University of Waterloo
Faculty of Engineering
Department of Electrical and Computer Engineering

Simulation and Analysis of Stochastic Processes

Faculty of Engineering, University of Waterloo
Waterloo, Ontario

Prepared by
Prabal Gupta

p25gupta@edu.uwaterloo.ca
2B Computer Engineering

3 January 2018

January 3, 2018

Vincent Gaudet, Chair
Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario
N2L 3G1

Dear Sir,

This report, entitled "Simulation and Analysis of Stochastic Processes", was prepared as my 2B Work Report for Prof. Ravi Mazumdar in the Electrical and Computer Engineering Department at the University of Waterloo. This report is in fulfillment of the course WKRPT 201. The purpose of this report is to summarize the major conclusions I reached when simulating and analyzing various random phenomena during my co-op work placement. This is done by analysing the "Gambler's Ruin" game, which is an active topic of research, as an example.

Prof. Ravi Mazumdar is a part of the Electrical and Computer Engineering Department, where he performs research in various fields such as applied probability and optimization, with emphasis on practical applications in communication systems, stochastic models, and statistical methods.

I worked as an Undergraduate Research Intern under the direct supervision of Prof. Ravi Mazumdar, and was responsible for performing and quantifying results of various simulations involving numerous random phenomena.

I would like to thank Prof. Ravi Mazumdar for assisting me throughout my co-op work placement. Prof. Mazumdar's academic expertise and guidance were quite valuable, and helped enrich my experience as a student and as an intern. I am also thankful to Prof. Mazumdar for proofreading my 2B Work Report and providing appropriate feedback. I hereby confirm that I have received no further help other than what is mentioned above in writing this report. I also confirm this report has not been previously submitted for academic credit at this or any other academic institution.

Sincerely,

Prabal Gupta

Contributions

During my co-op work placement, I was employed in the Faculty of Engineering at the University of Waterloo in the position of Research Assistant. I was working under the direct supervision of Prof. Ravi Mazumdar, who guides a small group of graduate students and postdoctoral scholars in the Department of Electrical and Computer Engineering.

Prof. Mazumdar's team studies various topics in the field of applied probability analysis, optimization problems, communication theory, statistical methods etc. The tasks I was assigned to were mostly related to probabilistic analysis and simulations.

My work as a Research Assistant was largely aimed at teaching me analysis and simulation of a wide array of stochastic processes. A stochastic process is defined as a phenomenon that is composed of multiple random variables, which attain random values in accordance with certain properties. Value of a random variable is determined by the result of an experiment. Stochastic processes often behave in unpredictable ways due to their inherent randomness, but often have certain predictable properties. Prof. Mazumdar trained me during the first month of my co-op term, and helped me learn about probabilistic analysis, computer simulations, and presenting simulation results. The theoretical aspect of my training process involved learning about the theory of discrete and continuous random variables, probability distributions, joint probability distributions, simulation techniques for producing different types of continuous and discrete probability distributions, Markov chains, and Queueing theory. The practical portion of my training included learning about MATLAB, software optimization techniques, and presentation of collected data.

The following are some of the major projects that I completed throughout the term:

1.) *Identifying Data Distribution:* Given a set of random data, we try to calculate how close the probability distribution of the random variable that produced the empirical data is to the ideal probability distribution.
2.) *Queue and Server Simulations:* We analyze the behaviour of a queueing and server systems that receive the k^{th} request after an inter-arrival delay S_k, and process it in D_k units of time. I performed simulations where S_k, D_k were different types of random variables. The performance of these systems is often sensitive to the type and behaviour of random variables S_k, D_k. One advantage of analyzing queueing and server systems is the ease of modelling and simulating them. Moreover, they are of great practical relevance in the field of computing. The theory for analyzing queues and servers is well developed, and it is often possible to analytically confirm a queue/server model's simulated performance.
3.) *Gambler's Ruin:* The "Gambler's Ruin" game involves some people with a certain amount of starting currency. During each iteration of the game, we randomly choose a pair of

players, from which we arbitrarily choose a winner and a loser. The winner gains \$1, and the loser forgoes \$1.

We simulate "Gambler's Ruin" game with the aim of quantifying game's completion time, which is a random variable. Additionally, we aim to study the observed mean, and asymptotic nature of the game's expected completion time with increase in the starting amount of money the players start with. Simulating "Gambler's Ruin" is quite challenging due to random completion time of each game, and the number of games you need to play to see the expected value converging to a constant.

The "Gambler's Ruin" game is currently an active topic of research. We use the "Gambler's Ruin" game as an example to outline the process of generating optimal simulation algorithms, and the process of analyzing data received from simulations.

In addition to the above projects, I also developed many general tools and techniques to ease and improve the process of developing a probabilistic simulation. These tools include:

- *Plotting Utility:* A MATLAB function `ezdraw` to reduce the lengthy process of printing data, resizing graphs, showing additional relevant information, printing data mean, filtering noisy data, etc. to a simple function call.
- *Simulation Optimization:* Techniques to improve simulation performance to increase the rate of data production. This involves optimizing existing simulation code to reduce bottlenecks, code branching, cache misses, etc. I was able to increase the performance of the "Gambler's Ruin" simulation by 750% by optimizing the simulation code, which allowed us to generate the relevant data at a rate that was nearly an order of magnitude faster than earlier.

The analytical component of my co-op term often required proofs to confirm the empirical evidence obtained from simulations, which helped my build my critical analysis skills in a mathematical context. One of the biggest advantages I had was the ability to leverage my knowledge of mathematical tools frequently used in engineering for probabilistic analysis, which allowed me to appreciate the applications of tools such as frequency analysis, convolution, discrete mathematics, and calculus in probability theory. Frequency analysis techniques, especially Fourier and Laplace transforms, were very useful in solving probability theory problems involving the convolution operator, which tends to appear when studying joint probability distributions involving multiple random variables. Additionally, my co-op term also helped me develop my presentation skills due to the necessity of condensing, formatting, and presenting data in a comprehensible form for review by Prof. Mazumdar.

The tasks I completed during my co-op term were aimed at training me in probability analysis, but indirectly led to creation of novel techniques and valuable datasets that might be useful for research purposes in future.

Summary

The main purpose of the report is to outline the process of creating stochastic simulations and analyzing obtained data using the "Gambler's Ruin" game as an example. The game involves multiple players who begin the game with some money. During each round, a pair of players is randomly selected, from which a winner and a loser are arbitrarily chosen. The winner gains $1 while the loser forgoes $1. The game is played till someone loses all the money they started with. We aim to study how long we can expect a game to last given how much money its players start with.

The major points documented in this report involving simulation algorithms revolve around generating optimized simulation algorithms for the "Gambler's Ruin" game using techniques such as coalescing multiple operations, predicting resource usage, pre-calculating results, reserving computer resources in advance, and minimizing performance penalties associated with usage of conditional statements.

The content concerning data obtained from simulations includes analysis of game's completion time and the rate at which a simulation's resource usage grows. Additionally, running averages are used to confirm whether the simulation results match the known solutions, and to predict the expected completion time of configurations without known analytical expressions.

The major conclusion concerning simulation algorithms is that the novel simulation algorithm developed using optimization techniques is 750% faster than a naïve implementation, and has the capability of simulating 10 million "Gambler's Ruin" games involving three players within the given constraint of 150 seconds.

It is shown and concluded that the completion time of the "Gambler's Ruin" game is directly proportional to m^2, where m is the amount of money that all players start with. Additionally, it is shown that the same result holds when a single player's wealth significantly exceeds that of the others.

It is recommended that the simulation algorithms be further improved using additional optimization techniques like parallelism, and extensive pre-calculation of resources prior to performing a simulation.

It is also recommended that additional techniques be used to study behaviour of the completion times of "Gambler's Ruin" games. Techniques such as creating a histogram of completion times, and using geometrical models to model the stochastic process are suggested.

Table of Contents

List of Figures

List of Tables

List of Code Fragments

1 Introduction

This report revolves around analysis and simulation of the "Gambler's Ruin" game, which involves multiple players, each of them starting with a certain amount of money. During each round of the game, we choose a random pair of players, and arbitrarily select a winner and a loser. The winner gains \$1, while the loser forgoes \$1. We keep repeating the process till some player has no wealth left. We aim to study the relationship between the wealth that the players start the game with, and the expected completion time of the game. In this context, completion time refers to number of gambles that occur in a game. It is known that the completion time of a "Gambler's Ruin" game is random, thus the completion time can be treated as a random variable [1].

This report involves studying the observed mean and asymptotic behaviour of completion time with respect to change in the wealth that the players start with. This particular section aims to explain the problem, known solutions for particular cases, and constraints for the solutions we seek.

1.1 General Notation

For the sake of notational convenience, we will refer to a "Gambler's Ruin" simulation with k players and r games as $G_r^{(k)}$, and the completion time of the *i*th game with k players as $T_i^{(k)}$.

Each simulation $G_r^{(k)}$ can be notated as a vector of multiple completion times $G_r^{(k)} = \langle T_1^{(k)}, T_2^{(k)}, T_3^{(k)}, \dots, T_4^{(k)} \rangle$. The starting wealth of each of the k players will be referred to as $m_1, m_2, \dots, m_k$. Each 'gamble' in a game refers to a single selection of a random pair of players (including the choice of a winner and a loser).

Additionally, any "Gambler's Ruin" game with k players will be represented as $G^{(k)}$ having completion time $T^{(k)}$.

Thus, the following is true for any simulation $G_r^{(k)}$:

$$\lim_{r \to \infty} \left(\frac{1}{r} \cdot \sum_{i=1}^{r} T_i^{(k)} \right) = \mathrm{E}[T^{(k)}] \tag{1}$$

Expression (1) simply expresses the fact that larger the number of games we play in a single simulation G_r^k, the closer their mean tends to be to the ideal expected value $\mathrm{E}[T^{(k)}]$.

1.2 Known Solutions

The expected completion times for the "Gambler's Ruin" game involving 2 and 3 players are known to be $\mathrm{E}[T^{(2)}] = m_1 m_2$, and $\mathrm{E}[T^{(3)}] = \frac{3m_1 m_2 m_3}{m_1+m_2+m_3}$ [1] [2]. The general expression for $\mathrm{E}[T^{(k)}]$ for some integer $k > 3$ is not known.

Note that the completion times $T^{(2)}, T^{(3)}$ of games $G^{(2)}, G^{(3)}$ with every player having the same amount of money m have the same asymptotic tight-bound:

$$\mathrm{E}\left[T^{(2)}\right] = \mathrm{E}\left[T^{(3)}\right] = m^2 \in \Theta(m^2) \qquad (2)$$

1.3 Solution Requirements

We are required to find the following information about the "Gambler's Ruin" game (see Appendix A for additional information):

1. A novel algorithm for simulating some game $G^{(k)}$, where $k \geq 2$ such that the asymptotic behaviour of the algorithm's runtime depends only on $\mathrm{E}[T^{(k)}]$.
2. Analyzing the following empirical data related to random variable $T_i^{(k)}$, where integer $i = 1, 2, 3 \ldots, r$ (r is the number of games simulated):
   - Expected value $\mathrm{E}[T^{(k)}]$ of random variable $T^{(k)}$.
   - Asymptotic behaviour of $\mathrm{E}[T^{(k)}]$ with respect to change in money $m_1, m_2, \ldots, m_k$ that players start with.

For a complete list of which simulations need to be performed, refer to Table 2 in Appendix A.

2 Possible Simulation Algorithms

We take a look at three different algorithms for running a simulation $G_r^{(k)}$. Although all three algorithms give the same results and have runtimes with identical asymptotic growth-rate, their actual performance varies significantly due to many differences in implementation.

Note that we treat k as a constant during asymptotic runtime analysis of these algorithms since our goal is to analyze behaviour of $G_r^{(k)}$ as $m_1, m_2, \dots, m_k$ grow. Moreover, error checking mechanisms have not been explicitly included in the presented algorithms to keep algorithms concise.

All of the three presented algorithms make use of discrete uniform random variable generator that uses the following probability mass function to generate random integers in the interval [a, b] (see Table 4 in Appendix B.2):

$$f(k) = \begin{cases} 0, & k \notin [a,b] \cap \mathbb{Z} \\ \dfrac{1}{b-a+1}, & k \in [a,b] \cap \mathbb{Z} \end{cases}$$

2.1 Naïve Implementation

This implementation manually iterates through each gamble of every game within a simulation, which causes a lot of overhead related to loop variables. We use memory pre-allocation to improve the performance of our algorithm.

2.1.1 Memory Pre-allocation

Instead of repeatedly resizing an array when an element needs to be added to it, a better alternative is to allocate the entire array before starting the loop, and then updating elements of the pre-allocated array. Pre-allocation provides significant performance gains since memory allocation and deallocation are big performance bottlenecks. Additionally, frequent allocation and de-allocation lead to memory fragmentation, which leads to further performance degradation [3].

In Code Fragment 1, we pre-allocated the `completion times` array, which led to significant improvement in performance for very large values of r in a simulation $G_r^{(k)}$.

Code Fragment 1: Naïve algorithm for simulating $G_r^{(k)}$

```
#initialize simulation parameters
k, r, m ← players, number_of_tests, [m₁, m₂, …, mₖ]

#pre-allocate ‘r’ element array for storing completion times
completion_times ← make_array(r)

#pre-calculate all possible pairs (sets) of players from ‘k’ players
possible_pairs ← make_pairs(r, ‘combinations’)
pair_count ← length(possible_pairs)

#run all games in simulation
for each game_number ∈ [1, 2, …, r]
   temporary_m, completion_time ← copy(m), 0

   #keep running ‘gambles’ till the game lasts
   while 0 ∉ temporary_m
        #choose winner and loser
        random_pair ← possible_pairs[random_integer([1, pair_count])]

        winner ← random_integer(1, 2)
        loser ← modulus(winner, 2) + 1

        winner, loser ← random_pair[winner], random_pair[loser]

        temporary_m[winner] ← temporary_m[winner] + 1
        temporary_m[loser] ← temporary_m[loser] - 1

        #increment timer
        completion_time ← completion_time + 1
   end

   #store game’s completion time in pre-allocated array
   completion_times[game_number] ← completion_time
end
```

2.2 Predictive Vectorization

This implementation, like the naïve implementation, manually iterates through each gamble of every game within a simulation. Thus, it also suffers from the associated overhead of loop variables. In addition to memory pre-allocation, we also used another technique to predict resource usage in order to improve our algorithm’s performance.

One of the major problems that hindered optimization of “Gambler’s Ruin” simulation is the fact that the completion time of each game within a simulation is random. This makes it hard to pre-allocate and pre-calculate resources used within the loop in advance, since we cannot know the length of the resource arrays beforehand. In this context, the word ‘resources’ refers to the random pairs of players, winners, and losers that need to be chosen for each gamble in a game. Repeatedly calculating each of these within the simulation loop led to significant performance costs, which we were able to partially overcome by predicting resource usage and pre-calculating values beforehand, and then providing additional resources if our actual resource consumption exceeded the predicted value.

2.2.1 Predicting Resource Usage

Note that the number of random pairs of players needed for simulating a single game depends upon the completion time, which cannot be known in advance. Moreover, the general expression for $\mathrm{E}[T^{(k)}]$ is not known for $k > 3$. In Code Fragment 2, we predict the simulation time $T_i^{(k)}$ of a game within a simulation $G_r^{(k)}$ by exponentially averaging $T_j^{(k)}$ for $j \in \{1, 2, 3, \dots, i-1\}$. Exponential averaging weighs the older values less, and newer values more [4]. Thus, as we simulated more games, the initial value of variable `expected_completion_times` in Code Fragment 2 became irrelevant. We used equation (3) for exponentially averaging completion times in a simulation (α is the weight assigned to the latest observed value).

$$\mathrm{PREDICTION}_i = \alpha \times T_{i-1}^{(k)} + (1-\alpha) \times \mathrm{PREDICTION}_{i-1} \qquad (3)$$

2.2.2 Vectorization

Pre-calculating resources before they are actually used is often faster than calculating resources on-the-run. This is because certain operations have better performance when processing elements collectively, than when the elements are processed individually. The process of reorganizing code to replace loops working individually on long data sets with fewer, coalesced operations is known as vectorization [5].

Modern computers support high-level operations that work on entire arrays of numerical inputs for efficiency. The significant performance gains are a result of reduced branching, long pipeline, fewer instruction fetches etc. [5] [6]. We utilized vectorization in Code Fragment 2 to pre-calculate a list of random pairs of people who would gamble in the game, and a list of winners and the losers.

Code Fragment 2: Predictive and vectorized algorithm for simulating $G_r^{(k)}$

```
#initialize simulation parameters
k, r, m ← players, number_of_tests, [$m_1$, $m_2$, …, $m_k$]

#pre-allocate ‘r’ element array for storing completion times
completion_times ← make_array(r)

#pre-calculate all possible player pairs (vectors) from ‘k’ players
possible_pairs ← make_pairs(r, ‘permutations’)
pair_count ← length(possible_pairs)

buffer ← 1.5
expected_completion_time ← buffer × ( $m_1$×$m_2$ if k = 2 else 3×$m_1 m_2 m_3$/($m_1$+$m_2$+$m_3$ ) )

#assign arrays for storing pre-calculated game trials
range ← [1, pair_count]
random_pair_choices, winner_choices, loser_choices ← [], [], []

#run all games in simulation
for each game_number ∈ [1, 2, …, r]
   temporary_m, completion_time ← copy(m), 0

   #keep running ‘gambles’ till the game lasts
   while 0 ∉ temporary_m
        index_access ← modulus(completion_time, expected_completion_time) + 1

        #if we have ‘run out’ of pre-calculated values, we re-calculate multiple
        #...possible pairs (combinations) of people, and lists of winners and losers
        if index_access = 1 then
                random_pair_choices ← random_integers(range, expected_completion_time)
                winner_choices ← row[1] of possible_pairs[random_pair_choices]
                loser_choices ← row[2] of possible_pairs[random_pair_choices]
        end

        #give $1 to winners, take $1 from losers
        increment temporary_m[winner_choices[index_access]] by 1
        increment temporary_m[loser_choices[index_access]] by -1

        #increment timer
        completion_time ← completion_time + 1
   end

   #store game’s completion time in pre-allocated array
   completion_times[game_number] ← completion_time

   #update the ‘expected_completion_time’ using exponential averaging
```

```
    updation_rate ← 0.1

    expected_completion_time ← expected_completion_time × (1-updation_rate)
    increment expected_completion_time by (completion_time × updation_rate)
    expected_completion_time ← ⌈expected_completion_time⌉
end
```

2.3 Massive Predictive Vectorization

We extended the concept of prediction and vectorization to other parts of the algorithm to further improve performance. This version of the simulation algorithm does not iterate through each gamble of every game in the simulation, and instead uses vectorization to simulate multiple gambles within each game. This led to a considerable gain in performance relative to our previous implementations.

2.3.1 Massive Vectorization

This implementation of the simulation algorithm replaces iteration through each gamble with multiple, vectorized gambles in a single game. Since the completion time $T_i^{(k)}$ is a random variable, the algorithm predicts the value of $T_i^{(k)}$ by exponentially averaging the completion times of prior games within the same simulation.

The algorithm then uses the predicted completion time of upcoming game to create a list of length `expected_completion_time` that contains multiple, randomly chosen gambles. Then it checks whether a gamble in the list of pre-calculated gambles leads to termination of the game. If not, the process of creating and testing another list of random gambles is repeated.

Code Fragment 3: Massively predictive and vectorized algorithm for simulating $G_r^{(k)}$

```
#initialize simulation parameters
k, r, m ← players, number_of_tests, [m₁, m₂, …, mₖ]

#pre-allocate ‘r’ element array for storing completion times
completion_times ← make_array(r)

#pre-calculate all possible player pairs (vectors) from ‘k’ players
possible_pairs ← make_pairs(r, ‘permutations’)
pair_count ← length(possible_pairs)

buffer ← 1.5
```

```
expected_completion_time ← buffer × ( $m_1 \times m_2$ if k = 2 else $3 \times m_1 m_1 m_1 / (m_1 + m_2 + m_3)$ )

#assign arrays for storing pre-calculated game trials
range ← [1, pair_count]
random_pair_choices, winner_choices, loser_choices ← [], [], []
cumulative_gamble ← []

#pre-calculate all possible gambling moves
sample_gamble ← make_array(k)
sample_gamble[1 to end-2], sample_gamble[end-1 to end] ← 0, [-1, 1]

possible_gambles ← permutations(sample_gamble)

#run all games in simulation
for each game_number ∈ [1, 2, …, r]
    temporary_m, completion_time ← copy(m), 0

    #keep running ‘gambles’ till the game lasts
    exit_loop ← False
    while not exit_loop
        #create multiple random gambling plays at once
        gambles ← possible_gambles(random_integers(range, expected_completion_time))
        cumulative_gamble ← running_total(gambles)

        #if we have ‘run out’ of pre-calculated values, we re-calculate multiple
        #...possible pairs (combinations) of people, and lists of winners and losers
        cumulative_gamble_result ← matrix(length(gambles), k) with temporary_m as rows
        cumulative_gamble_result ← cumulative_gamble_result + cumulative_gamble

        first_zero_address ← address where cumulative_gamble_result = 0
        if first_zero_address ≠ (0, 0) then
            #find the gamble number that led to termination of the game
            played_gambles_count ← first_zero_address[1]

            #terminate current game since someone is in ruin
            exit_loop ← True

            #update money counter to account ONLY for gambles played
            increment completion_time by played_gambles_count
        else
            #update money counter to account for ALL gambles
            increment completion_time by expected_completion_time
        end

        #update money counter
        temporary_m ← cumulative_gamble_result[end, 1 to k]
    end
    #store game's completion time in pre-allocated array
    completion_times[game_number] ← completion_time

    #update the 'expected_completion_time' using exponential averaging
    updation_rate ← 0.1

    expected_completion_time ← expected_completion_time × (1-updation_rate)
    increment expected_completion_time by (completion_time × updation_rate)
    expected_completion_time ← ⌈expected_completion_time⌉
end
```

2.4 Performance and Suitability

We used a series of simulations to compare performance of the three simulation algorithms in MATLAB. Table 1 lists the simulations and the amount of time taken by the three algorithms to complete them when all the players start with $10 as the initial amount of money. Note that all the timing data was calculated in accordance with equation (1).

Table 1: Series of simulations to test performance of simulation algorithms

Simulation	**Naïve Implementation (seconds)**	**Predictive Vectorization (seconds)**	**Massive Predictive Vectorization (seconds)**
$G_{100}^{(3)}$	0.014015	0.012466	0.001575
$G_{1,000}^{(3)}$	0.098004	0.046740	0.014157
$G_{10,000}^{(3)}$	0.988589	0.467948	0.117801
$G_{100,000}^{(3)}$	9.489032	4.479862	1.094917
$G_{1,000,000}^{(3)}$	94.739582	44.821218	10.627560
$G_{10,000,000}^{(3)}$	988.448633	461.566486	117.592251

We can see from the above table that the "Massive Predictive Vectorization" algorithm is the only one of the three choices that satisfied the wall-clock time-constraint mentioned in Appendix A. The "Massive Predictive Vectorization" algorithm is about 300% faster than the "Predictive Vectorization" algorithm, and about 750% faster than the "Naïve Implementation". Thus, the third algorithm was used to perform all the required simulations due to its superior performance.

3 Simulations Performed

We used the "Massive Predictive Vectorization" algorithm to run all sets of simulations mentioned in Table 2 in Appendix A.

During this section, we will implicitly refer to the starting wealth of players $1, 2, 3, \ldots, k-1$ in a simulation $G_r^{(k)}$ as m_A, and starting wealth of player k as m_B.

3.1 Asymptotic Analysis

This first two tests involved analysis of asymptotic growth of completion times $T^{(k)}$ with change in m_A, m_B.

We performed the tests for $k \in \{2, 3, 4, 5, 6\}$ when $m_A = m_B$, and for $k \in \{3, 4, 5, 6\}$ when $m_A = 1000 \times m_B$ i.e. when $m_B \gg m_A$.

3.1.1 When $m_A = m_B$

Figure 1 compares the observed growth rate of $T^{(2)}$ and $T^{(3)}$ when $m_A = m_B$. It was apparent that the asymptotic behaviour of $\mathrm{E}[T^{(2)}], \mathrm{E}[T^{(3)}]$ is similar to that of m^2 i.e. $\mathrm{E}[T^{(2)}], \mathrm{E}[T^{(3)}] \in \Theta(m^2)$, which was confirmed by equation (2) and the content relevant to it. Every simulation consisted of 200 games for each value of $m \in \{1, 2, 3, \ldots, 200\}$.

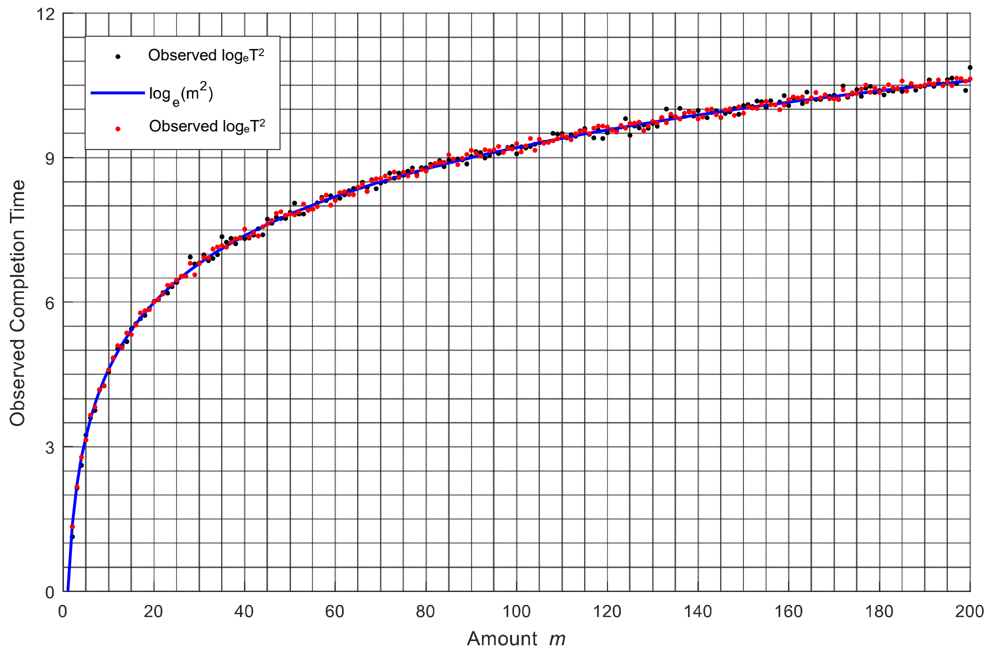

Figure 1: Growth of completion times when $m_A = m_B$, $k = 2$ and 3

Similar behaviour is seen when preforming the first test with $k \in \{4, 5, 6\}$. The results in Figure 2 indicate that $\mathrm{E}[T^{(4)}], \mathrm{E}[T^{(5)}], \mathrm{E}[T^{(6)}] \in \Theta(m^2)$ too, since the curves for $\ln(\mathrm{E}[T^{(4)}])$, $\ln(\mathrm{E}[T^{(5)}])$ and $\ln(\mathrm{E}[T^{(6)}])$ seem to be only a constant offset apart from $\ln(m^2)$.

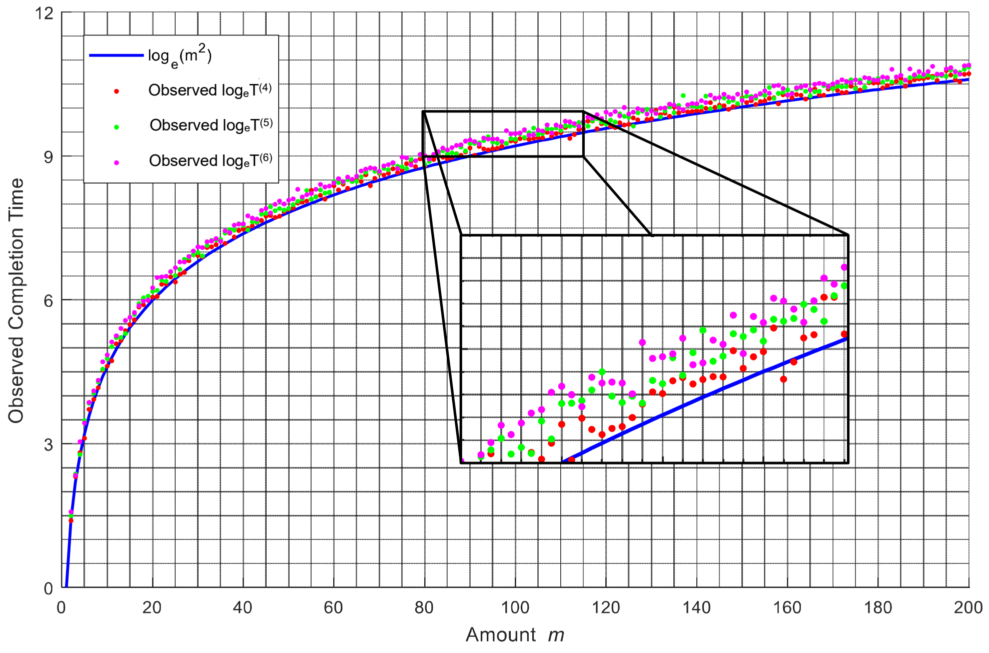

Figure 2: Growth of completion times when $m_A = m_B$, k = 4, 5, and 6

3.1.2 When $m_B = 1000 \times m_A$

Figure 3 compares the observed growth rate of $T^{(3)}$ with that of ${m_A}^2$. It was apparent that the asymptotic behaviour of $\mathrm{E}[T^{(3)}]$ is similar to that of ${m_A}^2$ i.e. $\mathrm{E}[T^{(3)}] \in \Theta({m_A}^2)$, which was confirmed using general expression for $\mathrm{E}[T^{(3)}]$ given in [1], [2]. Additionally, it is also possible to prove that $\mathrm{E}[T^{(2)}] \in \Theta({m_A}^2)$.

$$\begin{aligned} \mathrm{E}[T^{(2)}] &= m_A m_B \\ &= m_A \cdot (1000 \times m_A) \\ &= 1000 \cdot {m_A}^2 \in \Theta({m_A}^2) \end{aligned}$$

$$\begin{aligned} \mathrm{E}[T^{(3)}] &= \frac{3 \times m_A m_A m_B}{m_A + m_A + m_B} \\ &= \frac{3 \times m_A m_A \cdot (1000 \times m_A)}{m_A + m_A + (1000 \times m_A)} \\ &= {}^{500}/_{167} \times {m_A}^2 \in \Theta({m_A}^2) \qquad (4) \end{aligned}$$

Every simulation in Figure 3 consisted of 200 games for each value of $m_A \in \{1, 2, 3, \ldots, 200\}$, just like in Figure 1. Despite having conducted same number of trials for each value of m_A, Figure 3 data shows greater variance than data in Figure 1.

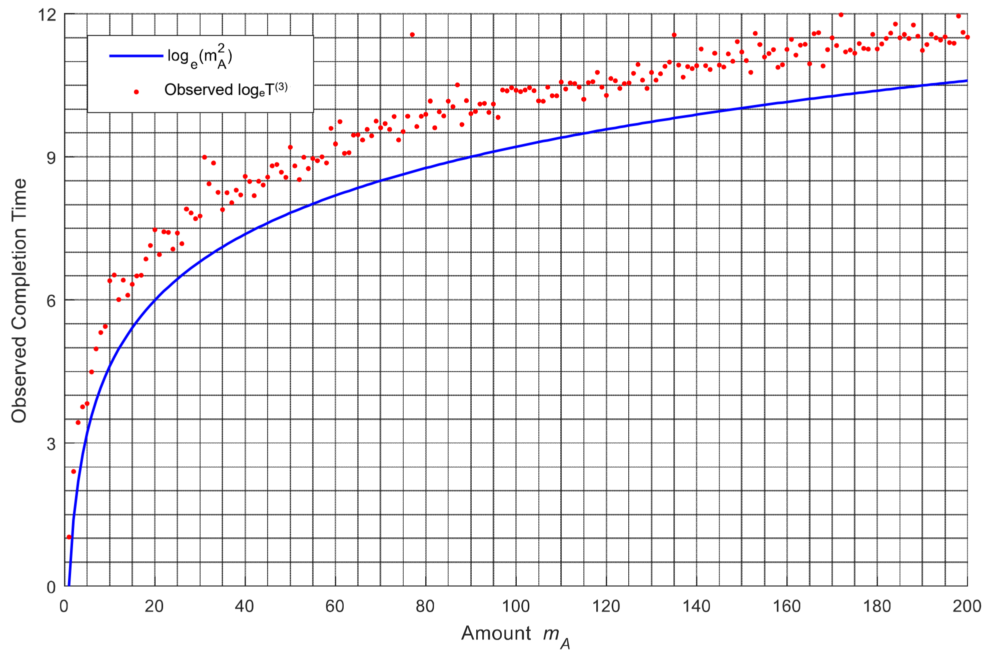

Figure 3: Growth of completion times when m_B = 1000×m_A, k = 3

Second test with $k \in \{4,5,6\}$ exhibited similar properties. The results in Figure 4 indicate that $\mathrm{E}[T^{(4)}], \mathrm{E}[T^{(5)}], \mathrm{E}[T^{(6)}] \in \Theta(\ m_A^2)$. This was easily discernible since the observed curves for $\ln(\mathrm{E}[T^{(4)}])$, $\ln(\mathrm{E}[T^{(5)}])$ and $\ln(\mathrm{E}[T^{(6)}])$ seem to be only a constant offset above $\ln(m_A^2)$. Note that in Figure 4 the curves for $\ln(\mathrm{E}[T^{(k)}])$ for higher values of k seem to be closer to their equivalent curves in Figure 2 i.e. as k increases, simulations tends to behave as if $m_A = m_B$.

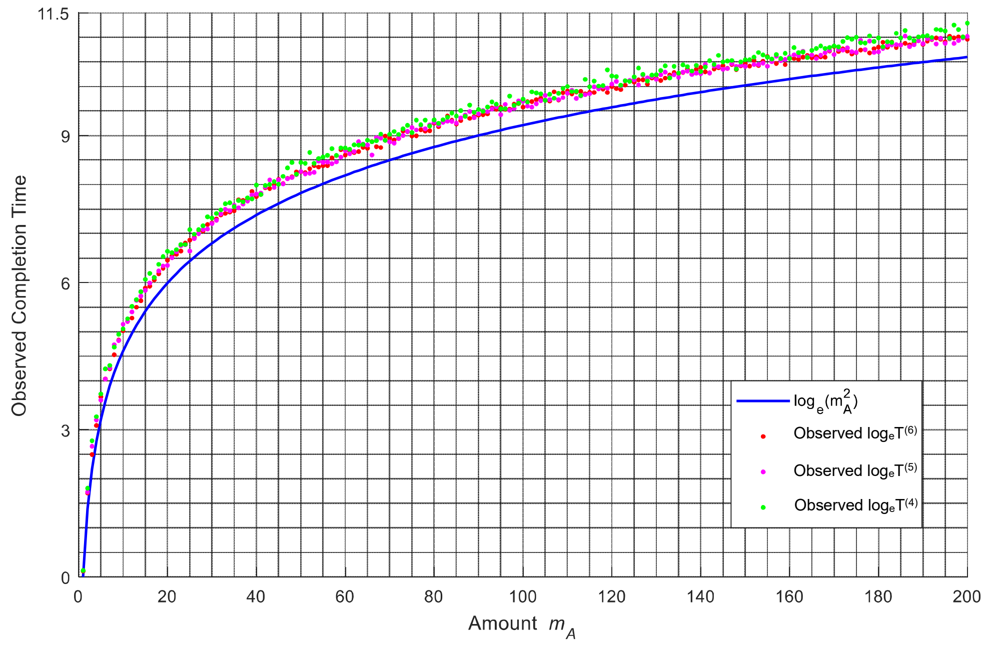

Figure 4: Growth of completion times when m_B = 1000×m_A, k = 4, 5, and 6

3.2 Running Average

The last two tests involved calculating running average of completion times in a simulation $G^{(k)}_{1,000,000}$.

We performed the test for $k \in \{2, 3, 4, 5, 6\}$ when $m_A = m_B$, and for $k \in \{3, 4, 5, 6\}$ when $m_B = 1000\times m_A$ (i.e. when $m_B \gg m_A$). A logarithmic scale has been used to properly present subtle simulation details.

3.2.1 When $m_A = m_B = 10$

Given that $m_A = m_B = 10$, calculations using general expressions for $\mathrm{E}[T^{(2)}]$, $\mathrm{E}[T^{(3)}]$ given in [1], [2] revealed that the running average of observed completion times in simulations $G^{(2)}_{1,000,000}$ and $G^{(3)}_{1,000,000}$ should eventually converge to 100, which is confirmed by Figure 5 below.

Additionally, it was observed that the running averages converge relative quickly i.e. around game number 10^4, all the running averages were quite close to their final values.

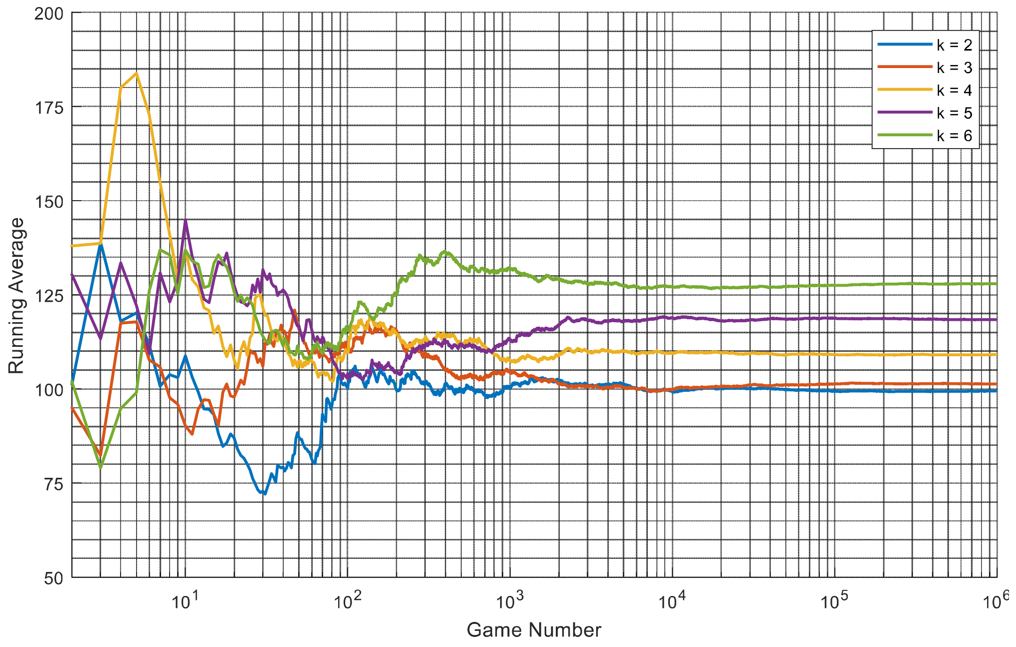

Figure 5: Running average of completion times when $m_B = m_A$, k = 2, 3, 4, 5, and 6

3.2.2 When $m_A = 10, m_B = 1000\times 10$

We used equation (4) with $m_A = 10$ to calculate that the running average for $T^{(3)}$ should converge around 299.401, which was confirmed by Figure 6.

One important observation regarding running average of $T^{(3)}$ was its high variance in Figure 6 when $m_B = 1000 \times m_A$, despite the relatively low variance in Figure 5 when $m_B = m_A$. This is consistent with the behaviour of $\mathrm{E}\left[T^{(3)}\right]$ in Figure 3. Moreover, the running average of $T^{(3)}$ converges nearly an order of magnitude slower in Figure 6 than in Figure 5.

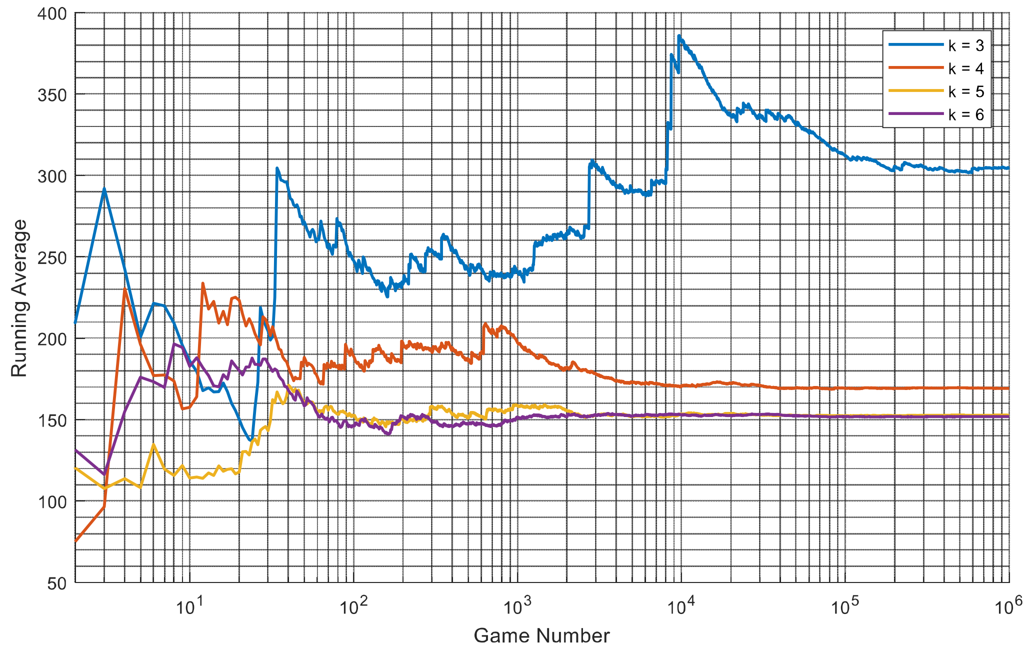

Figure 6: Running average of completion times when m_B = 1000×m_A, k = 3, 4, 5, and 6

4 Conclusions

From the simulation algorithm analysis in the report body, it was concluded that many optimization techniques can significantly boost performance of simulation algorithms. The most effective techniques for simulation algorithm optimization are vectorization, predicting resource usage, and pre-calculation of resources (instead of calculating them on-the-run). These techniques improve performance by effectively utilizing computer hardware and by reducing inefficiencies associated with branches, loops, and unpredictable resource usage. The “Massive Predictive Vectorization” algorithm used all of the mentioned optimization techniques, and was about 750% faster than the naïvely implemented version of the simulation algorithm. Additionally, the “Massive Predictive Vectorization” algorithm also satisfied the two constraints mentioned in Appendix A – to simulate $G_{10,000,000}^{(3)}$ within 150 seconds, and to produce the data-sets mentioned in Table 2.

From the simulation analysis aspect of the report body, we concluded that some simulation $G_r^{(k)}$ starting with all players having m as the starting amount of money can expect the completion time to be of order m^2 i.e. $\mathrm{E}[T^{(k)}] \in \Theta(m^2)$. Moreover, the same behaviour is observed in simulations where everyone starts with amount m_A, but the *k^th^* player starts with amount m_B ($m_B >> m_A$) – the completion time $\mathrm{E}[T^{(k)}]$ still exhibits $\Theta(m_A^2)$ growth. As k becomes larger, the simulations with $m_B >> m_A$ approach the behaviour of simulations having $m_A = m_B$, since the *k*th player becomes less likely to be chosen, and it becomes more likely for a game to terminate because of the ruin of players $1, 2, 3 \dots k-1$. Thus, though the completion time is random, we can estimate its growth rate given the amount of money players begin the game with.

5 Recommendations

For the simulation algorithm analysis portion of the report, we recommend:

- The "Massive Predictive Vectorization" simulation algorithm should be used to perform simulations for the "Gambler's Ruin" problem as it significantly outperformed the naïve simulation algorithm.
- The performance of simulation algorithms can still be considerably improved by using some additional optimization techniques like multithreading. Since each game within a simulation is completely independent of another game, the part of the algorithm responsible for simulating a game can be implemented as a multithreaded method to boost algorithm throughput. Parallelization can considerably improve the performance by allowing multiple simulations to occur simultaneously instead of simulating everything in a serial manner.
- Another technique for optimizing the simulation algorithms would be extending the concept of vectorization to the production of random numbers. The "Massive Predictive Vectorization" simulation algorithm spends considerable amount of time producing random numbers for generating a new sequence of gambles. The time spent on generating random numbers can be significantly reduced by pre-calculating them prior to beginning of the simulation. Although it is impossible to know the exact number of random numbers required for completing a simulation, it is possible to pre-calculate a large number of recyclable random numbers i.e. we reuse pre-calculated random numbers if needed. Instead of generating `expected_completion_time` number of random values again, we can simply generate a single random value to choose a vector of length `expected_completion_time` from the pre-calculated list of random numbers.

The following recommendations are made for the simulation analysis aspect of the report:

- In addition to analyzing the asymptotic behaviour and the running average of completion times, we can also study the empirical probability mass function of the completion times in a single simulation $G_r^{(k)}$. The empirical probability mass function can then be used to check the types of random variables that the completion time $T^{(k)}$ can be modelled as.
- Another possible approach to studying the behaviour of expected completion times in a simulation $G_r^{(k)}$ would be to model each player's starting wealth separately on a unique axis. This would transform the analysis to a $k + 1$ dimensional empirical optimization problem.

Glossary

Random Variable: a variable that attains random values in accordance with certain properties, and is generally assumes a value based on the result of an experiment.[1]

- **Discrete Random Variable:** a random variable that can only a attain value from a set of discrete values.
- **Continuous Random Variable:** a random variable that can attain any value from a set of continuous values.

Stochastic Process: A process or phenomenon whose behaviour is largely dependent upon multiple random variables. These processes are largely unpredictable due to their inherent randomness, but often have certain predictable features.[2]

Gambler's Ruin: A game with k people, where each of the k people start with a certain amount of money. In every round of the game, we randomly choose a pair from $\binom{k}{2}$ different possibilities, and randomly choose a winner from the chosen pair. The winner gains \$1, and the loser forgoes \$1. We continue the game till one of the players is left with \$0.[3]

Asymptotic Behaviour: Lower-bound, upper-bound, or tight-bound on the growth of number of operations in an algorithm as input size increases.

Vectorization: The process of reorganizing code to replace loops working individually on long data sets with fewer, coalesced operations.[4]

Exponential Averaging: An averaging technique that assigns lower weight to older values, and higher weight to newer values.

MATLAB: Matrix Laboratory; a software package used for numerical analysis, matrix manipulation, and symbolic mathematics, etc.[5]

Cache: Hardware component that provides fast access to frequently used portions of memory. Code with minimal cache-misses operates multiple orders of magnitude faster than something with a large cache-miss rate.

Pipelining: Computer architecture style allowing efficient instruction level parallelism within the same processor.

[1] S. Ross, A First Course in Probability, Ninth Edition, Boston, Massachusetts: Pearson, 2014
[2] Joseph L. Doob, *Stochastic Processes.* Charlottesville, VA, USA: Wiley, 1990.
[3] Yvik C. Swan and F. Thomas Bruss, "A Matrix-Analytic Approach to the N-Player Ruin Problem", *Journal of Applied Probability*, Vol. 43, No. 3, pp. 755-766, Sep., 2006.
[4] MATLAB, "Vectorization," MathWorks, 2017. [Online]. Available: https://www.mathworks.com/help/matlab/matlab_prog/vectorization.html. [Accessed 6 December 2017]
[5] "MATLAB - MathWorks", *Mathworks.com*, 2017. [Online]. Available: https://www.mathworks.com/products/matlab.html. [Accessed: 08- Dec- 2017].

Memory Fragmentation: Fragmentation of available memory into non-contiguous blocks which restricts contiguous allocation of large amounts of memory.

Branching: This includes '*if…then*', '*while*', and '*for*' statements in code, which often lead to inefficiencies in code when used improperly. These instructions cause the computers to choose a path of execution depending upon whether a certain condition is true or false.

WKRPT: Work-Term Report; an acronym used by the University of Waterloo Undergraduate Calendar.[6]

[6] D. W. Harder, "*Engineering report writing using Word 2010*", University of Waterloo, Waterloo, 2015.

References

[1] A. Alabert, M. Farr´e and R. Roy, "Exit Times from Equilateral Triangles," *Springer-Verlag,* vol. 49, no. 1, pp. 44-46, December 2004.

[2] A. Rocha, "The asymmetric n-player gambler's ruin problem with equal initial fortunes," *Advances in Applied Mathematics,* vol. 33, no. 3, p. 513, 2004.

[3] MATLAB, "Preallocation," MathWorks, 2017. [Online]. Available: https://www.mathworks.com/help/matlab/matlab_prog/preallocating-arrays.html. [Accessed 6 December 2017].

[4] J. Zarnett, "Lecture 26 - Scheduling Algorithms," University of Waterloo, 2017. [Online]. Available: https://github.com/jzarnett/ece254/blob/master/lectures/L26.pdf. [Accessed 6 December 2017].

[5] MATLAB, "Vectorization," MathWorks, 2017. [Online]. Available: https://www.mathworks.com/help/matlab/matlab_prog/vectorization.html. [Accessed 6 December 2017].

[6] D. A. Patterson, "Lecture 6: Vector Processing (Computer Science 252)," University of California, Berkeley, 1998. [Online]. Available: https://people.eecs.berkeley.edu/~pattrsn/252S98/Lec06-vector.pdf. [Accessed 6 December 2017].

Appendix A: Solution Requirements

We are required to find the following information about the "Gambler's Ruin" game:

1. A novel algorithm for simulating some game $G^{(k)}$, where $k \geq 2$ such that the asymptotic behaviour of the algorithm's runtime depends only on E$[T^{(k)}]$.
2. Analyzing the following empirical data related to random variable T_i^k, where integer $i = 1, 2, 3 \ldots, r$ (r is the number of games simulated):
   - Expected value E$[T^{(k)}]$ of random variable $T^{(k)}$.
   - Asymptotic behaviour of E$[T^{(k)}]$ with respect to change in money $m_1, m_2, \ldots, m_k$ that players start with.

The selected algorithm must satisfy the following conditions:

1. Wall-clock time constraint of 150 seconds for simulating $G^{(3)}_{10,000,000}$.
2. Simulate all the required sets of games with the given parameters.
   The following table expresses all the simulations that need to be performed. Assume that $m_1 = m_2 = \cdots = m_{k-1} = m_A$ and $m_k = m_B$.

Table 2: Sets of various simulations that need to be performed

Number of People	**Number of Simulations**	**Money Constraint**	**Type of Data**
$k \in \{2, 3, 4, 5, 6\}$	$r \in \{1, 2, 3 \ldots, 200\}$	$m_A \in \{1, 2, 3 \ldots, 200\}$ $m_B = m_A$	Asymptotic Analysis
$k \in \{3, 4, 5, 6\}$	$r \in \{1, 2, 3 \ldots, 200\}$	$m_A \in \{1, 2, 3 \ldots, 200\}$ $m_B = 1000 \times m_A$	Asymptotic Analysis
$k = \{2, 3, 4, 5, 6\}$	$r = 1{,}000{,}000$	$m_A = 10$ $m_B = m_1$	Running Average
$k = \{3, 4, 5, 6\}$	$r = 1{,}000{,}000$	$m_A = 10$ $m_B = 1000 \times m_A$	Running Average

Appendix B: Tools for Probabilistic Analysis

B.1 Probability Definitions, Notations

This table outlines the various terms and notations commonly used in this work-report.

Table 3: Notational conventions used in this report

Term	Notation	Definition	
Cumulative Distribution Function	$F_{\mathrm{X}}(x)$	$F_{\mathrm{X}}(x) = \mathrm{P}(\mathrm{X} \leq x)$	
Probability Mass Function	$f_{\mathrm{X}}(x)$	$f_{\mathrm{X}}(x) = \mathrm{P}(\mathrm{X} = x)$	
Notation	$\mathrm{X} \sim f_{\mathrm{X}}(x)$	"Random Variable X has a Probability Mass Function $\mathrm{f}_{\mathrm{X}}(x)$"	
Expected Value	$\mathrm{E}[\mathrm{X}]$	*Discrete Random Variable*	$\mathrm{E}[\mathrm{X}] = \int_{-\infty}^{\infty} x f_{\mathrm{X}}(x) dx$
		Continuous Random Variable	$\mathrm{E}[\mathrm{X}] = \sum_{-\infty}^{\infty} k f_{\mathrm{X}}(k)$
Variance	$\mathrm{Var}(\mathrm{X})$	$\mathrm{Var}(\mathrm{X}) = \mathrm{E}[\mathrm{X}^2] - (\mathrm{E}[\mathrm{X}])^2$	
"Gambler's Ruin" Game	$G^{(k)}$	Some game of "Gambler's Ruin" with *k* people	
"Gambler's Ruin" Completion Time	$T^{(k)}$	Random variable indicating completion time of a "Gambler's Ruin" with *k* people	
"Gambler's Ruin" Simulation	$G_r^{(k)}$	Set of *r* games of "Gambler's Ruin" with *k* people such that $G_r^{(k)} = \langle T_1^{(k)}, T_2^{(k)}, T_3^{(k)}, \dots, T_r^{(k)} \rangle$, where $T_i^{(k)}$ is the completion time of *i*th game in the simulation	
Starting Money of "Gambler's Ruin" Game's Players	$m_1, m_2, \dots, m_k$ for some game $G^{(k)}$	m_i is the amount of money the *i*th player started the game with	

B.2 Types of Random Variables

The following table outlines the different types of random variables frequently used in probability theory and their important properties.

Table 4: Properties of common random variables[7]

Type	Probability Density Function	Properties
Discrete Uniform Random Variable	$f_X(k) = \begin{cases} 0, & k \notin [a,b] \cap \mathbb{Z} \\ \frac{1}{b-a+1}, & k \in [a,b] \cap \mathbb{Z} \end{cases}$	$X \in [a,b] \cap \mathbb{Z}$ $E[X] = (a+b)/2$
Binomial Random Variable	$f_X(k) = \binom{n}{k} p^k (1-p)^{n-k}$	$X \in [0,n] \cap \mathbb{Z}$ $E[X] = np$
Poisson Random Variable	$f_X(k) = \begin{cases} 0, & k < 0 \\ e^{-\lambda} \frac{\lambda^k}{k!}, & k \geq 0 \end{cases}$	$X \in [0,\infty) \cap \mathbb{Z}$ $E[X] = \lambda$
Continuous Uniform Random Variable	$f_X(x) = \begin{cases} 0, & x \notin [a,b) \\ \frac{1}{b-a}, & x \in [a,b) \end{cases}$	$X \in [a,b)$ $E[X] = (a+b)/2$
Continuous Degenerate Random Variable	$f_X(x) = \delta(x-k) = \begin{cases} 0, & x \neq k \\ \infty, & x = k \end{cases}$	$X = k$ $E[X] = k$
Normal Random Variable	$f_X(x) = \frac{\exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right)}{\sqrt{2\pi}\sigma}$	$X \in (-\infty,\infty)$ $E[X] = \mu$
Exponential Random Variable	$f_X(x) = \begin{cases} 0, & x < 0 \\ \lambda e^{-\lambda x}, & x \geq 0 \end{cases}$	$X \in [0,\infty)$ $E[X] = 1/\lambda$

[7] S. Ross, *A first course in probability*. Boston: Pearson., 2014.

Appendix C: Miscellaneous Mathematical Tools

This table outlines the definitions and properties of some additional mathematical tools we utilize in the work-report.

Table 5: Summary of miscellaneous mathematical tools[8]

Tool	Definition	Properties
Dirac Delta Function	$\delta(x) = \begin{cases} 0, & x \neq 0 \\ \infty, & x = 0 \end{cases}$	$\delta(x) \in \{0, \infty\}$ $f(x) * \delta^{(k)}(x-h) = f^{(k)}(x-h)$
Unit Step Function	$\mathrm{u}(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$	$\mathrm{u}(x) \in \{0,1\}$
Interval Indicator	$\mathrm{I}[a,b](x) = \begin{cases} 0, & x \notin [a,b) \\ 1, & x \in [a,b) \end{cases}$	$\mathrm{I}[a,b](x) \in \{0,1\}$ $\mathrm{I}[a,b](x) = \mathrm{u}(x-a) - \mathrm{u}(x-b)$
Fourier Transform	$\mathrm{F}\{f(x)\}_{(\omega)} = \hat{f}(\omega) = \int_{-\infty}^{\infty} f(x)e^{-j\omega x}dx$	$\mathrm{F}^{-1}\{\hat{f}(\omega)\}_{(x)} = f(x)$ $\mathrm{F}\{f(x) * g(x)\}_{(\omega)} = \hat{f}(\omega) \cdot \hat{g}(\omega)$
Laplace Transform	$\mathcal{L}\{f(x)\}_{(s)} = \hat{f}(s) = \int_{0^-}^{\infty} f(x)e^{-sx}dx$	$\mathcal{L}^{-1}\{\hat{f}(s)\}_{(x)} = f(x) \cdot u(x)$ $\mathcal{L}\{f(x) * g(x)\}_{(s)} = \hat{f}(s) \cdot \hat{g}(s)$
Discrete Convolution	$f(x) * g(x) = \sum_{k=-\infty}^{\infty} f(k)g(x-k)$	$f(x) * g(x) = g(x) * f(x)$ $f(x) * g(x-k) = f(x-k) * g(x)$
Continuous Convolution	$f(x) * g(x) = \int_{-\infty}^{\infty} f(k)g(x-k)dk$	$f(x) * g(x) = g(x) * f(x)$ $f(x) * g(x-k) = f(x-k) * g(x)$

[8] B. Lathi, *Linear systems and signals*. New York: Oxford University Press, 2010.