if the player decides to switch the door. With the same logic as for previous variations, it will be the probability of picking a certain original door multiplied by the expected value of winnings in case this certain original door was picked and it was decided to switch when offered such a choice. It is important to note that it does not make a difference to which exact door the player decides to switch (as there 4 doors, hence, more than one option).

$$
\begin{gathered}
Ex.W.' = P(0)ex.w.(0) + P(1)ex.w.(1) + P(x)ex.w.(x) = \\
= \frac{1}{2} \cdot ex.w.(0) + \frac{1}{4} \cdot ex.w.(1) + \frac{1}{4} \cdot ex.w.(x) = \\
= \frac{1}{2}\left(\frac{1}{2} \cdot 1 + \frac{1}{2} \cdot x\right) + \frac{1}{4}\left(\frac{1}{2} \cdot 0 + \frac{1}{2} \cdot x\right) + \frac{1}{4}\left(\frac{1}{2} \cdot 0 + \frac{1}{2} \cdot 1\right) = \\
= 0.375x + 0.375
\end{gathered}
$$

3.2.2 Answer

These calculations show that as long as $x > -1$ it is always a good decision to switch, so unless x is a penalty, which is also a possibility for this variation, it is always the right decision to switch. Please also note that saying $x > -1$ is equivalent to saying $x + 1 > 0$ which means that the sum of all prizes needs to be positive. This trend will be noticeable in further variations.

3.3 Inhomogeneous prizes - generalization for four doors

This variation is a generalization of the previous variation. Now there are two doors with 0 euros behind them, 1 door with x euros behind it, and one door with y euros behind it.

3.3.1 Solution

Same as in variation 4.2, let's calculate the expected winning if the player decides not to switch.

$$
\begin{gathered}
Ex.W. = \frac{1}{2} \cdot 0 + \frac{1}{4} \cdot x + \frac{1}{4} \cdot y = \\
= \frac{1}{4}(x + y)
\end{gathered}
$$

Now let's calculate the expected winnings if the player decides to switch to another door when offered to do so. We will do it the same way as in variation 4.2.

$$
\begin{gathered}
Ex.W.' = P(0)ex.w.(0) + P(x)ex.w.(x) + P(y)ex.w.(y) = \\
= \frac{1}{2} \cdot ex.w.(0) + \frac{1}{4} \cdot ex.w.(x) + \frac{1}{4} \cdot ex.w.(y) = \\
= \frac{1}{2}\left(\frac{1}{2} \cdot x + \frac{1}{2} \cdot y\right) + \frac{1}{4}\left(\frac{1}{2} \cdot 0 + \frac{1}{2} \cdot y\right) + \frac{1}{4}\left(\frac{1}{2} \cdot 0 + \frac{1}{2} \cdot x\right) = \\
= 0.375(x + y)
\end{gathered}
$$

3.3.2 Answer

$Ex.W' > Ex.W$ as long as $x + y > 0$, which proves the conjecture for the previous problem: **If the sum of the prizes is positive, it is always a good decision to switch**.

3.4 Generalization for n doors and inhomogeneous prizes

This generalization is for $m(m \geq 2)$ doors with 0 euros behind them, and n doors with $n(n \geq 1)$ prizes $a_1, a_2, a_3 \ldots a_n$ behind them. This generalization will have more difficult calculations, however, the framework for the solution will be absolutely the same as in variations 4.2 and 4.3.

3.4.1 Solution

First, let's calculate the expected winnings if the player decides not to switch.

$$Ex.W. = \frac{m}{m+n} \cdot 0 + \frac{1}{m+n} \cdot a_1 + \frac{1}{m+n} \cdot a_2 + \ldots \frac{1}{m+n} \cdot a_n =$$

$$= \sum_{i=1}^{n} \frac{a_i}{m+n} = \frac{1}{m+n} \sum_{i=1}^{n} a_i$$

Let $\sum_{i=1}^{n} a_i = A$, then $Ex.W = \frac{A}{m+n}$

Now, let's calculate the expected winnings if the player decides to switch the door, the same way we calculated it in variations 4.3 and 4.2.

$$Ex.W' = P(0)ex.w.(0) + \sum_{i=1}^{n} P(a_i)ex.w.(a_i) =$$

$$= \frac{m}{m+n}\left(\frac{m-2}{m+n-2} \cdot 0 + \frac{1}{m+n-2} \sum_{i=1}^{n} a_i\right) + \frac{1}{m+n}\left(\frac{m-1}{m+n-2} \cdot 0 + \frac{1}{m+n-2} \sum_{i=2}^{n} a_i\right)$$

$$+ \frac{1}{m+n}\left(\frac{m-1}{m+n-2} \cdot 0 + \frac{1}{m+n-2}\left(\sum_{i=1}^{1} a_i + \sum_{i=3}^{n} a_i\right)\right) +$$

$$+ \frac{1}{m+n}\left(\frac{m-1}{m+n-2} \cdot 0 + \frac{1}{m+n-2}\left(\sum_{i=1}^{2} a_i + \sum_{i=4}^{n} a_i\right)\right) + \ldots$$

$$\ldots + \frac{1}{m+n}\left(\frac{m-1}{m+n-2} \cdot 0 + \frac{1}{m+n-2} \sum_{i=1}^{n-1} a_i\right)$$

Where, $A = \sum_{i=1}^{n} a_i$, hence,

$$\sum_{i=1}^{t} a_i + \sum_{i=t+1}^{n} a_i = A - a_t$$

Then,

$$\begin{aligned}
Ex.W' = \frac{m}{m+n} \cdot \frac{A}{m+n-2} + \frac{1}{m+n} \cdot \frac{A-a_1}{m+n-2} + \frac{1}{m+n} \cdot \frac{A-a_2}{m+n-2} + \\
+ \frac{1}{m+n} \cdot \frac{A-a_3}{m+n-2} + \ldots + \frac{1}{m+n} \cdot \frac{A-a_n}{m+n-2} = \\
= \frac{m}{m+n} \cdot \frac{A}{m+n-2} + \frac{1}{(m+n)(m+n-2)} \sum_{i=1}^{n} A - a_i = \\
= \frac{m}{m+n} \cdot \frac{A}{m+n-2} + \frac{nA - \sum_{i=1}^{n} a_i}{(m+n)(m+n-2)} = \\
= \frac{mA}{(m+n)(m+n-2)} + \frac{nA - A}{(m+n)(m+n-2)} = \frac{(m+n-1)A}{(m+n)(m+n-2)}
\end{aligned}$$

3.4.2 Answer

Note that $\frac{m+n-1}{m+n-2} > 1$, as $m \geq 2$ and $n \geq 1$, hence, $m+n \geq 3$, therefore, $m+n-1 > m+n-2 > 0$, hence, $\frac{m+n-1}{m+n-2} > 1$.

This means that if $A > 0$ then it is always the right decision to switch. This leads us to the same conjecture as other variations with inhomogeneous prizes.

3.4.3 Conjecture

Let $m+n$ be the number of doors, m of which are empty and n of which have something behind them. This something can be a prize or a penalty. Let the values of those prizes and penalties behind these doors be $a_1, a_2, \ldots, a_n$. Then, if

$$\sum_{i=1}^{n} a_i > 0$$

then it is always the right decision to switch.

3.5 Two players

In this variation, we will consider the version of the game with two players playing by the following rules[And09]. Three identical doors concealing two goats and one car. There are two players in the game. Each player randomly chooses one of the doors but does not open it. Each player knows there is another person in the game, but neither knows which door the other player selected. Monty now opens a door according to the following procedure:

1. If both players select the same door, then everything proceeds as in the classical game. Monty opens a goat-concealing door, choosing randomly if he has a choice.

2. If the players selected different doors, then Monty opens the one remaining door, regardless of what is behind it.

3.5.1 Solution

Assume player 1 chooses door 1, then Monty opens a goat concealing door 2, then there are three possible scenarios for Monty.

1. Player 2 chose door one, door one conceals the car, and Monty chose door two randomly.

2. Player 2 chose door one, door three concealed the car, and Monty was forced to open door two.

3. Player 2 chose door three, Monty was forced to open door two.

Now that we can say player 2 didn't choose door 2 based on Monty's door opening criteria. Now, it is more likely that player 2 chose door 3 over door 1. Consider these cases:

If player 2 chooses door three, then Monty is forced to open door two. It conceals a goat with a probability of $\frac{2}{3}$.

If Player 2 chose door one then there are two further cases to consider.

1. The car is behind door one with probability $\frac{1}{3}$. In this case, Monty opened door two randomly, which happens with probability $\frac{1}{2}$. It follows that scenario one above happens with probability $\frac{1}{6}$.

2. The car is behind door three with probability $\frac{1}{3}$. In this case, Monty is forced to open the goat-concealing door two. It follows that the probability that Player 2 chose door one is $\frac{1}{3} + \frac{1}{6} = \frac{1}{2}$.

The event in which Monty opens the goat-concealing door two after Player 1 chooses door one is more likely to occur when Player 2 has chosen door one than when he has chosen door three. Specifically, it is $\frac{4}{3}$ times more likely that Player 2 has chosen door three.

Therefore the probabilities we now assign to "Player 2 chose door three," and "Player 2 chose door one," must preserve this $4 : 3$ ratio. Consequently we assign probabilities of $\frac{4}{7}$ and $\frac{3}{7}$ in each case.

From Player 1's perspective, there are now four possibilities. Player 2 could have chosen door one or door three, and the car could be behind either of those doors. Let us denote these possibilities via ordered pairs of the form (Player 2's Door, Location of the Car):

The four possibilities are $(3, 1), (3, 3), (1, 1), (1, 3)$.

Consider the first two pairs. If Player 2 chose door three, then Monty was forced to open door two. Consequently, we learn nothing regarding the probability of doors one and three. Since these two scenarios collectively have a probability of $\frac{4}{7}$, and since they are equally likely, we now assign the following probabilities: $P(3,1) = P(3,3) = \frac{2}{7}$.

The remaining two pairs, however, are not equiprobable. Suppose that Player 2 chose door one, just as Player 1 did. If the car is behind door one then Monty chose door two randomly, which happens with probability $\frac{1}{2}$. If the car is behind door three then Monty is forced to choose door two. It follows that it is twice as likely that the car is behind door three than that it is behind door two. Since these scenarios have a collective probability of $\frac{3}{7}$, we assign the following probabilities: $P(1,1) = \frac{1}{7}$ and $P(1,3) = \frac{2}{7}$.

3.5.2 Answer

Of the four scenarios, the two in which Player 1 wins by switching are $(1,3)$ and $(3,3)$. Since both have probability $\frac{2}{7}$, this gives a total probability of winning by switching of $\frac{4}{7}$.

It is in the player's best interest to switch in this particular scenario of the Monty Hall Problem.

Chapter 4

Simulation

To prove the conjecture and probabilities it was decided to write a Python code that would simulate multiple trials of the game.

4.1 Algorithm

The algorithm that we programmed works as follows

1. The user is asked to input the number of doors with nothing behind them n.
2. The list of prizes is uploaded from the corresponding file *prizes.txt* located in the same directory as the program.
3. The list of all doors $0, 1, 2, \ldots, n+m$ and everything that is behind them $a_0, a_1, a_2, ..., a_{n+m}$ is created and printed out on the screen, where the index of the number in the list is the number of the door and the value at this index is the amount of money behind the door, the doors with no prizes go before the doors with prizes in the list but it does not affect the probability as the choice is randomized.
4. The user is asked to input the number of trials they want to run on a certain set of doors.
5. Using library *random.py* the random number from 0 to $n + m - 1$ is generated and this is the first choice of the player.
6. If the prize or penalty is behind the chosen door the amount of wins[1] without switching is increased and the sum of values of prizes and penalties won without switching is increased by the value of the prize or decreased by the value of penalty.
7. The door opened by the host is the first door unless the first door was the first choice of the player, then it is the second door, the number of the door that is opened by the host does not matter as the choice is randomized.

8. Then, the number to represent choice after switching is also randomized from 0 to $n + m - 1$ but if the number after the switch is equal to the door opened by the host or to the first choice, the randomization is redone using *random.py*.

9. If the prize is behind the second door chosen the wins[1] after switching count is increased and the sum of values of prizes and penalties won after switching is increased by the value of the prize or decreased by the value of penalty.

10. Then theoretical values are calculated. Probability of winning or choosing non-empty door[1] with and without switching and expected winnings for variations with inhomogeneous prizes, using following formulas[2]:

$$P(win) = \frac{m}{m+n}$$

$$P(win\ switching) = \frac{m(m-1)+mn}{(m+n)(m+n-2)}$$

$$Ex.W.(no\ switching) = \frac{1}{m+n}\sum_{i=1}^{n+m} a_i$$

$$Ex.W.(switching) = \frac{m+n-1}{(m+n)(m+n-2)}\sum_{i=1}^{n+m} a_i$$

11. The standard deviation and sum of the set of values behind the doors are calculated. Where standard deviations are calculated using the formula

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n+m}(x_i - \mu)}{N}}, \text{ where}$$

N - the number of doors
x_i - the value behind $i - th$ door
μ - mean calculated by the formula $\mu = \frac{\sum_{i=1}^{n+m} x_i}{N}$

12. The amount of wins[1] with and without switching is printed out on the screen

13. The average value of winnings before and after the switch is calculated, using the formula:

$$Av.W.(no\,switching) = \frac{value\ of\ prizes\ and\ penalties\ won\ no\ switching}{number\ of\ trials}$$

[1]By winning it is meant that the chosen door was non-empty. This defines win in simulations with goats and cars.

[2]In terms of names of variables in the program

$$Av.W.(switching) = \frac{value\ of\ prizes\ and\ penalties\ won\ switching}{number\ of\ trials}$$

And printed on the screen and compared to theoretically calculated expected winnings. Then the percentage error is calculated using the following formula:

$$Error = \left| \frac{Simulation - Theoretical}{Theoretical} \right| * 100\% \tag{4.1}$$

14. The percentage probability of wins[1] in each case is calculated as

$$P(wins\ switching) = \frac{wins\ switching}{number\ of\ trials}$$

$$P(wins\ no\ switching) = \frac{wins\ no\ switching}{number\ of\ trials}$$

and printed out on the screen as well as theoretically calculated probabilities of picking non-empty doors and the percentage error calculated with the formula 4.1.

4.2 Test generation programs

To create tests with hundreds and thousands of non-empty doors the program for generating tests was written. To research how the accuracy of the calculations in the simulation will change depending on different ways of generating a set of values behind non-empty doors.

4.2.1 Uniformly distributed randomised set

This generator uses simple randomization. Hence the values should be distributed uniformly on a population with a size much bigger than range. This generator uses *random.py*.

```python
import random as r

# Open a file to store the prizes
f = open("prizes.txt", "w")

# User input for the number of non-empty doors
n = int(input('Enter the amount of non-empty doors wanted: '))

# Generate random prizes for each non-empty door
prizes = []
for i in range(n - 1):
    element = r.randint(-1000, 1000)
    prizes.append(element)

# Write prizes to the file
for i in range(len(prizes) - 1):
    f.write(str(prizes[i]) + ",")
    print(str(prizes[i]) + ",", end="")

# Write the last prize
f.write(str(prizes[-1]))

# Close the file
f.close()

```

```python
# Print the last prize
print(str(prizes[-1]), end="\n")

```

4.2.2 Normally distributed randomised set

This variation. generates a set of numbers that are normally distributed on a set mean and a standard deviation. It uses *numpy.py*.

```python
import numpy as np

# User input for the number of non-empty doors, standard deviation, and mean
num_doors = int(input("Enter the amount of non-empty doors wanted: "))
std_deviation = float(input("Enter the standard deviation wanted: "))
mean_value = float(input("Enter the mean of the distribution wanted: "))

# Generate random numbers following a normal distribution
random_numbers = np.random.normal(loc=mean_value, scale=std_deviation, size=num_doors)

# Open a file to store the generated random numbers
with open("prizes.txt", "w") as file:
    # Write the random numbers to the file
    for i in range(len(random_numbers) - 1):
        file.write(str(random_numbers[i]) + ",")
        print(random_numbers[i], end=",")

    # Write the last random number
    file.write(str(random_numbers[-1]))
    print(random_numbers[-1])

```

4.2.3 Uniformly distributed set

This is the only non-random generator used for testing. For a set of n doors, it generates value from 0 to $\frac{n}{2}$ each of value repeated twice in a set.

```python
# User input for the number of non-empty doors
num_doors = int(input("Enter the amount of non-empty doors wanted: "))

# Generate a list of random numbers in a pattern (i + 1, i + 1)
random_numbers = []
for i in range(int(num_doors / 2)):
    random_numbers.append(i + 1)
    random_numbers.append(i + 1)

# Open a file to store the generated random numbers
with open("prizes.txt", "w") as file:
    # Write the random numbers to the file and print them
    for i in range(len(random_numbers) - 1):
        file.write(str(random_numbers[i]) + ",")
        print(random_numbers[i], end=",")

    # Write the last random number and print it
    file.write(str(random_numbers[-1]))
    print(random_numbers[-1])

```

4.3 Code of the simulation program

The code of the program is following the Algorithm described in the section 4.1

```python
import random as r
import math as mt

# User input for the number of doors with nothing behind them
n = int(input('Enter the number of doors with nothing behind them: '))

# Reading the list of prizes from a file
print('Prizes list will be read from the prepared file...')
f = open("prizes.txt", "r")
```

```python
af = str(f.readline())
a = af.split(',')
m = len(a)
for i in range(m):
    a[i] = int(a[i])

# Creating a list of doors, with n doors having nothing behind them and m doors with prizes
doors = [0 for i in range(n)]
doors += a
print(doors)

# Initializing variables to count wins with and without switching
wins = 0
wins_s = 0
average_winnings = 0
average_winnings_s = 0

# User input for the number of trials
trials = int(input("Enter the number of trials wanted: "))

# Simulating the specified number of trials
for iter in range(trials):
    # Making an initial choice
    choice = r.randint(0, m + n - 1)

    # Checking if the chosen door has a prize
    if doors[choice] != 0:
        wins += 1
        average_winnings += doors[choice]

    # Switching doors
    l = 0
    if choice == 0:
        l = 1
    choice_switch = r.randint(0, m + n - 1)

    # Host opened the first door with nothing (a goat) behind it
    while choice_switch == choice or choice_switch == l:
        choice_switch = r.randint(0, m + n - 1)

    # Checking if the switched door has a prize
    if doors[choice_switch] != 0:
        wins_s += 1
        average_winnings_s += doors[choice_switch]

# Calculating expected winnings using the formula
total_sum = sum(doors)
ex_w = float(total_sum) / float(m + n)
ex_s_w = float((m + n - 1) * total_sum) / float((m + n) * (m + n - 2))

# Calculating probability to open a non-empty door (win in variation with goats and cars)
p_win = float(m) / float(m + n)
p_s_win = float(m * (m - 1) + m * n) / float((m + n) * (m + n - 2))

# Calculating standard deviation
mean = float(total_sum) / float(len(doors))
sd = mt.sqrt(sum((float(i) - mean) ** 2 for i in doors) / float(len(doors)))
print(f"\n\tStandard deviation: {sd}")

# Printing the results
print(f"\nSum of all prizes is {total_sum}")
print(f"\nNumber of times a non-empty door was picked if decided not to switch: {wins}")
print(f"Number of times a non-empty door was picked if decided to switch: {wins_s}\n")

print(f"\nAverage winnings if decided not to switch: {float(average_winnings) / float(trials)} compared to theoretical expected winnings of {ex_w}")
print(f"\tHence the percentage error is {abs(((float(average_winnings) / float(trials)) - ex_w) / ex_w) * 100}%")
print(f"Average winnings if decided to switch: {float(average_winnings_s) / float(trials)} compared to theoretical expected winnings of {ex_s_w}")
print(f"\tHence the percentage error is {abs(((float(average_winnings_s) / float(trials)) - ex_s_w) / ex_s_w) * 100}\%\n")

# Calculating and printing the win ratio without switching
ratio = float(wins) / float(trials)
print(f"If deciding not to switch, the percentage of picking a non-empty door is {ratio * 100}\% compared to theoretical percentage of {p_win * 100}%")
print(f"\tHence the percentage error is {abs((ratio - p_win) / p_win) * 100}%")

# Calculating and printing the win ratio with switching
ratio = float(wins_s) / float(trials)
print(f"If deciding to switch, the percentage of picking a non-empty door is {ratio * 100}\% compared to theoretical percentage of {p_s_win * 100}%")
print(f"\tHence the percentage error is {abs((ratio - p_s_win) / p_s_win) * 100}%")

# Closing the file
f.close()

```

4.4 Tests

With these test generation programs, a series of tests was carried out. Two types of tests were carried out.

4.4.1 Conjecture testing

For this testing a function version of Simulator[3] and generators[4] were used as well as a driver program[5].

The results are presented in a table. For each simulation, 10^6 trials were carried out. Thus, considering that with each generator 100 different populations[6] including different sizes were tested the number of trials behind each of the percentages is 10^8. For each trial, the number of empty doors was 2. The percentages in the table represent the portion of tests that confirmed the conjecture 3.4.3. Raw logs of the testing are presented in Appendix C.

Number of the test	Range of the Population Size	Uniformly distributed set	Uniformly distributed randomised set	Normally distributed randomised set
1	100 - 100100(step: 10^4)	57%	55%	57%
2	100 - 100100(step: 10^4)	48%	62%	58%
3	100 - 100100(step: 10^4)	59%	46%	49%
4	1000 - 1001000(step: 10^5)	59%	45%	51%
5	1000 - 1001000(step: 10^5)	60%	53%	48%
6	1000 - 1001000(step: 10^5)	62%	43%	50%
7	1000 - 1001000(step: 10^5)	62%	47%	47%

Table 4.1: Table of conjecture testing.

As seen from 4.1 the results on the non-random set are almost always in favor of confirming conjecture 3.4.3. With random tests, the error prevents the confirmation of conjecture 3.4.3. There is a noticeable trend with randomly generated sets. That the bigger the test is the less consistent the conjecture testing is. This topic is further discussed in section 4.4.2.

Generally, the testing showed that the conjecture holds.

4.4.2 Percentage error testing

The percentage error testing was carried out with a different version of the driver program[7] and the same function for generating tests and a slightly differ-

[3] Appendix A.2
[4] Appendix A.1.2, A.1.1, A.1.3
[5] Appendix A.3
[6] By Population a set of prizes behind non-empty doors is meant
[7] Appendix B.2

ent simulation program[8]. Then the graphs were built using a scatter building website[sta]. All the graphs discussed further are presented in Appendix D[9].

Uniformly distributed randomised set

Three tests were carried out, with 10 sets of different sizes for each set the simulation was carried out 10 times with 10^6 trials each. The following parameters were recorded:

1. The percentage error compared to the theoretical value of average winnings when not switching.
2. The percentage error compared to the theoretical value of average winnings when switching.
3. The size of the population
4. The total sum of all values in the population
5. The standard deviation of the population

The graphs were built and showed the following results.

- With graphs of the population size against the percentage error the line of best fit showed the trend that the bigger the population is the higher the percentage error is.
- With the graphs of the total sum of the population against the percentage error the moving average showed a big increase when the sum is equal to 0 which leads to the conclusion that if the total value of penalties and the total value of prizes is equal in their absolute values, the theoretical calculations are less accurate.
- Graphs plotting the standard deviation against the percentage error showed higher average percentage error values in the places of concentration of scatter plot points. It was affected by the outliers. The presence of the outliers is a natural behavior for a simulation based on the randomization.

Normally distributed randomized set

Two tests were carried out for this generator with the same parameters recorded as in the section 4.4.2. And the same tendencies as in the results of section 4.4.2.

[8]Appendix B.1

[9]P.S. Number 6 was accidentally skipped during test numbering

Uniformly distributed set

Two tests were carried out for this generator with the same parameters recorded as in the section 4.4.2. Upon observing the graphs it was noticed that the standard deviation and the total sum of the population were proportional to the size of the set. So the graphs were only built of Population Sum against percentage error.

These tests showed extremely low average percentage error compared to randomly generated tests. The line of best fit showed that the accuracy was decreasing with the increase of the population size on some tests and that accuracy was increasing with the increase of the population size. In both cases, the slope of the line of best fit was very low, which leads to the conclusion that the accuracy on average stays the same with the increase in the size of the population.

Chapter 5

Conclusion

The Monty Hall problem demonstrates the counter-intuitive nature of probability and challenges our understanding of the probability used in decision-making. Our investigation into the Monty Hall problem shows the importance of relying on mathematical principles rather than intuition in probability-based decision-making. Through our analysis of the problem and its many variations, we have shown the complex nature of the problem, showing that switching doors statistically increases the likelihood of success. To sum up, a study of the Monty Hall problem offers a wealth of opportunities to investigate the complex interactions among decision science, psychology, and mathematics. By removing the layers from this seemingly straightforward probability puzzle, scientists can learn important things about human cognition, biases in decision-making, and the wider ramifications for disciplines outside of mathematics.

Chapter 6

Reflection

Our research and exploration of the Monty Problem have allowed us to understand the complexities of decision-based probability. The counter-intuitive solution that confused many has various implications for real-world scenarios. From decision-makers in fields such as finance to game theory and risk management, the Monty Hall problem demonstrates the importance of not relying on intuition alone in situations that involve probability. This problem goes far beyond a game show and applications of the Monty Hall problem can be seen throughout our daily lives. This seemingly simple problem allows us to learn valuable lessons about strategic decision-making, risk assessment, and the importance of effectively communicating probability concepts.

Bibliography

[And09] Stephen Lucas Andrew Schepler Jason Rosenhouse. “The Monty Hall Problem, Reconsidered”. In: *Mathematics Magazine* 82 (2009), pp. 332–342.

[sta] statschartz@gmail.com. *Website for making scatter plots*. `https://statscharts.com/scatter/scatterchart?status=edit`.

Appendices

Appendix A

Code for conjecture testing

A.1 Code for test generators

A.1.1 Uniformly distributed randomised set

```python
import random as r
def gen_unif_rand(n, low, high):
    # Open a file to store the prizes
    f = open("prizes.txt", "w")

    # Generate random prizes for each non-empty door
    prizes = []
    for i in range(n - 1):
        element = r.randint(low, high)
        prizes.append(element)

    # Write prizes to the file
    for i in range(len(prizes)):
        f.write(str(prizes[i]) + "\n")

    # Close the file
    f.close()
```

A.1.2 Normally distributed randomised set

```python
import numpy as np

def gen_norm_test(num_samples):
    """
    Generates a set of random numbers from a normal distribution for Monty Hall simulation.

    Parameters:
    - num_samples (int): Number of random numbers to generate.

    Returns:
    - np.array: Array of random numbers.
    """
    # Parameters for normal distribution
    mean = 0
    standard_deviation = 500

    # Generate random numbers from a normal distribution
    random_numbers = np.random.normal(loc=mean, scale=standard_deviation, size=num_samples)

    return random_numbers

def save_to_file(data, filename="prizes.txt"):
    """
    Saves a set of data to a file.

    Parameters:
    - data (iterable): Data to be saved.
    - filename (str): Name of the file to save data. Default is "prizes.txt".
    """
    with open(filename, "w") as file:
        for item in data:
            file.write(str(item) + "\n")

# Number of samples for Monty Hall simulation
num_samples = 1000

# Generate random numbers
random_numbers = generate_normal_distribution(num_samples)

# Save the generated numbers to a file
save_to_file(random_numbers)

```

A.1.3 Uniformly distributed set

```python
def gen_range_test(n):
    """
    Generates a set of numbers for non-empty doors in a Monty Hall simulation.

    Parameters:
    - n (int): Number of non-empty doors wanted.

    Returns:
    - list: List of door numbers.
    """
    door_numbers = []

    # Populate the list with pairs of door numbers
    for i in range(int(n / 2)):
        door_numbers.append(i + 1)
        door_numbers.append(i + 1)

    return door_numbers

def save_door_numbers_to_file(door_numbers, filename="prizes.txt"):
    """
    Saves a set of door numbers to a file.

    Parameters:
    - door_numbers (iterable): List of door numbers.
    - filename (str): Name of the file to save door numbers. Default is "prizes.txt".
    """
    with open(filename, "w") as file:
        for number in door_numbers:
            file.write(str(number) + "\n")

# Number of non-empty doors for Monty Hall simulation
num_doors = int(input("Enter the amount of non-empty doors wanted: "))

# Generate door numbers
door_numbers = generate_door_numbers(num_doors)

# Save the generated door numbers to a file
save_door_numbers_to_file(door_numbers)

```

A.2 Simulation version

```python
import random as r
import math as mt

def Sim(n, trials):
    # Reading the list of prizes from a file
    f = open("prizes.txt", "r")
    prizes = [float(line) for line in f.readlines()]
    m = len(prizes)

    # Creating a list of doors, with n doors having nothing behind them and m doors with prizes
    doors = [0 for _ in range(n)]
    doors += prizes

    # Initializing variables to count wins with and without switching
    wins = 0
    wins_s = 0
    average_winnings = 0
    average_winnings_s = 0

    # Running the simulation for the specified number of trials
    for _ in range(trials):
        # Making an initial choice
        choice = r.randint(0, m + n - 1)

        # Checking if the chosen door has a prize
        if doors[choice] != 0:
            wins += 1
            average_winnings += doors[choice]

        # Switching doors
        l = 0
        if choice == 0:
            l = 1
        choice_switch = r.randint(0, m + n - 1)

        # Host opened the first door with nothing (a goat) behind it
        while choice_switch == choice or choice_switch == l:
            choice_switch = r.randint(0, m + n - 1)

        # Checking if the switched door has a prize
        if doors[choice_switch] != 0:
            wins_s += 1
            average_winnings_s += doors[choice_switch]

    # Calculating expected winnings using the formula
    total_sum = sum(doors)
    ex_w = total_sum / (m + n)
    ex_s_w = ((m + n - 1) * total_sum) / ((m + n) * (m + n - 2))

    # Calculating probability to open non-empty door (win in variation with goats and cars)
    p_win = m / (m + n)
    p_s_win = (m * (m - 1) + m * n) / ((m + n) * (m + n - 2))

    # Standard deviation
    mean = total_sum / len(doors)
    sd = mt.sqrt(sum((i - mean) ** 2 for i in doors) / len(doors))

    # Returning relevant results
    return [(average_winnings / trials), (average_winnings_s / trials), total_sum]

```

A.3 Driver program

```python
import random as r
import numpy as nmp
import math as mt

from SCI10010gentest import gen_unif_rand
from SCI10010gennormaltest import gen_norm_test
from SCI10010genrangetest import gen_range_test
from SCI10010MontyHallSim import Sim

# n = int(input("Aount of prizes: "))
# m = int(input("Amount of empty doors: "))
# t = int(input("Amount of trials: "))
# low = int(input('Lower: '))
# high = int(input('Higher: '))

low = -1000
high = 1000
t = 1000000
m = 2

sample = 10
# n_arr = [10, 100, 1000, 10000, 50000, 100000, 1000000]
n_arr = range(100, 100100, 10000)

res = []
conj = [[0, 0], [0, 0], [0, 0]]
av_pererr = [0, 0]

count = 0

for i in n_arr:
    for j in range(sample):
        gen_range_test(i)
        res = Sim(m, t)
        count += 1
        print(f"{count}/{len(n_arr) * sample * 3}")
        if ((res[0] < res[1] and res[2] >= 0) or (res[0] > res[1] and res[2] <= 0)):
            conj[0][0] += 1
        else:
            conj[0][1] += 1

        gen_unif_rand(i, low, high)
        res = Sim(m, t)
        count += 1
        print(f"{count}/{len(n_arr) * sample * 3}")
        if ((res[0] < res[1] and res[2] >= 0) or (res[0] > res[1] and res[2] <= 0)):
            conj[1][0] += 1
        else:
            conj[1][1] += 1

        gen_norm_test(i)
        res = Sim(m, t)
        count += 1
        print(f"{count}/{len(n_arr) * sample * 3}")
        if ((res[0] < res[1] and res[2] >= 0) or (res[0] > res[1] and res[2] <= 0)):
            conj[2][0] += 1
        else:
            conj[2][1] += 1


print("\n", conj)
print(f"\nRange test: {(float(conj[0][0])/float(conj[0][0] + conj[0][1])) * 100}% confirming \t {(float(conj[0][1])/float(conj[0][0] + conj[0][1])) * 100}% disproving \n")
print(f"Uniform randomisation test: {(float(conj[1][0])/float(conj[1][0] + conj[1][1])) * 100}% confirming \t {(float(conj[1][1])/float(conj[1][0] + conj[1][1])) * 100}% disproving \n")
print(f"Normal distribution randomisation test: {(float(conj[2][0])/float(conj[2][0] + conj[2][1])) * 100}% confirming \t {(float(conj[2][1])/float(conj[2][0] + conj[2][1])) * 100}% disproving \n")
```

Appendix B

Code for percentage error tests

B.1 Simulation program

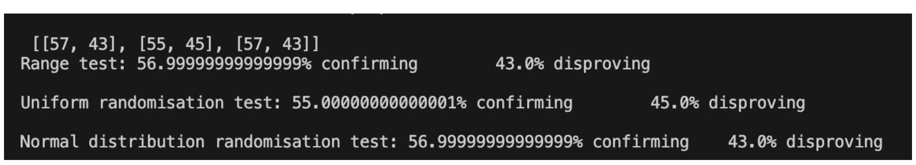
```python
import random as r
import math as mt

def Sim(n, trials):
    # Reading the list of prizes from a file
    f = open("prizes.txt", "r")
    prizes = [float(line) for line in f.readlines()]
    m = len(prizes)

    # Creating a list of doors, with n doors having nothing behind them and m doors with prizes
    doors = [0 for _ in range(n)]
    doors += prizes

    # Initializing variables to count wins with and without switching
    wins = 0
    wins_s = 0
    average_winnings = 0
    average_winnings_s = 0

    # Running the simulation for the specified number of trials
    for _ in range(trials):
        # Making an initial choice
        choice = r.randint(0, m + n - 1)

        # Checking if the chosen door has a prize
        if doors[choice] != 0:
            wins += 1
            average_winnings += doors[choice]

        # Switching doors
        l = 0
        if choice == 0:
            l = 1
        choice_switch = r.randint(0, m + n - 1)

        # Host opened the first door with nothing (a goat) behind it
        while choice_switch == choice or choice_switch == l:
            choice_switch = r.randint(0, m + n - 1)

        # Checking if the switched door has a prize
        if doors[choice_switch] != 0:
            wins_s += 1
            average_winnings_s += doors[choice_switch]

    # Calculating expected winnings using the formula
    total_sum = sum(doors)
    ex_w = total_sum / (m + n)
    ex_s_w = ((m + n - 1) * total_sum) / ((m + n) * (m + n - 2))

    # Calculating probability to open a non-empty door (win in variation with goats and cars)
    p_win = m / (m + n)
    p_s_win = (m * (m - 1) + m * n) / ((m + n) * (m + n - 2))

    # Standard deviation
    mean = total_sum / len(doors)
    sd = mt.sqrt(sum((i - mean) ** 2 for i in doors) / len(doors))

    # Calculating percentage error for switching and not switching
    error_switch = abs((wins_s / trials - p_s_win) / p_s_win) * 100
    error_no_switch = abs((wins / trials - p_win) / p_win) * 100

    # Closing the file
    f.close()

    # Returning relevant results
    return [total_sum, error_no_switch, error_switch, sd, total_sum]

```

B.2 Driver program

```python
import random as r
import numpy as nmp
import math as mt

from SCI10010gentest import gen_unif_rand
from SCI10010gennormaltest import gen_norm_test
from SCI10010genrangetest import gen_range_test
from SCI10010MontyHallSim import Sim

low = -1000
high = 1000
t = 1000000
m = 2

sample = 10

# n_arr = [10, 100, 1000, 10000, 50000, 100000, 1000000]
n_arr = range(1000, 1001000, 100000)

res = []
pererr = []
av_pererr = [0, 0]

count = 1

for i in n_arr:
    for j in range(sample):
        gen_range_test(i)
        pererr = Sim(m, t)
        print(f"{count}/{len(n_arr) * sample}")
        count += 1
        av_pererr[0] += pererr[0]
        av_pererr[1] += pererr[1]
        res.append([i, pererr[0], pererr[1], pererr[2], pererr[3]])
    av_pererr[0] /= sample
    av_pererr[1] /= sample


f1 = open("PopulationSize.txt", "w")
f2 = open("PerErrNoS.txt", "w")
f3 = open("PerErrS.txt", "w")
f4 = open("PopulationStandardDeviation.txt", "w")
f5 = open("PopilationSum.txt", "w")
for i in res:
    f1.write(str(i[0]) + "\n")
    f2.write(str(i[1]) + "\n")
    f3.write(str(i[2]) + "\n")
    f4.write(str(i[3]) + "\n")
    f5.write(str(i[4]) + "\n")
    # print(f'\n\n\nPopulation size:{i[0]} \n \t Percentage error no switch:{i[1]} \n \t Percentage error switch:{i[2]}')
```

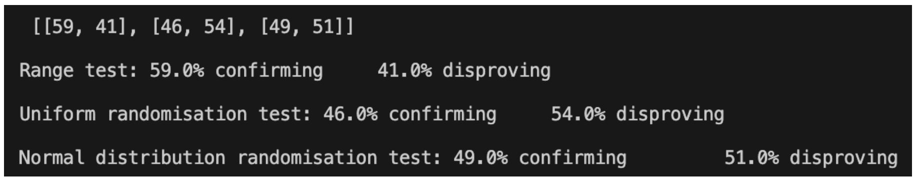
Appendix C

Logs from the conjecture test



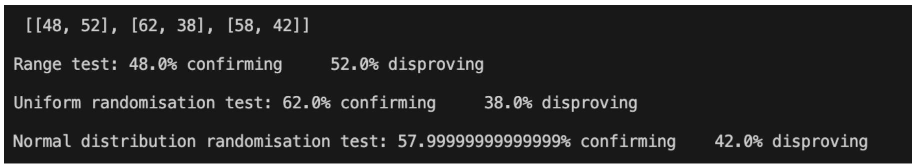
```
 [[57, 43], [55, 45], [57, 43]]
Range test: 56.99999999999999% confirming        43.0% disproving

Uniform randomisation test: 55.00000000000001% confirming        45.0% disproving

Normal distribution randomisation test: 56.99999999999999% confirming    43.0% disproving
```

Figure C.1: Test 1



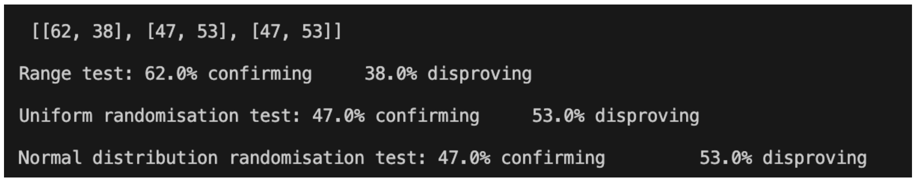
```
 [[59, 41], [46, 54], [49, 51]]

Range test: 59.0% confirming     41.0% disproving

Uniform randomisation test: 46.0% confirming     54.0% disproving

Normal distribution randomisation test: 49.0% confirming        51.0% disproving
```

Figure C.2: Test 2



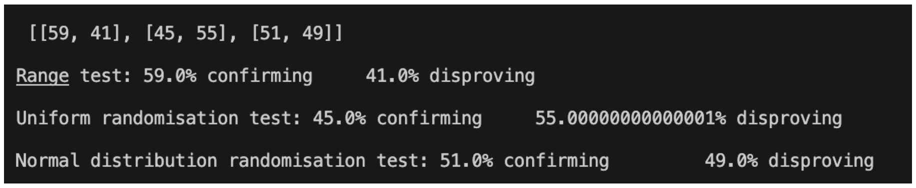
```
 [[48, 52], [62, 38], [58, 42]]

Range test: 48.0% confirming     52.0% disproving

Uniform randomisation test: 62.0% confirming     38.0% disproving

Normal distribution randomisation test: 57.99999999999999% confirming    42.0% disproving
```

Figure C.3: Test 3



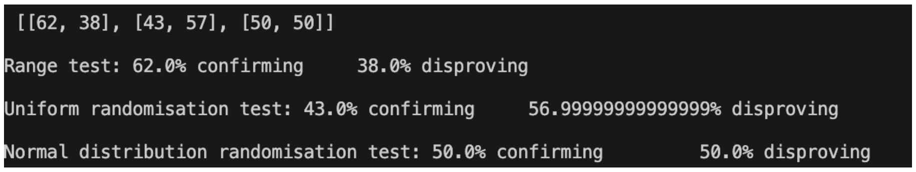
```
 [[62, 38], [47, 53], [47, 53]]
Range test: 62.0% confirming     38.0% disproving
Uniform randomisation test: 47.0% confirming     53.0% disproving
Normal distribution randomisation test: 47.0% confirming        53.0% disproving
```

Figure C.4: Test 4



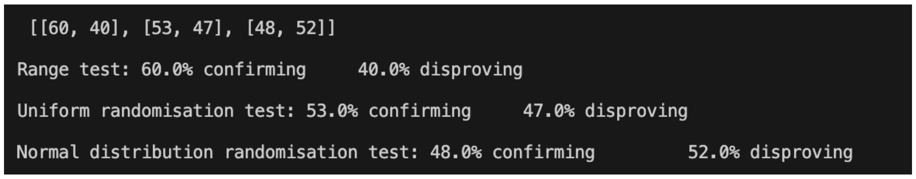
```
 [[59, 41], [45, 55], [51, 49]]
Range test: 59.0% confirming     41.0% disproving
Uniform randomisation test: 45.0% confirming     55.00000000000001% disproving
Normal distribution randomisation test: 51.0% confirming        49.0% disproving
```

Figure C.5: Test 5



```
 [[62, 38], [43, 57], [50, 50]]
Range test: 62.0% confirming     38.0% disproving
Uniform randomisation test: 43.0% confirming     56.99999999999999% disproving
Normal distribution randomisation test: 50.0% confirming        50.0% disproving
```

Figure C.6: Test 6



```
 [[60, 40], [53, 47], [48, 52]]
Range test: 60.0% confirming     40.0% disproving
Uniform randomisation test: 53.0% confirming     47.0% disproving
Normal distribution randomisation test: 48.0% confirming        52.0% disproving
```

Figure C.7: Test 7

Appendix D

Graphs for tendencies research

D.1 Population Size - Uniformly distributed randomized set

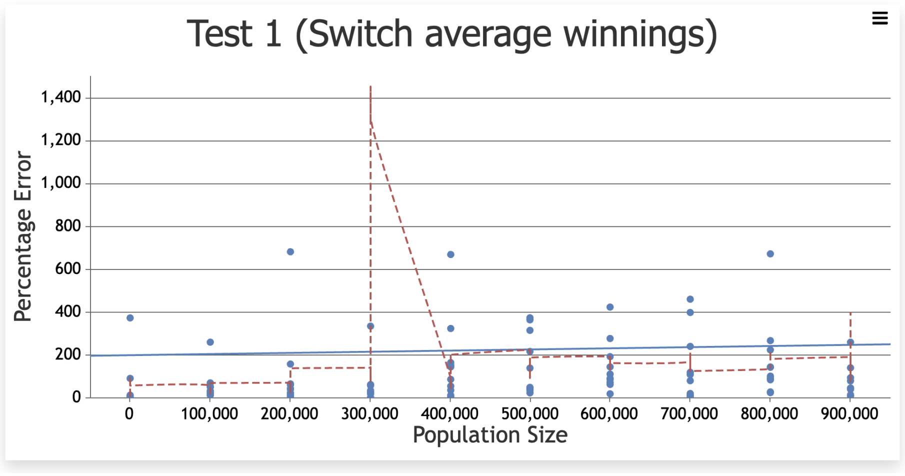


Figure D.1: Test 1 Switching

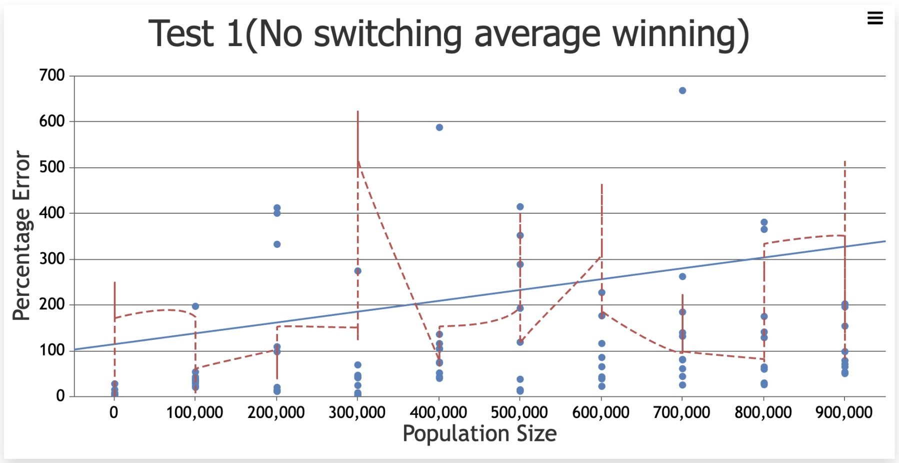


Figure D.2: Test 1 No switching

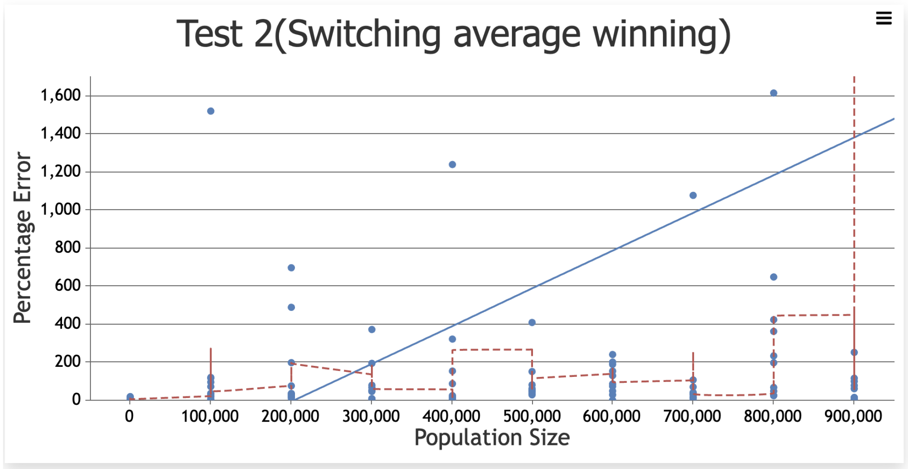


Figure D.3: Test 2 Switching

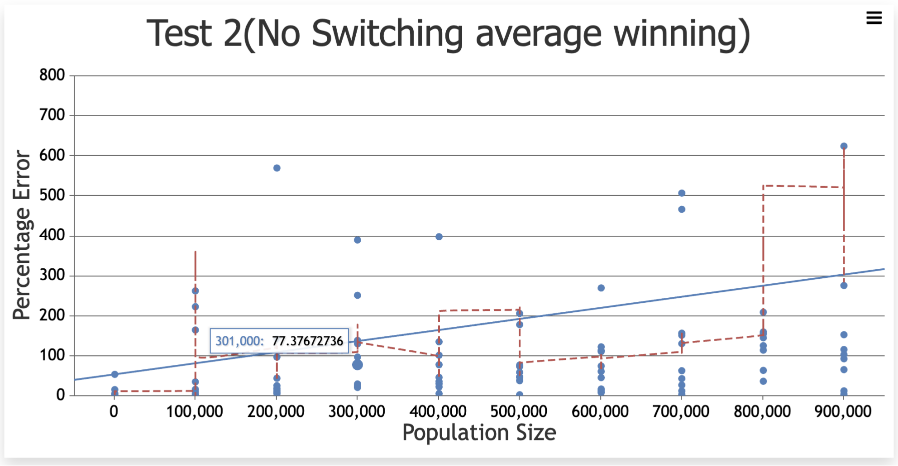


Figure D.4: Test 2 No switching

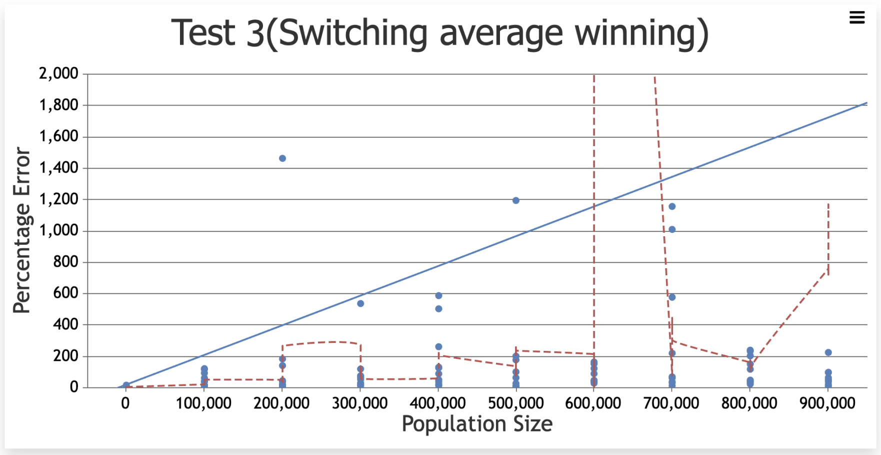


Figure D.5: Test 3 Switching

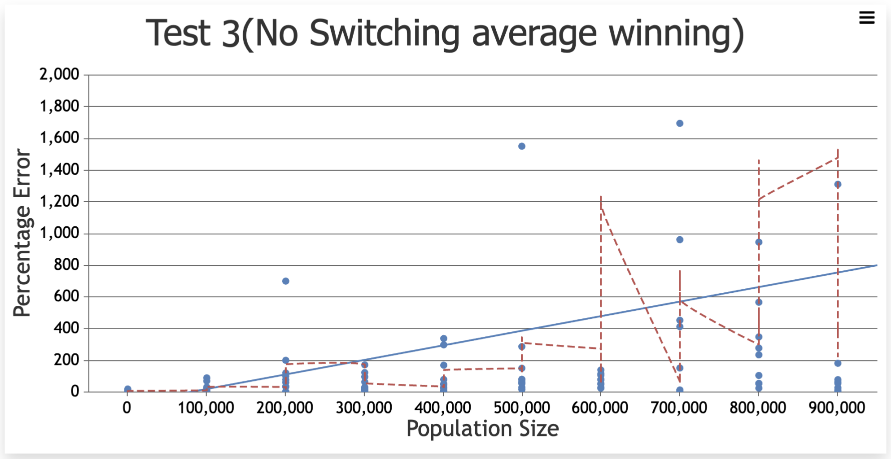


Figure D.6: Test 3 No switching

D.2 Total Sum of the Population - Uniformly distributed randomized set

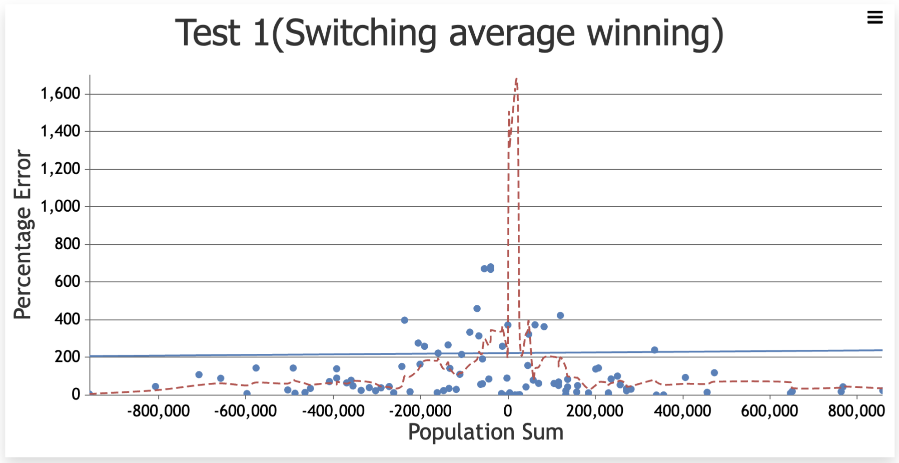


Figure D.1: Test 1 Switching

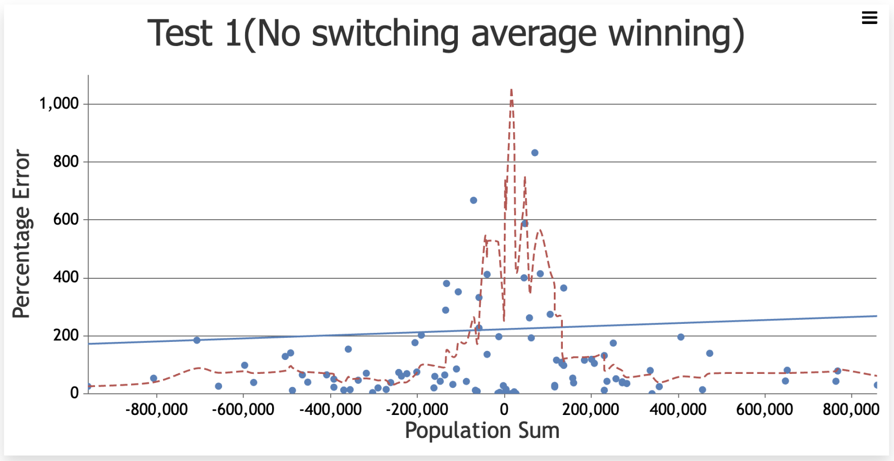


Figure D.2: Test 1 No switching

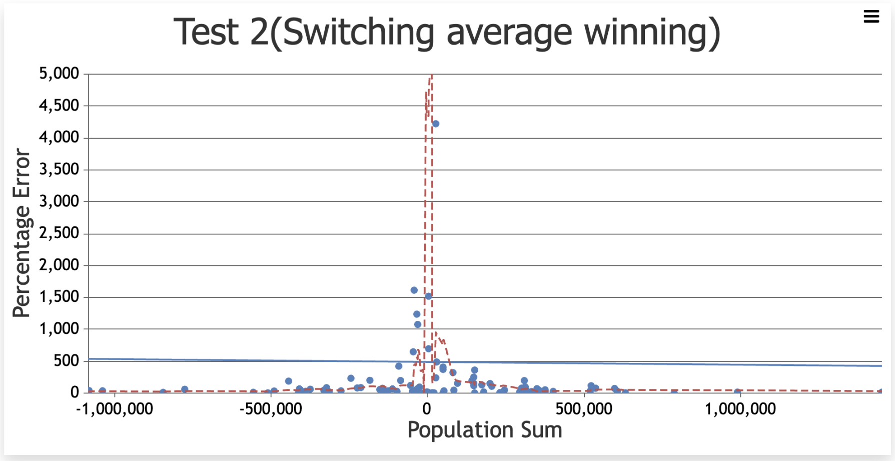


Figure D.3: Test 2 Switching

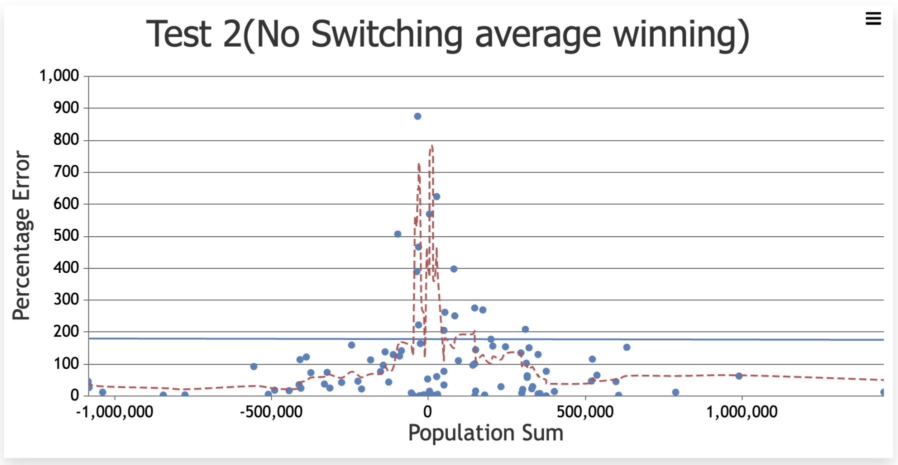


Figure D.4: Test 2 No switching

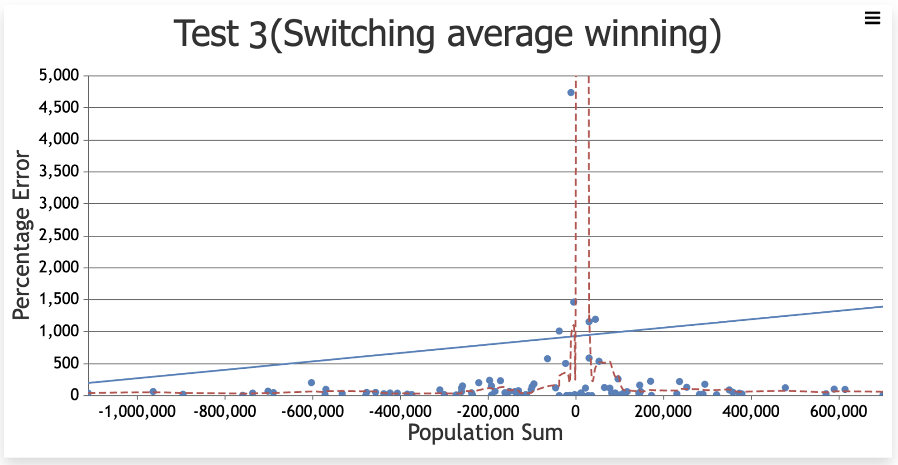


Figure D.5: Test 3 Switching

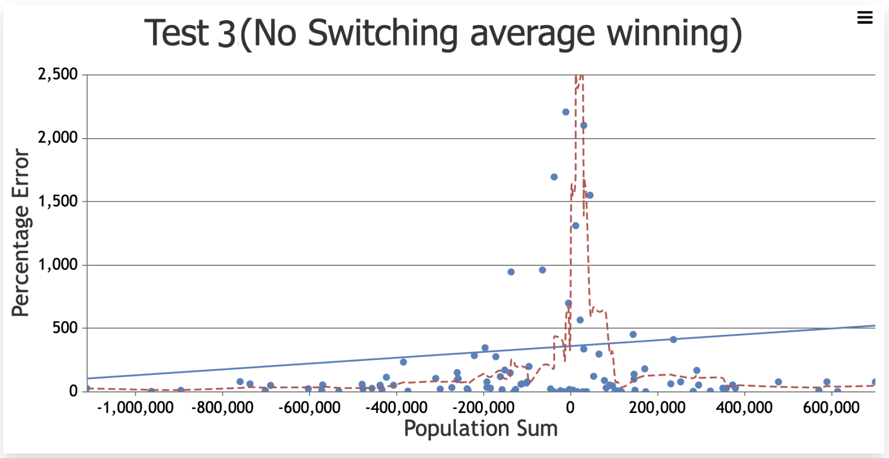


Figure D.6: Test 3 No switching

D.3 Standard Deviation - Uniformly distributed randomized set

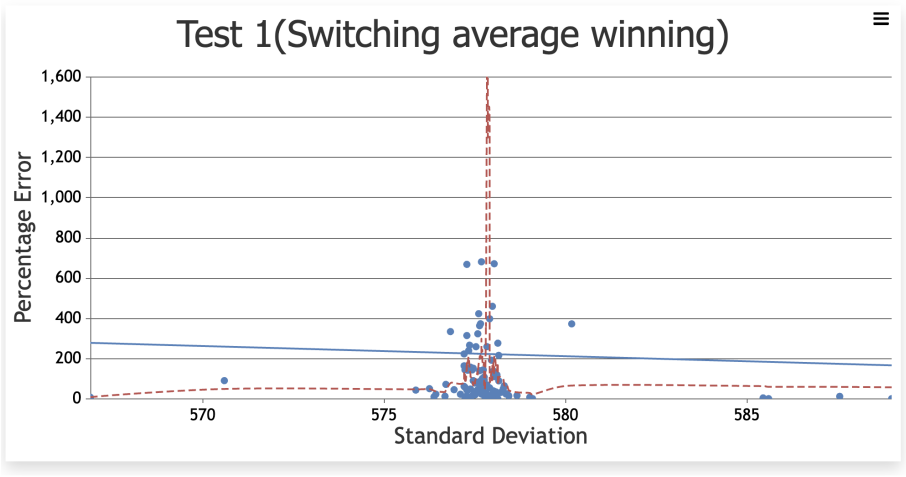


Figure D.1: Test 1 Switching

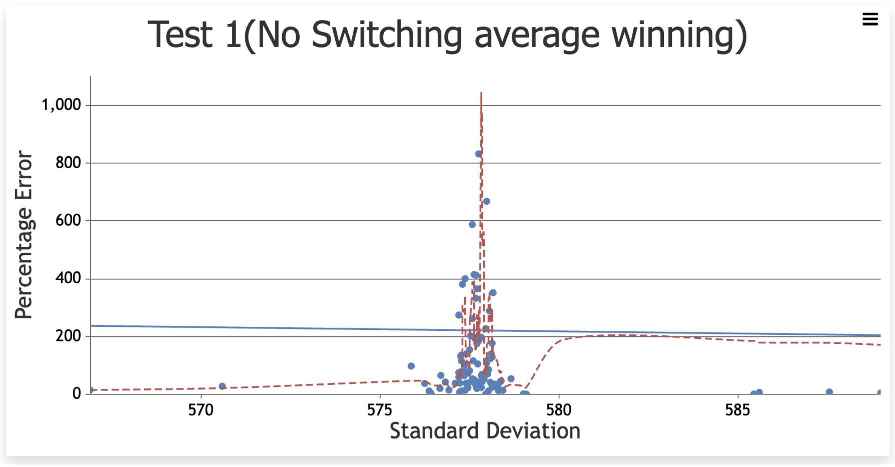


Figure D.2: Test 1 No switching

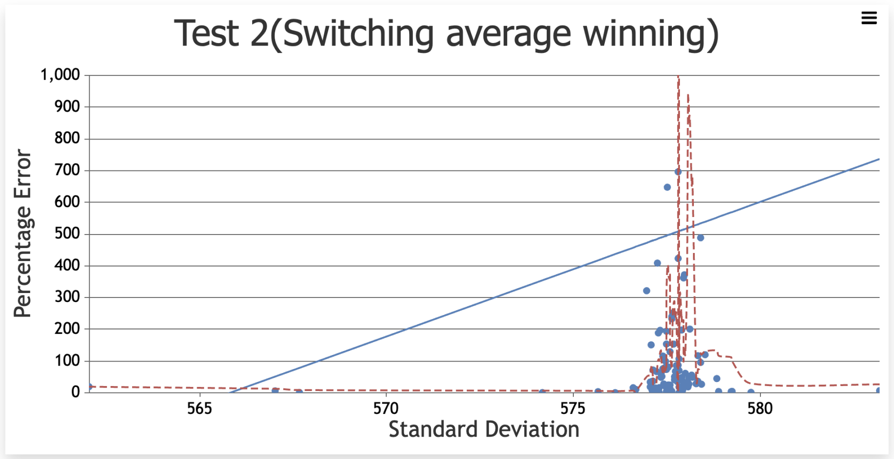


Figure D.3: Test 2 Switching

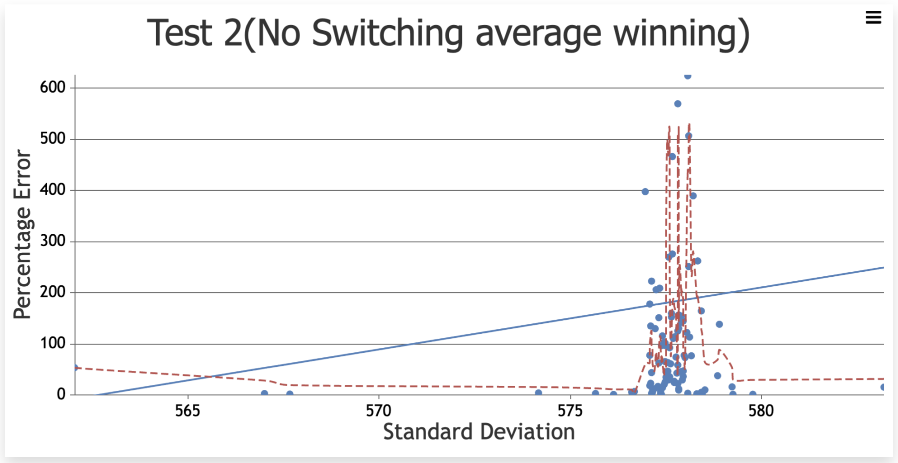


Figure D.4: Test 2 No switching

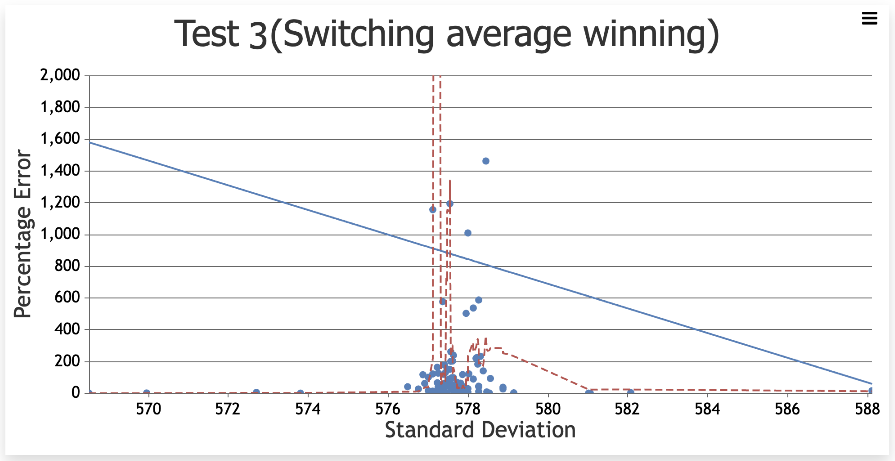


Figure D.5: Test 3 Switching

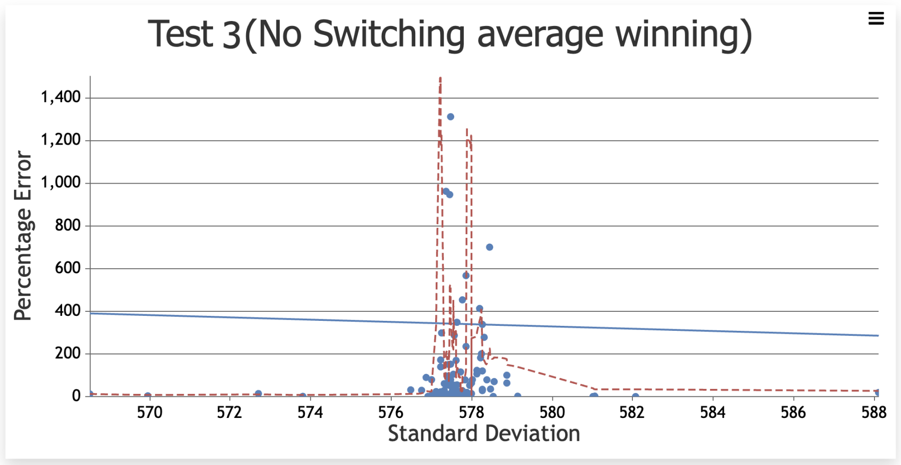


Figure D.6: Test 3 No switching

D.4 Population Size - Normally distributed randomized set

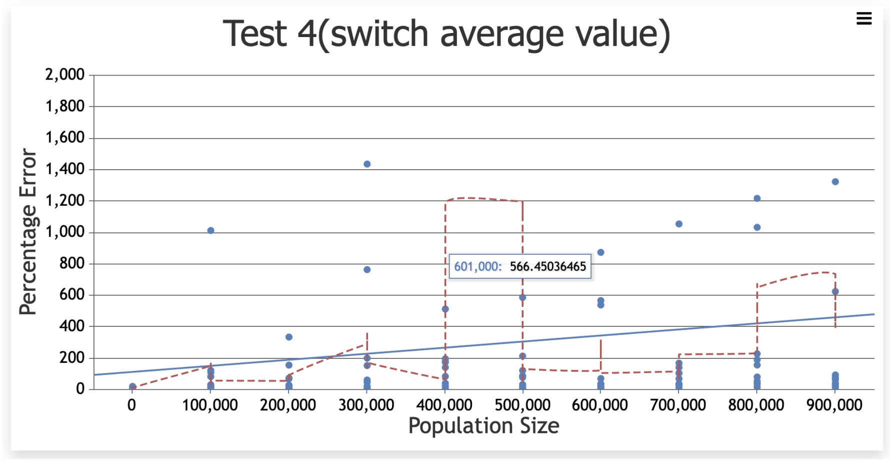


Figure D.1: Test 4 Switching

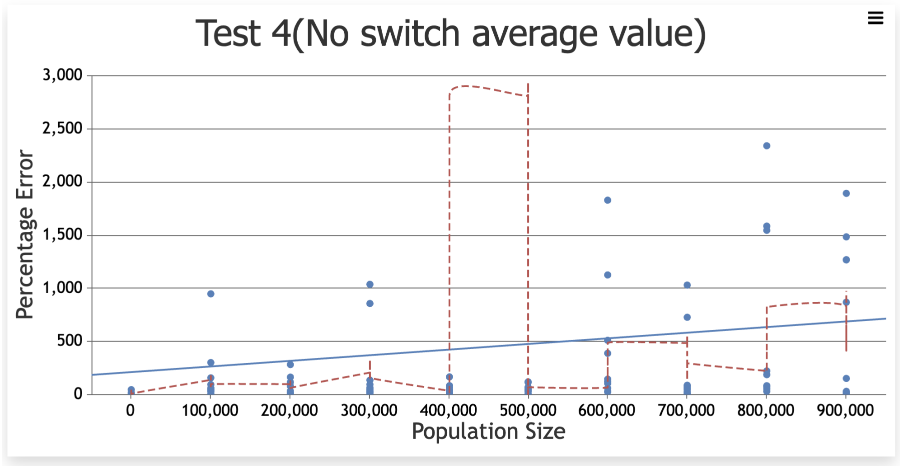


Figure D.2: Test 4 No switching

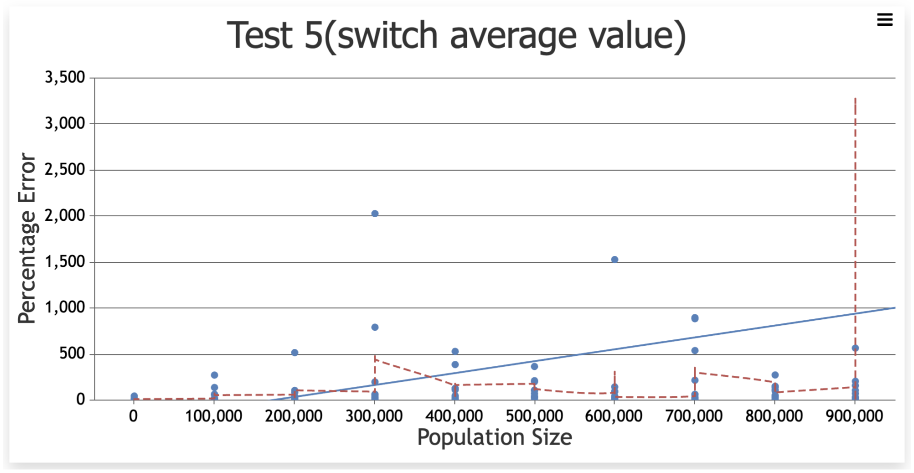


Figure D.3: Test 5 Switching

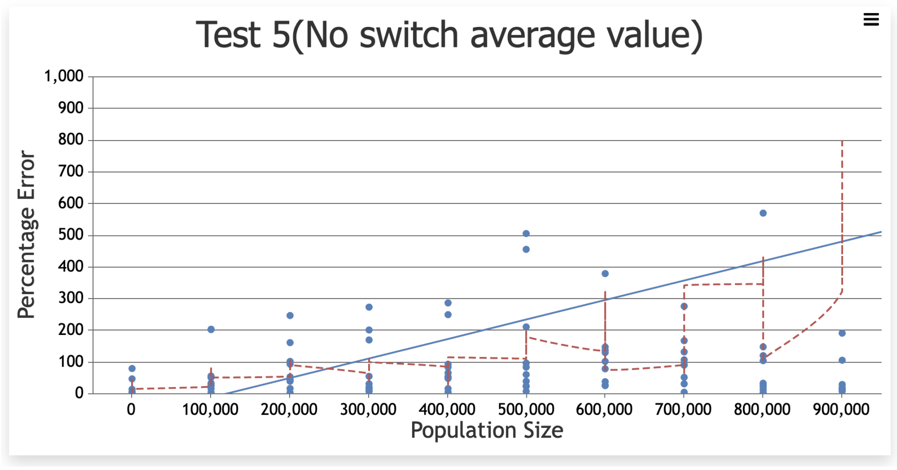


Figure D.4: Test 5 No switching

D.5 Total Sum of the Population - Normally distributed randomized set

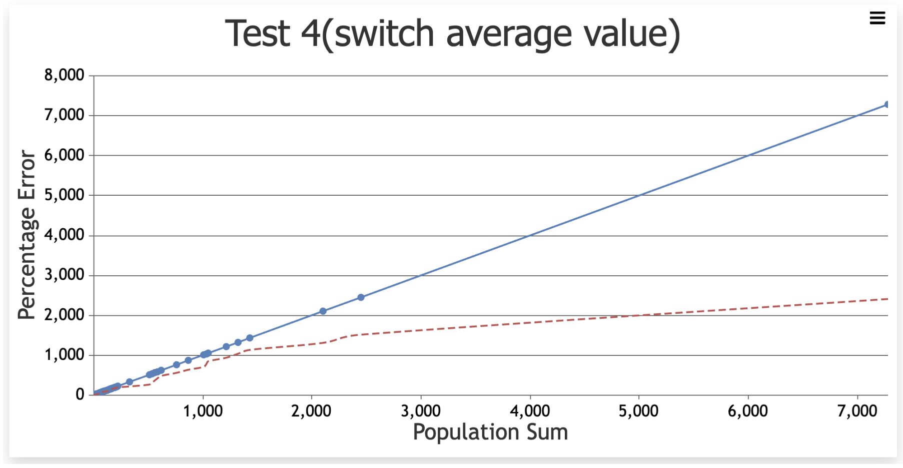


Figure D.1: Test 4 Switching

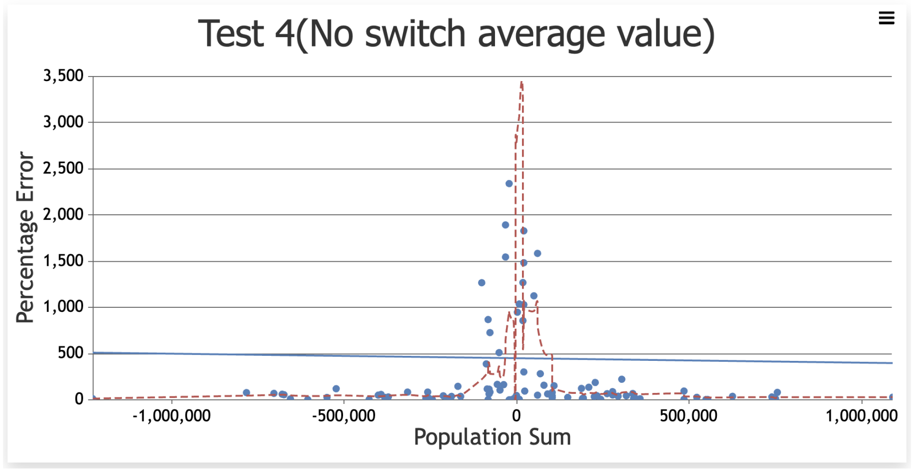


Figure D.2: Test 4 No switching

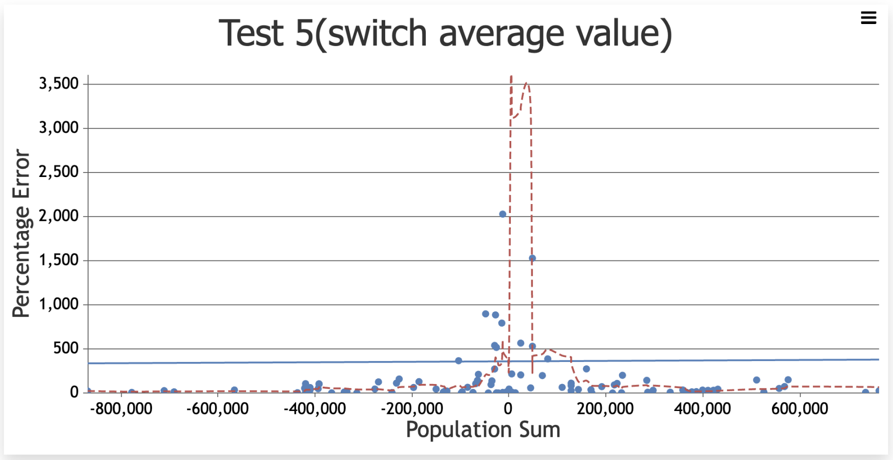


Figure D.3: Test 5 Switching

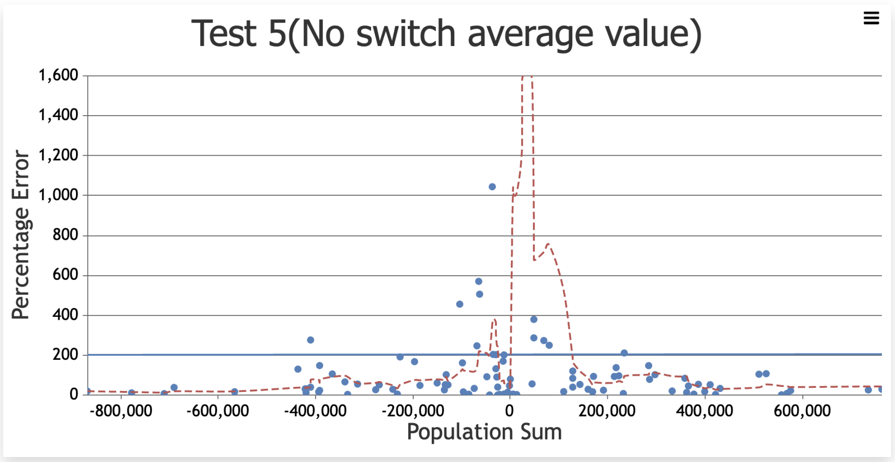


Figure D.4: Test 5 No switching

D.6 Standard deviation - Normally distributed randomized set

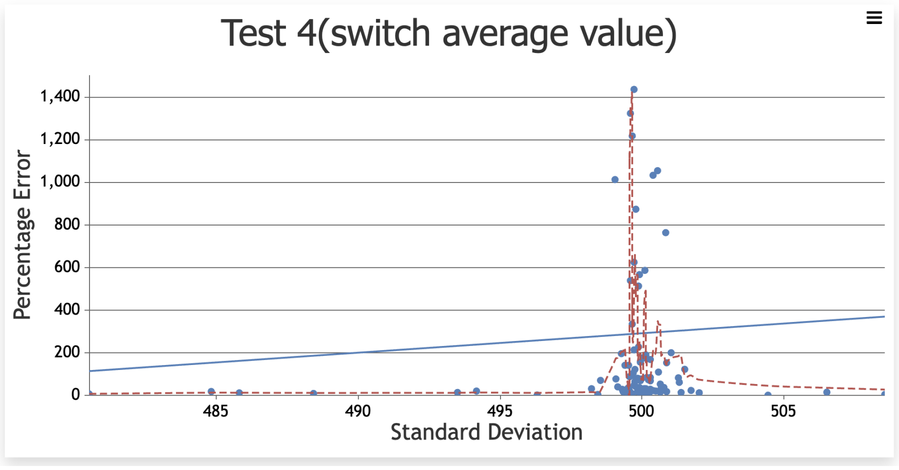


Figure D.1: Test 4 Switching

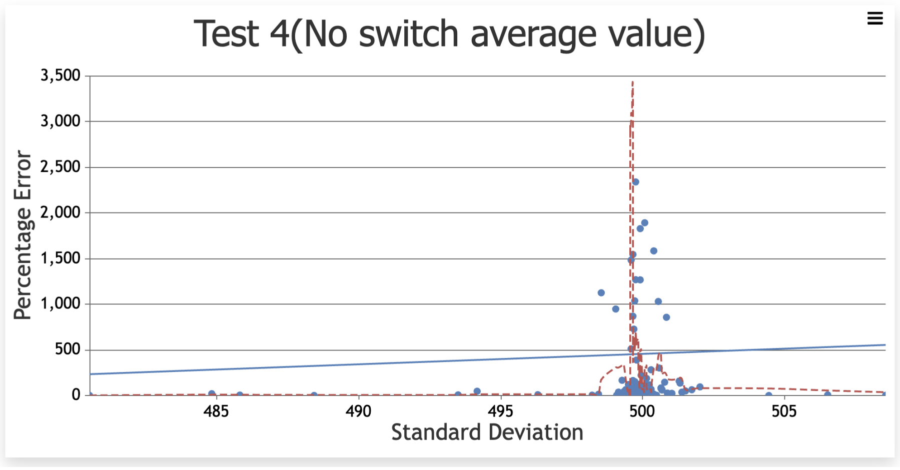


Figure D.2: Test 4 No switching

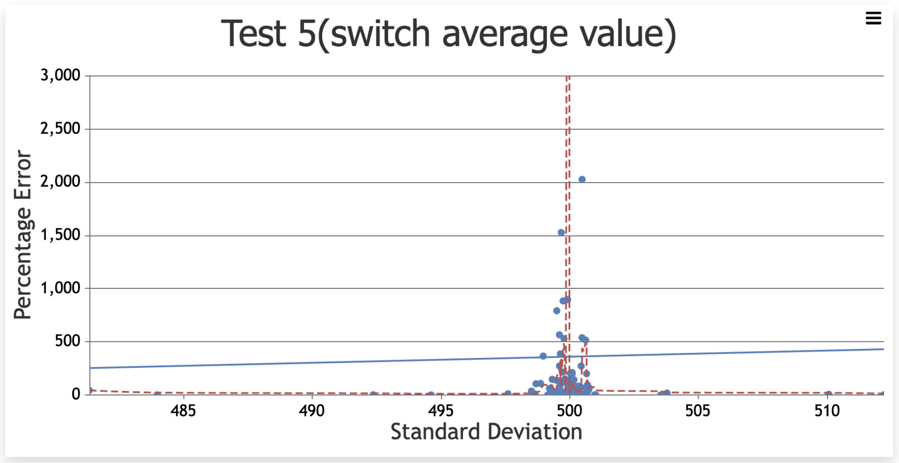


Figure D.3: Test 5 Switching

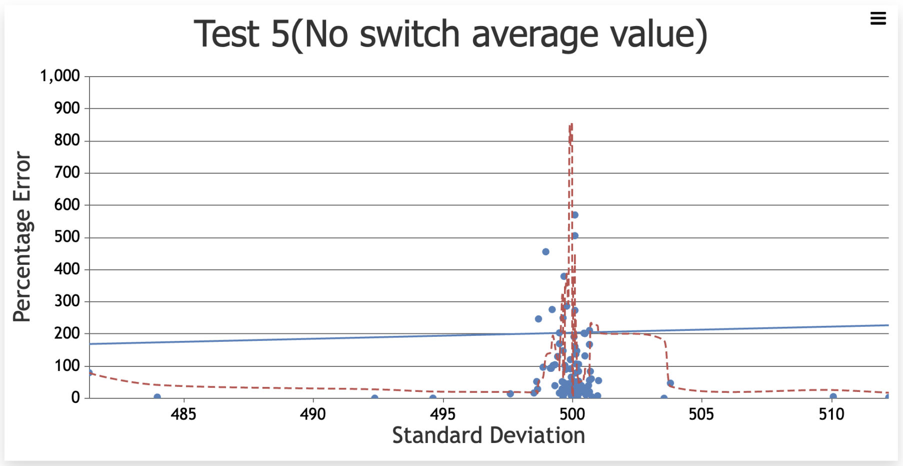


Figure D.4: Test 5 No switching

D.7 Population Size - Uniformly distributed set

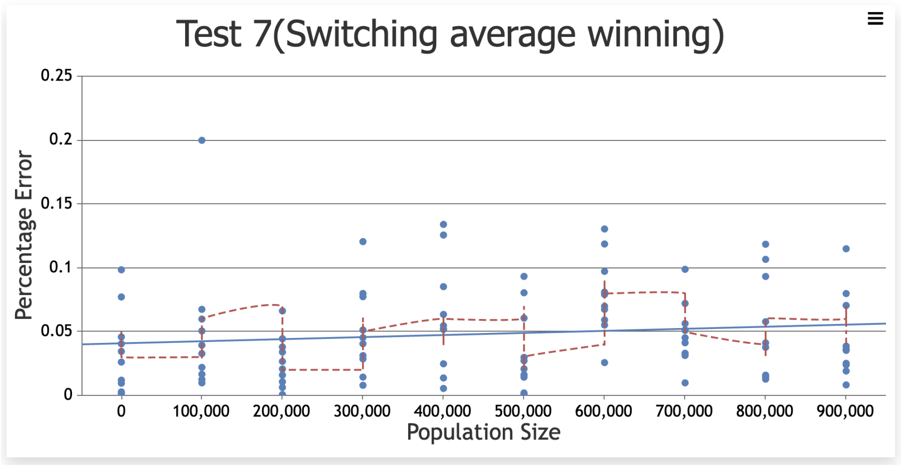


Figure D.1: Test 7 Switching

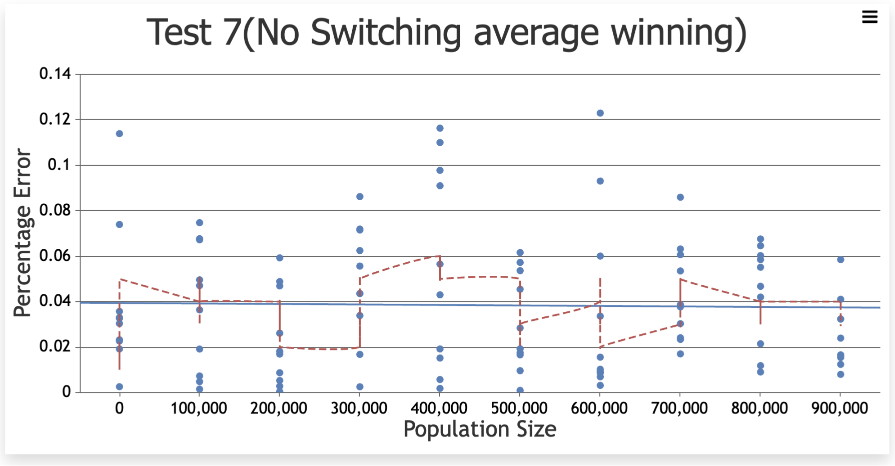


Figure D.2: Test 7 No switching

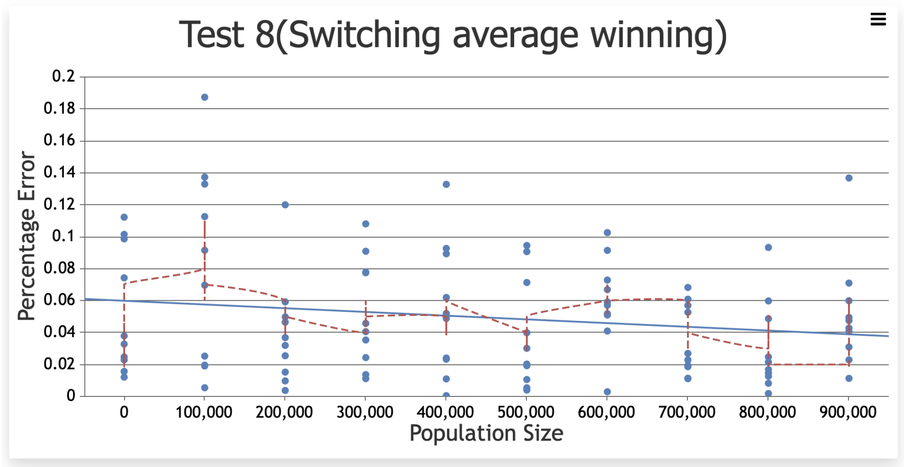


Figure D.3: Test 8 Switching

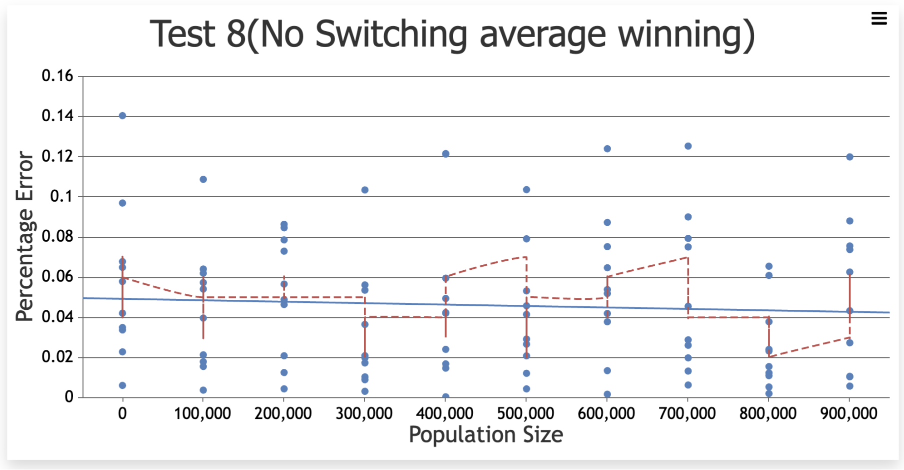


Figure D.4: Test 8 No switching