

**Simulation of stochastic systems using  
antithetic multilevel Monte Carlo on GPUs**

by

Harold A. Lay, Jr.

A Dissertation Submitted in Partial Fulfillment  
of the Requirements for the Degree of  
Doctor of Philosophy in Computational Science  
Middle Tennessee State University

May 2020

**Dissertation Committee:**

Dr. Abdul Khaliq (Chair)

Department of Mathematical Sciences

Dr. Wandi Ding

Department of Mathematical Sciences

Dr. Zachariah Sinkala

Department of Mathematical Sciences

Dr. Anatoliy Volkov

Department of Chemistry

## DECLARATION

I declare that this dissertation has not been submitted at any other university for any degree. The results presented in this dissertation are published or accepted by the following refereed journals:

### *Published*

H. Lay, Z. Colgin, V. Reshniak, and A. Khaliq, “On the implementation of multilevel Monte Carlo simulation of the stochastic volatility and interest rate model using multi-GPU clusters,” *Monte Carlo Methods and Applications*, vol. 24, no. 4, pp. 309-321, 2018.

### *Accepted but not published*

H..Lay, “Heavy Equipment Demand Forecasting Using GPU Computing,” *Proceedings of the 2017 Federated Conference on Computer Science and Information Systems*, IEEE, accepted June 28, 2017.

## **DEDICATION**

Dedicated to my wife Jeannettea and our children Victoria,  
Zachary, Daniel, and Heather for their love and support.

## ACKNOWLEDGEMENTS

I wish to thank my advisor Dr. Abdul Khaliq for his guidance, encouragement, and patience over the last 9.3 years during the pursuit of my Ph.D. Without it, this work would not have been possible. I would like to thank Dr. John Wallin, Director of the Computational Science program, for his support. I am grateful for the valuable time my dissertation committee Dr. Wandi Ding, Dr. Zachariah Sinkala, and Dr. Anatoliy Volkov spent reviewing this research and their feedback.

I would like to thank my employer Thompson Machinery. Without the use of their computer resources and data much of this research could not have been performed. I am grateful for the encouragement provided by Jim Ezzell, Eric Newsom, and Randy Oden while struggling to balance a full-time job and graduate student.

Most of all I wish to acknowledge the love and support that my family provided for nearly a decade during my academic pursuit. I want to thank my wife Jeannettea for her love, support, and encouragement. I want to thank my children Victoria, Zachary, Daniel, and Heather for sacrificing time with their father and occasionally proofreading my homework while I checked their homework.

# ABSTRACT

Numerous natural processes ranging from chemical reactions to wind speed can be modeled using systems of stochastic differential equations (SDEs) [1]. Like ordinary differential equations (ODEs), stochastic differential equations model the rate of change of a variable in relation to one or more variables. However, SDEs differ from ODEs in that they contain a term that introduces an element of randomness to a normally deterministic process. Because of this uncertainty, only the simplest SDEs have analytical solutions, and one must turn to computational simulations in order to approximate their solutions. [2]

Monte Carlo methods are a common and useful means of solving systems of SDEs. The SDE is simulated by taking small steps through time or space until the terminating condition is met - either the final time or distance. This process is repeated numerous times after which the mean and variance of the solutions obtained are calculated in order to estimate the confidence interval containing the true value. [3]

When there are multiple processes involved (e.g., multiple assets in a financial portfolio or multiple ports on a server), there can be correlation among the individual processes. Unfortunately simulating numerous processes is computationally expensive unless advanced computational and numeric methods are employed. The goal of this dissertation is to propose a high performance computational framework for solving systems of stochastic differential equations with correlation in finance that utilizes

antithetic multilevel Monte Carlo on a cluster of GPU-enabled servers.

The dissertation includes the following sections. In chapter one, an introduction to stochastic differential equations is provided along with relevant background. Next, various computing architectures are reviewed. The third chapter introduces the concept of Monte Carlo simulations and some of its variants including antithetic multilevel Monte Carlo. The previously discussed topics are applied to the area of financial options in chapter four. Forecasting, specifically exponential smoothing in a high performance environment, is covered in chapter five. The final chapter summarizes the results and provides an overview of future research.

# TABLE OF CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Introduction . . . . .	1
1.2	Stochastic Differential Equations . . . . .	3
1.2.1	Introduction . . . . .	3
1.2.2	Euler-Maruyama Discretization . . . . .	4
1.2.3	Milstein . . . . .	7
1.3	Conclusion . . . . .	9
<b>2</b>	<b>Computing Architectures</b>	<b>11</b>
2.1	Introduction . . . . .	11
2.2	Multicore . . . . .	12
2.3	MPI . . . . .	14
2.4	GPU . . . . .	16
2.4.1	NVIDIA CUDA . . . . .	19
2.4.2	OpenCL . . . . .	20
2.5	Conclusion . . . . .	20
<b>3</b>	<b>Multilevel Monte Carlo</b>	<b>23</b>
3.1	Introduction . . . . .	23
3.2	Monte Carlo . . . . .	24
3.3	Multilevel Monte Carlo . . . . .	28

3.4	Antithetic Multilevel Monte Carlo . . . . .	32
3.5	Alternative Monte Carlo Methods . . . . .	36
3.6	Conclusion . . . . .	36
<b>4</b>	<b>Applications in Finance</b>	<b>38</b>
4.1	Option Pricing . . . . .	38
4.1.1	Stochastic Volatility Model . . . . .	38
4.1.2	Stochastic Volatility, Stochastic Interest Rate Model . . . . .	39
4.2	Implementation . . . . .	41
4.3	Validation . . . . .	42
4.4	Results . . . . .	46
4.5	Conclusion . . . . .	47
<b>5</b>	<b>Applications in Inventory Management</b>	<b>50</b>
5.1	Introduction . . . . .	50
5.1.1	Motivation . . . . .	50
5.2	Exponential Smoothing Models . . . . .	52
5.2.1	Single Exponential Smoothing . . . . .	55
5.2.2	Double Exponential Smoothing . . . . .	59
5.2.3	Triple Exponential Smoothing . . . . .	63
5.3	Implementation . . . . .	70
5.4	Validation . . . . .	74
5.5	Results . . . . .	75
5.6	Conclusion . . . . .	79
<b>6</b>	<b>Conclusion</b>	<b>81</b>
6.1	Introduction . . . . .	81
6.2	Significant Contribution . . . . .	83



6.3	Future Work . . . . .	84
-----	-----------------------	----

# LIST OF FIGURES

1.1	Number of Google Scholar results for the phrase <i>stochastic differential equation</i> . . . . .	5
1.2	Example of five random samples from the equation 1.3 with $\alpha_t = rSdt$ and $\beta_t = \sigma S$ for $0 \leq t \leq 1$ with 50 time steps and $dt = 0.02$ . The bold line is the average of the five samples. . . . .	7
1.3	Impact of varying $dt$ on the estimation of mean and standard deviation for $dS_t = rS_tdt + \sigma SdW_t$ . . . . .	8
2.1	Specifications for the seven node GPU-enabled cluster used in the research . . . . .	17
3.1	Example of five random samples from the equation $dS = rSdt + \sigma SdW$ for $0 \leq t \leq 1$ with 50 time steps and $dt = 0.02$ . The bold black line is the average of the five samples, and the bold red line is the analytical solution. . . . .	26
3.2	Weak error caused by the discretization of the sample problem in relation to the step size $dt$ using $10^8$ simulations. . . . .	27
3.3	Multilevel Monte Carlo . . . . .	29
3.4	Antithetic multilevel Monte Carlo process . . . . .	33

4.1	Comparison of runtimes for the stochastic volatility, stochastic interest rate model for different platforms and Monte Carlo methods using a single-threaded CPU and a single NVIDIA K20 GPU [4] . . . . .	45
4.2	Performance improvement of MLMC and GPU over CPU-based MC [4]	45
4.3	Impact of error control variable $\epsilon$ on the computational cost of the stochastic variable, stochastic interest rate simulation [4] . . . . .	46
4.4	Performance improvement of Milstein discretization over Euler-Murayama for multilevel Monte Carlo [4] . . . . .	47
4.5	Runtime of the multilevel Monte Carlo for varying values of the error control variable $\epsilon$ . . . . .	48
5.1	Time series data exhibiting trend, seasonality, and error. . . . .	53
5.2	Normalized industry deliveries from January 2010 through December 2012 with linear regression. . . . .	53
5.3	Seasonality factor for each period shown in Figure 5.2. . . . .	54
5.4	Average seasonality factor by period shown in Figure 5.3. . . . .	54
5.5	Fit of single exponential algorithm to actual data with $\alpha = 0.1$ . . . .	56
5.6	Error versus $\alpha$ for single exponential smoothing . . . . .	57
5.7	The single exponential smoothing forecast during expansion period. .	58
5.8	The single exponential smoothing forecast during contraction period.	59
5.9	Double exponential smoothing. . . . .	60
5.10	The double exponential smoothing forecast during expansion period. .	62
5.11	The double exponential smoothing forecast during contraction period.	63
5.12	Double exponential smoothing error versus $\alpha, \gamma$ . . . . .	64
5.13	Error versus $\alpha, \gamma$ . . . . .	65
5.14	Triple exponential smoothing . . . . .	66
5.15	$\min(\log(\text{MSE}))$ for $\alpha$ and $\gamma$ . . . . .	67

5.16	$\min(\log(\text{MSE}))$ for $\beta$ and $\gamma$ . . . . .	68
5.17	Triple exponential smoothing forecast during expansion period . . . .	69
5.18	Triple exponential smoothing forecast during contraction period . . .	69
5.19	Actual data used in the example taken with permission from Thompson Machinery's industry activity . . . . .	70
5.20	Components of triple exponential smoothing . . . . .	72
5.21	Results of triple exponential smoothing on subset of data . . . . .	72
5.22	GPU Launch for triple exponential smoothing showing different threads performing the algorithm over different periods of time and data sets	73
5.23	Data structure for forecast results. The rows are products, the columns are the last period of the $2N$ cycles used to create the forecast, and the slices are the number of periods forecast in advance. . . . .	76
5.24	The forecast analysis dashboard created with Microsoft Power BI for analyzing product forecasts . . . . .	77
5.25	The forecast analysis dashboard created with Microsoft Power BI for analyzing accuracy for specific time periods . . . . .	78
5.26	MAPE for forecasts based on data from January 2000 through June 2018 for product level and aggregated categories. . . . .	78

## LIST OF TABLES

5.1	An example of single exponential smoothing for a subset of the data shown in Fig. 5.2. . . . .	56
5.2	Forecasting using the single exponential smoothing . . . . .	57
5.3	An example of double exponential smoothing for a subset of the data shown in Fig. 5.2. . . . .	61
5.4	Forecasting using the double exponential smoothing . . . . .	61
5.5	An example of triple exponential smoothing for a subset of the data shown in Fig. 5.2. . . . .	66
5.6	Forecasting using triple exponential smoothing . . . . .	67
5.7	Performance results for double precision forecasting . . . . .	79

## LIST OF ABBREVIATIONS

<b>CISC</b> Complex Instruction Set Computing	<b>MPI</b> Message Passing Interface
<b>CPU</b> Central Processing Unit	<b>ODE</b> Ordinary differential equation
<b>CUDA</b> Compute Unified Device Architecture	<b>OpenACC</b> Open Accelerators
<b>FLOPS</b> Floating Operations Per Second	<b>OpenCL</b> Open Computing Language
<b>Gbps</b> Gigabits per second	<b>OpenMP</b> Open Multi-Processing
<b>GPU</b> Graphics Processing Unit	<b>PDE</b> Partial differential equation
<b>HPC</b> High Performance Computing	<b>POD</b> Penguin On Demand
<b>IoT</b> Internet of Things	<b>POSIX</b> Portable Operating System Interface
<b>MC</b> Monte Carlo	<b>QPU</b> Quantum Processing Unit
<b>MCMC</b> Markov Chain Monte Carlo	<b>RISC</b> Reduced Instruction Set Computing
<b>MLMC</b> Multilevel Monte Carlo	<b>SDE</b> Stochastic differential equation
	<b>SIMD</b> Single Instruction, Multiple Data

# CHAPTER I: INTRODUCTION

## 1.1 Introduction

Modeling of complex systems such as cybersecurity attacks, crude oil prices, financial portfolios, and windspeed for renewable energy is challenging due to the lack of analytical solutions and the needed computational resources. Simulations are a useful method of approximating complex systems when analytical solutions are unavailable. These systems can range from multidimensional Black-Scholes partial differential equations when pricing a financial option [5] [6] to systems of stochastic differential equations (SDEs) used to model wind speed [7] and cybersecurity attacks [8]. Monte Carlo methods are powerful but relatively easy to implement tools to estimate solutions to complex systems when analytical solutions are unavailable. For large systems these simulations can become computationally expensive, and it becomes necessary to leverage numerical methods and high performance computing environments to improve performance.

In this research, Monte Carlo techniques are applied to the problem of modeling financial options involving both stochastic volatility and interest rate using a GPU-enabled clustered. This problem is challenging due to the complexity of the system, especially if extended beyond a single asset to a portfolio of assets.

This chapter provides a brief introduction to stochastic differential equations. Discretization methods used to solve continuous functions are reviewed. The issues

of stability and convergence are also addressed.

Computer architectures are reviewed in chapter two. Fundamental concepts of multicore computing is introduced, and the advantages and disadvantages are presented. These concepts are extended from a single system to clusters, and the application of the Message Passing Interface (MPI) to this environment is discussed. Finally, high performance computing (HPC) using graphics processing units (GPUs) is introduced.

In chapter three, the reader is introduced to Monte Carlo methods, specifically multilevel Monte Carlo. A comparison between the different methods of multilevel Monte Carlo, antithetic multilevel Monte Carlo, quasi Monte Carlo, and Markov chain Monte Carlo. The motivation behind our selection of antithetic Monte Carlo is reviewed.

With the fundamentals covered, chapter four of the dissertation applies them to the problem of pricing options in the field of finance. Specifically, the problem involves both stochastic interest rate and stochastic volatility model which includes a system of three SDEs: price, volatility, and interest rate. This system is solved using an antithetic multilevel Monte Carlo simulation on a cluster of nodes each equipped with two NVIDIA Tesla K40 GPUs.

In chapter five, the reader is provided with an introduction to the problem of time series forecasting as applied to the problem of inventory management, and the topic of exponential smoothing is discussed. Exponential smoothing is implemented on GPUs for the inventory management problem to demonstrate the capability for real-time forecasting of large inventories.

Chapter six provides a summary of the contributions of the dissertation and an overview of problems in other fields to which this work can be applied.



## 1.2 Stochastic Differential Equations

### 1.2.1 Introduction

Stochastic ordinary differential equations (SDE's) resemble ordinary differential equations (ODE's) in that they model the rate of change of a variable in relation to other variables include a stochastic term driven by Brownian motion [3] [9]. One of the simplest applications of ODE's learned is the standard equation of a line  $y = mx + b$  where  $dy/dx = m$  (though most students do not see it in this form until many years later). The general form of an ODE can be expressed as [2]

$$dx/dt = f(t, x(t)), x(0) = x_0 \quad (1.1)$$

This form shows that the rate of change of  $x$  with respect to time (i.e.,  $dx/dt$ ) is determined by the value of  $x$  at time  $t$  and the rate of change in  $t$  (i.e.,  $dt$ ). This deterministic equation always produces the same values for  $x$  if the initial parameters are reused. However, SDE's introduce a new term to include an element of randomness in the form of Brownian motion, a phenomena found in many natural and artificial models [3]. The general form of an SDE can be written as [3], [2]

$$dx = f(t, x(t, \omega))dt + g(t, x(t, \omega))dW(t, \omega), x(0) = x_0 \quad (1.2)$$

where  $f(t, x(t, \omega))$  is the *drift* coefficient and  $g(t, x(t, \omega))$  is the *diffusion* coefficient.  $x(t, \omega)$  is the value of  $x$  at time  $t$  given the series of random noise  $w$ .

The key difference between ordinary differential equations and stochastic differential equations is the Brownian motion term  $dW(t, \omega)$  [2], also known as the noise term. The noise term is a random process following the normal distribution with a mean of zero and a variance of  $dt$  often expressed as  $N(0, dt)$  [3]. Even with identical initial

values for an SDE, the values of  $x$  will be unique due to the random term.

Stochastic differential equations have applications in a variety of fields including population models in biology [10], [11], [12], chemical reactions [13], sun spot activity in astrophysics [14], pricing of options in finance [5], macroeconomics [15], cybersecurity attacks [16], and windspeed models in the area of renewable energy [17].

SDEs have found numerous applications in the field of biology. In [18] a system of SDEs was applied to the swarming behavior of fish foraging. Two equations for each individual in the group described position and velocity in a two-dimensional space. Predator-prey systems were modeled in [10] as a series of stochastic differential equations to show the impact of Brownian noise on models.

In [19], sunspot activity was modeled using a system of two stochastic differential equations. The first equation modeled the rate at which sunspots formed while the second equation modeled the number of sunspots active at a specific time  $t$ . The system was based on data collected from three eleven year solar cycles ranging from 1943 to 1975 while selecting the 2500 days around the peak of each cycle to minimize noise in the data.

The area of stochastic differential equations is active due to the numerous fields to which they can be applied. Over the last seven years, there have been an average of 43000 papers per year generated on the topic based on data from Google Scholar (Fig. 1.1). By September 2019, there were 28000 results for the year 2018 using the search phrase “stochastic differential equations”.

### 1.2.2 Euler-Maruyama Discretization

Complex stochastic differential equations and systems of SDEs often lack analytical solutions and require other methods such as simulations to approximate their solutions. The continuous equation 1.2 must be discretized into time steps that yield results

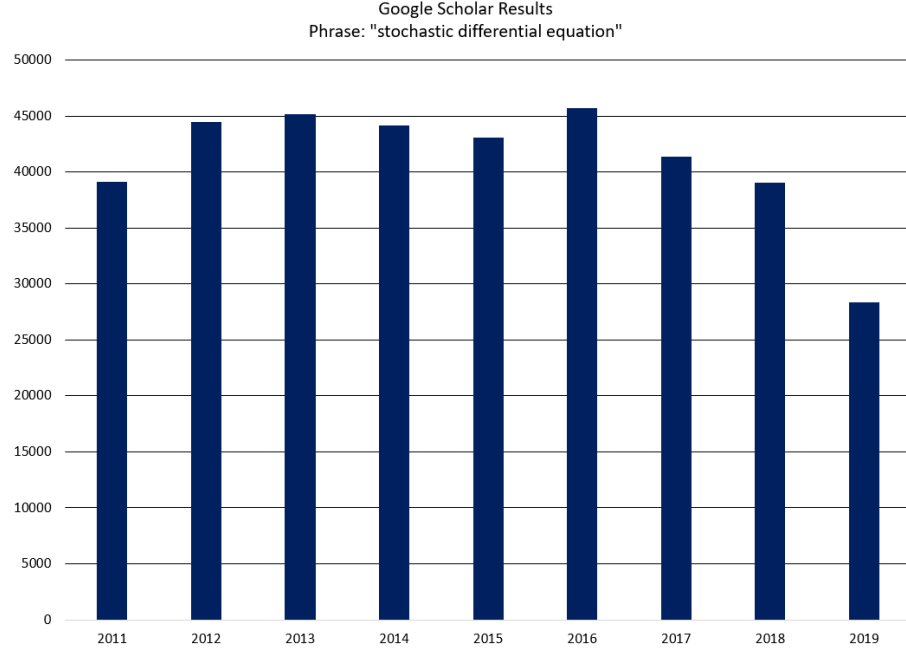


Figure 1.1: Number of Google Scholar results for the phrase *stochastic differential equation*

with minimal error. Let  $X_t$  be the value of the SDE at time  $t$ , the drift coefficient  $\alpha_t(X_t) = f(t, X(t, \omega))$ , and the diffusion coefficient  $\beta_t(X_t) = g(t, X(t, \omega))$ ,  $dW_t = \sqrt{dt} \times N(\mu = 0, \sigma = 1)$ , then:

$$dX_t = \alpha_t(X_t)dt + \beta_t(X_t)dW_t \quad (1.3)$$

The Euler-Maruyama method [20] can be easily implemented in applications with a single SDE as shown in Algorithm 1. The extension to a system of SDEs is straightforward and simply requires a loop over each equation inside the time loop. The implementation requires the availability of a random number generator that can produce normally distributed values for the Brownian motion term  $dW$  [2]. Such functions are often found in libraries such as the C++ `random` library.

Fig. 1.2 shows the results of five samples from equation 1.3 with the constants  $\alpha_t = rSdt$  and  $\beta_t = \sigma S$  and the step size  $dt = 0.02$ . The red dashed line shows the

---

**Algorithm 1** Euler-Maruyama Discretization Algorithm

---

```

1: {Define variables}
2:  $T = \{\text{End of simulation}\}$ 
3:  $Steps = \{\text{Number of time steps}\}$ 
4:  $dt = T/Steps$ 
5:  $X = X_0$ 
6:
7: {Define functions}
8:  $alpha(X) = \{\text{Drift function}\}$ 
9:  $beta(X) = \{\text{Diffusion function}\}$ 
10:
11: for  $n = 1$  to  $Steps$  do
12:    $dW = normal\_random(mean = 0.0, variance = 1.0) * sqrt(dt)$ 
13:    $dX = alpha(X)dt + beta(X)dW$ 
14:    $X = X + dX$ 
15: end for

```

---

analytical solution  $S = S_0 e^{rt}$  while the bold black line shows the mean of the five samples at each of the time steps, which oscillates around the analytical solution. Because  $dW$  is taken from a normal random distribution, then  $\mathbb{E}[\sigma S dW] = 0.0$ , and  $\mathbb{E}[r S dt] = S_0 e^{rt}$ .

The three graphs in Fig. 1.3 show the effect of varying the time step upon the mean and standard deviation of the results.  $dt$  was varied from  $4.6 \times 10^{-4}$  to 1.0, and the simulation was run 1000 times. Figures 1.3a displays the mean of  $\mathbb{E}[X]$  with a 95% confidence interval, and Fig. 1.3b shows the standard deviation. The results were generated by performing 1000 samples simulation for 1000 different time steps. The source code can be found at <https://github.com/jjlay/Dissertation/>.

The Euler-Maruyama method is a simple method to implement. However it is not numerically efficient with a strong order of convergence  $\gamma = \frac{1}{2}$  and a weak order of convergence of  $\beta = 1$  [2] [21]. When implemented within a Monte Carlo method, other techniques such as antithetic methods and multilevel Monte Carlo can be employed to improve performance of the Euler-Maruyama discretization.

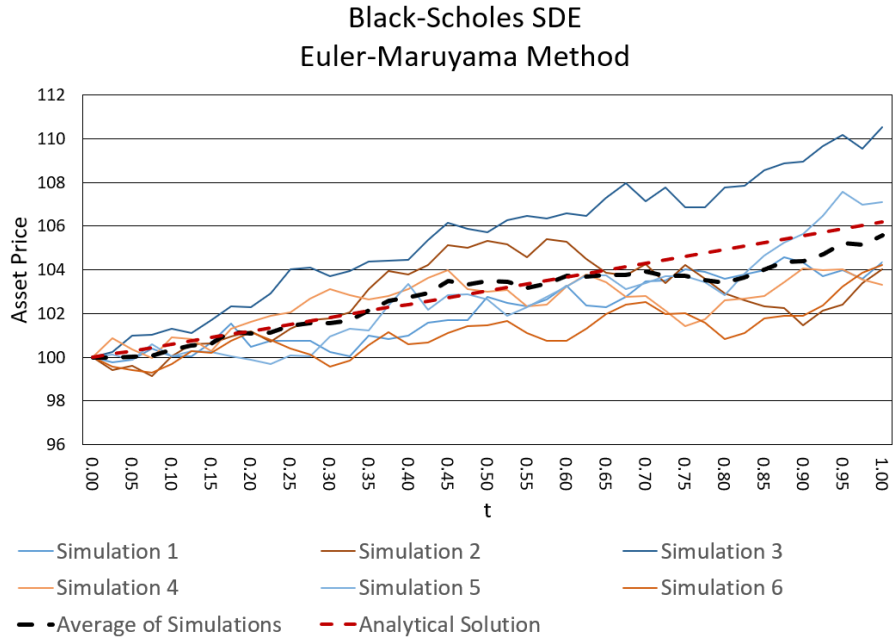


Figure 1.2: Example of five random samples from the equation 1.3 with  $\alpha_t = rSdt$  and  $\beta_t = \sigma S$  for  $0 \leq t \leq 1$  with 50 time steps and  $dt = 0.02$ . The bold line is the average of the five samples.

### 1.2.3 Milstein

The order of convergence for the Euler-Maruyama method can be improved through inclusion of an additional term from the Itô-Taylor expansion [2].

$$dX_t = \alpha_t(X_t)dt + \beta_t(X_t)dW_t + \frac{1}{2}\beta_t(X_t)\beta'_t(X_t)(dW_t^2 - dt) \quad (1.4)$$

The additional function  $\beta'(X_t)$  is the derivative of the diffusion function with respect to  $X$ . When  $\beta_t(X_t)$  is a constant or has another mathematical form that does not contain  $X$ , then  $\beta'_t(X_t) = 0$  and equation 1.4 reduces to the Euler-Maruyama equation 1.3. This is referred to as *additive noise*. SDEs that contain  $X_t$  within the diffusion function (and thus the derivative is non-zero) are described as having *multiplicative noise*. However, when implemented for higher dimension problems, the Milstein discretization includes the Lévy area, cross-derivative terms [2].

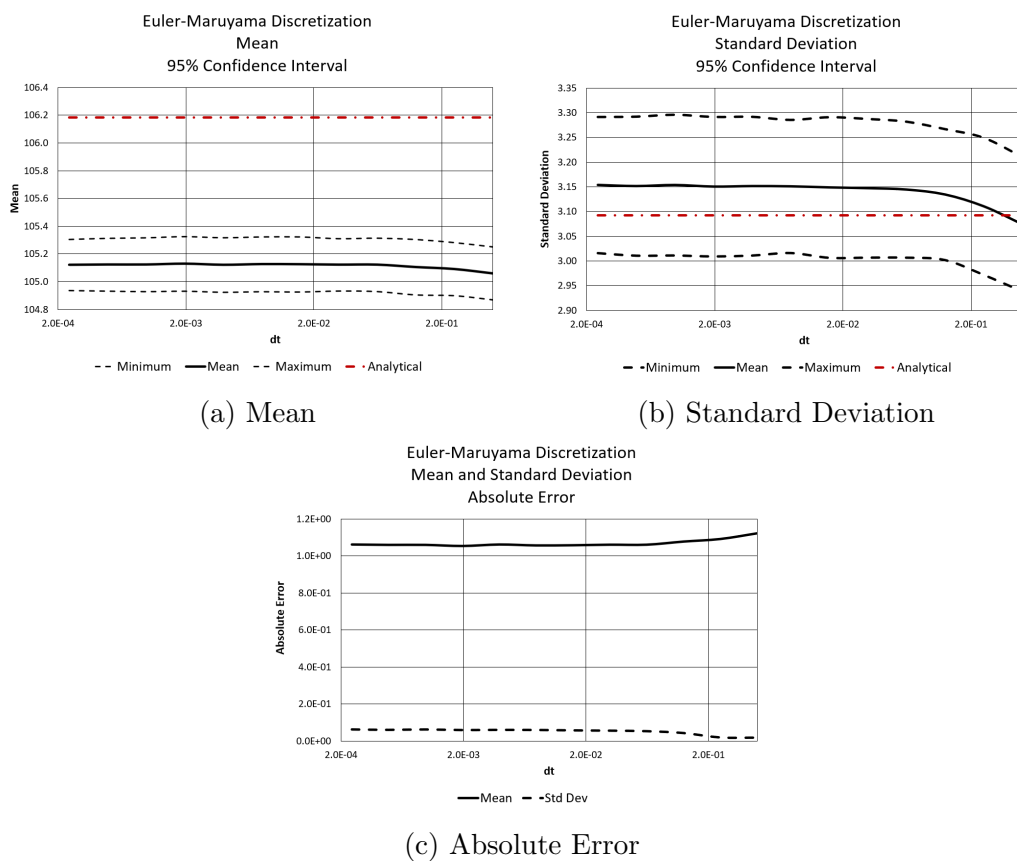


Figure 1.3: Impact of varying  $dt$  on the estimation of mean and standard deviation for  $dS_t = rS_t dt + \sigma S dW_t$

The pseudocode in algorithm 2 outlines the implementation of the Milstein algorithm. For a single SDE, the code varies only from algorithm 1 by the additional term.

---

**Algorithm 2** Milstein Discretization Algorithm

---

```

1: {Define variables}
2:  $T = \{\text{End of simulation}\}$ 
3:  $Steps = \{\text{Number of time steps}\}$ 
4:  $dt = T/Steps$ 
5:  $X = X_0$ 
6:
7: {Define functions}
8:  $alpha(X) = \{\text{Drift function}\}$ 
9:  $beta(X) = \{\text{Diffusion function}\}$ 
10:  $beta'(X) = \{\text{Derivative of diffusion function}\}$ 
11:
12: for  $n = 1$  to  $Steps$  do
13:    $dW = normal\_random(mean = 0.0, variance = 1.0) * sqrt(dt)$ 
14:    $dX = alpha(X)dt + beta(X)dW + 0.5 * beta(X) * beta'(X) * ((dW * dW) - dt)$ 
15:    $X = X + dX$ 
16: end for

```

---

The Milstein method has the advantage of a strong order of convergence  $\gamma = 1$  compared to the Euler-Murayama method of  $\gamma = 0.5$  due to the additional term [2]. However, at higher dimensions, the stochastic double integral Lévy area must be approximated. This can be avoided under the special condition when the noise is diagonal reducing the discretization to 1.4 [3]. In systems without diagonal noise, the antithetic multilevel Monte Carlo method (discussed in Section 3.4) eliminates the necessity to calculate the Lévy areas [21].

## 1.3 Conclusion

In this chapter we introduced stochastic differential equations which are a class of differential equations containing an element of randomness controlled by Brownian noise. It is shown that this noise can take the form of *additive noise* such that the

randomness is not dependent on the current quantity of the variable of interest. With *multiplicative noise*, the variable is included in the diffusion equation. Few types of SDEs have analytical solutions requiring other techniques to estimate their solution. This requires a means of converting a continuous problem into discrete steps that can be simulated.



## CHAPTER II: COMPUTING ARCHITECTURES

### 2.1 Introduction

High performance computing is a broad term to encompass environments beyond traditional desktop computing which can include clusters of servers and supercomputers. These systems can include *complex instruction set computing* (CISC) processors with large instruction sets, *reduced instruction set computing* (RISC) processors, or a combination of the two. Processors can be the computer's main CPU or a co-processor such as a *graphics processing unit* (GPU) or a *quantum processing unit* (QPU).

Since 1993, the TOP500 list provides performance and architectural information about the fastest supercomputers in the world. In addition to performance, the manufacturers, networking infrastructure, and software is summarized. 367 of the systems consist of clusters powered solely by CPUs without the aid of coprocessors such as GPUs. Seven of the systems use IBM's Power 9 RISC processors while the remaining rely of CISC processors such as those manufactured by AMD and Intel [22].

One common attribute shared among the systems is the mixture of local multiprocessing and distributed computing. Multiprocessing is the use of multiple processors or cores within a physical server while distributed computing leverages processors located in multiple servers connected over a high speed network connection.

## 2.2 Multicore

Computer processors (CPUs) have evolved to be smaller and have a higher performance/cost ratio as a result of improved design and manufacturing. Low-cost computers such as the Raspberry Pi 4 are equipped with quad-core processors, and mobile devices such as cell phones now have octa-core CPUs. Developers can utilize multiple threads to improve the performance of their code even on the smallest of the devices.

Multicore programming has a distinct advantage over other parallel paradigms such as message passing due to the use of shared memory, and thus programs perform better for applications that require data to be exchanged among threads. Consider a simple one dimension grid on which a single thread is assigned to a single element that is updated during each time step. Since the entire grid is maintained in a memory range shared among the threads, the result from neighboring threads is immediately available at the beginning of the next step. In distributed environments such as message passing, data must be exchanged between steps introducing a communication overhead not found in the multicore paradigm [23].

Multicore programming is limited by the number of processors that can be fit on a single motherboard and the number of cores that can be placed within a processor. As of October 2019, Intel<sup>©</sup> Xeon<sup>©</sup> Platinum 9282 processor had 56 cores [24] while AMD Ryzen<sup>™</sup> Threadripper processors had a maximum of 32 cores [25].

Two commonly used multicore programming libraries are POSIX threads (*pthread*s) and OpenMP. Pthreads is a library found on Unix-style operating systems that provides developers with the ability to launch and control multiple threads. Its functionality is at a low level and thus provides more granular control for the programmer at a cost of complexity. There are functions for semaphores, locks, and synchronization. The programmer must create and manage these resources [26].

OpenMP, in contrast, is a directive-based library that attempts to isolate the programmer from the low-level control of pthreads. Code can be written single-threaded and tested, then the developer adds compiler directives to indicate sections that can be performed in parallel. In addition, there are directives to indicated variables to be shared among threads as well as those that are local. As its name implies, OpenMP code is portable among hardware platforms and operating systems. Microsoft's Visual Studio development environment supports OpenMP directives allowing code written on other systems such as MacOS to be compiled and executed [27].

While most CPUs now have multiple cores and multicore programming can be accomplished using readily available libraries, performance is restricted by physical limitations such as physical space on the motherboard, heat, and power. Scaling, therefore, must be accomplished by the addition of servers connected over a high-speed network such as Ethernet or Infiniband.

---

**Algorithm 3** Multicore Monte Carlo method

---

```

1: {Define variables}
2:  $N = \{\text{Number of simulations}\}$ 
3:  $Q = \{\text{Number of threads}\}$ 
4:  $n = \lceil N/Q \rceil$  {Number of simulations per thread}
5:  $Sum[Q] = 0$  {Storage for each thread to store results}
6:
7: {Launch threads}
8: for  $q = 1$  to  $Q$  do
9:    $pthread\_create(Simulate(n, \&Sum[q]))$ 
10: end for
11:
12:  $pthread\_join()$ 
13: for  $q = 1$  to  $Q$  do
14:    $Result = Result + Sum[q]$ 
15: end for
16:
17: {Note that the denominator contains  $n$  rather than  $N$ . Since  $n$  is calculated using
    the ceiling function,  $Q \times n$  may be larger than  $N$ .}
18:  $Result = Result / (Q \times n)$ 

```

---

## 2.3 MPI

Communication between servers involves a complex process of exchanging messages and synchronizing data. While operating systems provide a means of performing these functions, they are low-level interfaces requiring the developer to perform routine housekeeping tasks. This in turn forces the developer to create custom libraries reducing the portability of code.

The Message Passing Interface (MPI) was developed to provide a standard set of application programming interfaces (APIs) to programmers that allowed portability between disparate systems. Code written using MPI could be compiled on Linux, Unix, or even Microsoft Windows (assuming no operating system specific calls were used). There are over 100 functions in the MPI specification providing memory movement, thread synchronization, and process communication interfaces that eliminate the need of the programmer to develop this low-level functionality [23].

Besides portability, MPI insulates the developer from the underlying hardware to some extent. Calls among threads are performed on a multiprocessor, shared memory system in the same manner as calls among threads running on different servers. Performance can be improved by leveraging calls optimized for the architecture being used such as the CUDA-aware Open MPI implementation [28].

Monte Carlo simulations are ideal candidates for MPI as each simulation is independent of the others and the communication between the nodes occurs at the beginning when parameters are exchanged and at the end when the results are aggregated. Each thread works without the need for synchronization. Algorithm 4 details the process for implementing a Monte Carlo simulation using MPI. A hybrid implementation using distributed computing and multiprocessing can be used for applications where there is parallelism within the simulation such as solving along a grid.

---

**Algorithm 4** MPI Monte Carlo method

---

```

1: {Define variables}
2:  $myID = MPI\_THREAD\_ID$ 
3:  $N = \{\text{Number of simulations}\}$ 
4:  $Q = \{\text{Number of MPI worker nodes}\}$ 
5:  $n = \lceil N/Q \rceil$  {Number of simulations per worker}
6:
7: if  $myID = 0$  then
8:    $Result[Q] = 0$  {Storage on worker 0 for each node's result}
9: else
10:   $Result = 0$ 
11: end if
12:
13: {Transmit data}
14: if  $myID = 0$  then
15:  Broadcast parameters to other workers
16: else
17:  Receive parameters from worker 0
18: end if
19: Perform simulation
20: Synchronize workers
21:
22: if  $myID = 0$  then
23:  Receive results from other workers
24: else
25:  Transmit results to worker 0
26: end if
27:  $FinalResult = \frac{1}{(Q \times n)} \sum_{i=0}^{Q-1} Result[i]$ 

```

---

The primary disadvantage of MPI is the communication cost among servers. High speed networks such as Ethernet and Infiniband lack the performance of data transfers along the bus within the server. Ethernet is capable of 100 Gbps whereas Infiniband can achieve 56 Gbps with development for both standards striving for 250 Gbps in the future. However, the theoretical maximum throughput of a 9th generation Intel processor between memory is 635 Gbps [29].

Other hardware architectures can be utilized within a host to provide the benefits of large scale multiprocessing while reducing the communication cost. For example,

graphics processing units are designed to handle a large number of identical calculations in parallel such as those performed in Monte Carlo simulations.

## 2.4 GPU

Graphics processing units (GPUs) are specialized processors optimized to perform calculations necessary for the display of graphics on computing systems. Driven by industries such as video gaming the need for more advanced graphics capabilities grew, and GPUs evolved from highly specialized (and less flexible) devices to a programmable device capable of running software separate from the host. GPUs fall into the category of SIMD, *single instruction, multiple data*, because they execute the same instruction in parallel over different memory addresses.

In 2007, NVIDIA released the first version of its CUDA toolkit to the public enabling developers to make use of the parallelism of graphics processing unit (GPU) in their applications [30]. Developers had to write their code with the GPU in mind, and it required the programmer to transfer the data and program to the GPU, execute it, and then download the results. Current versions of NVIDIA's CUDA toolkit along with third-party tools such as the directive-driven OpenACC attempt to reduce the overhead required to learn this computing paradigm by partially automating the housekeeping tasks such as memory management and data transfer.

GPU computing is now a mainstream technology with 125 of the Top 500 and Green 500 supercomputers employing NVIDIA GPUs [22]. GPUs are desirable for their massive parallelism in a small footprint. Along with the space-saving attributes, GPUs are energy efficient with a lower FLOPS per watt than traditional CPU's [31].

However, NVIDIA is not the only manufacturer that produces GPUs that can be utilized to accelerate applications. AMD produces graphics cards capable of high performance computing and is the most popular GPU manufacturer for cryptocurrency

mining [32] [33]. Unlike NVIDIA which has its proprietary development environment, AMD uses an industry standard called OpenCL.

Apple released the first version of OpenCL in 2009, and it has been adopted by Intel, IBM, NVIDIA, and others. It is a specification meant to insulate the developer from the underlying hardware and provide portability across platforms. Applications written using the OpenCL APIs can be recompiled and executed on a different platform; this allows a program written and tested on a multi-core CPU to be placed in production on a multi-GPU host. While being platform agnostic eliminates or reduces the need to rewrite code when moving to a different systems, the primary disadvantage of this cross-platform approach is the inability of the developer to exploit platform specific optimizations.

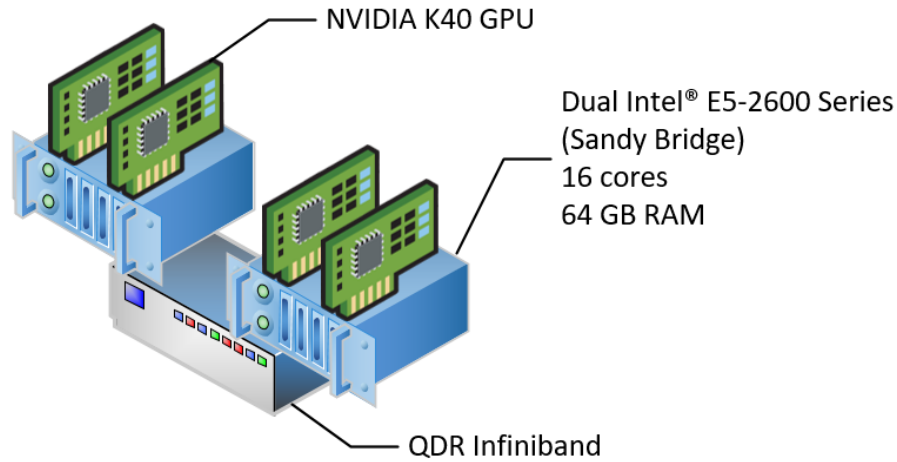


Figure 2.1: Specifications for the seven node GPU-enabled cluster used in the research

The code being developed for this dissertation was written to utilize multiple NVIDIA GPUs on a cluster of Linux hosts with communication between the nodes accomplished using MPI. Each of the seven hosts in the cluster has two NVIDIA Tesla K40's with 2880 CUDA cores and 12 GB of memory each. This architecture is presented in Fig. 2.1, and the pseudocode is outlined in Algorithm 5.

Not all problems work well when ported to a GPU. A GPU's strength is evident in

---

**Algorithm 5** MPI multi-GPU Monte Carlo method

---

```

1: {Define variables}
2:  $myID = MPI\_THREAD\_ID$ 
3:  $N = \{\text{Number of simulations}\}$ 
4:  $Q = \{\text{Number of MPI worker nodes}\}$ 
5:  $n = \lceil N/Q \rceil$  {Number of simulations per worker}
6:
7: if  $myID = 0$  then
8:    $Result[Q] = 0$  {Storage on worker 0 for each node's result}
9: else
10:   $Result = 0$ 
11: end if
12:
13: {Transmit data}
14: if  $myID = 0$  then
15:  Broadcast parameters to other workers
16: else
17:  Receive parameters from worker 0
18: end if
19:
20: Allocate GPU memory for parameters and results
21: Upload parameters to GPU
22: Launch kernel
23: Perform simulation on GPU
24: Download results from GPU
25: Synchronize workers
26:
27: if  $myID = 0$  then
28:  Receive results from other workers
29: else
30:  Transmit results to worker 0
31: end if
32:  $FinalResult = \frac{1}{(Q \times n)} \sum_{i=0}^{Q-1} Result[i]$ 

```

---

applications that require the same calculation to be performed over a set of data (e.g., processing a matrix or performing a Monte Carlo simulation). Scenarios involving excessive branching causes *warp divergence* and results in the GPU performing unnecessary calculations [34].



### 2.4.1 NVIDIA CUDA

NVIDIA is one of the predominant suppliers of discrete GPUs used in consumer and commercial applications with 67.9% of the market [35] during the second quarter of 2019. They were the first manufacturer to allow developers to execute code within the GPU through programmable shaders in 2001 with the GeForce 3 graphics card. In 2007, NVIDIA released the CUDA toolkit which provided a series of compiler tools to ease the development of applications that could leverage the GPUs parallel nature.

The CUDA toolkit consists of the `nvcc` C/C++ compiler and numerical libraries such as `cuBLAS` for linear algebra, `cuFFT` for Fast Fourier Transforms, and `nvGRAPH` for graph analysis. Libraries such as `cuBLAS` have APIs identical to their CPU counterpart making it easier to port existing CPU code to the GPU architecture and to reduce the learning curve for developers.

While C/C++ is the default language provide by NVIDIA, other languages can be used using third-party products. PGI offers a CUDA-enabled FORTRAN compiler developed in conjunction with NVIDIA. The `Numba` compiler for Python can utilize CUDA-enabled GPUs and help overcome the performance of an interpreted language. The Julia programming language is gaining interest from the scientific computing community as it offers the ease of development from Python with the performance of compiled languages such as C, and it has CUDA support through libraries.

CUDA is designed for NVIDIA GPUs, and this gives developers the advantage of accessing proprietary features of the hardware to improve performance. The cost of this optimization is portability in that CUDA APIs are unavailable on other GPU manufacturers; moving to another platform would require rewriting the code.

## 2.4.2 OpenCL

The OpenCL standards were developed by the Khronos Group, a collective of organizations with the desire to create a series of cross-platform APIs:

“Khronos royalty-free open standards for 3D graphics, Virtual and Augmented Reality, Parallel Computing, Neural Networks, and Vision Processing.”

[36]

OpenCL code can be compiled and executed on a variety of platforms from multicore CPUs, GPUs, hardware accelerators, and even mobile devices. Developers can create applications that can be compiled on devices that are significantly different in hardware and capabilities. Abstraction from the underlying hardware and portability come at the price of granular control. Programmers are at the mercy of the manufacturers to deliver optimized OpenCL drivers.

Like CUDA applications, OpenCL programs consist of two parts: (1) the code that is executed on the CPU, and (2) the *kernel* which is executed on the targeted device. The kernel is written in OpenCL C++, which is a subset of the C++ 14 standard [37]. The kernel is uploaded to the device and compiled at runtime rather than being precompiled.

There is a broad range of programming languages that support OpenCL APIs ranging from popular languages such as C++, FORTRAN, and Python to more esoteric languages such as Haskell, LISP, and Delphi. However, all require the kernel to be written in OpenCL C++.

## 2.5 Conclusion

Complex simulations demand computational resources found in high-performance computing environments. These systems can differ in their hardware and software

designs making portable code a challenge. Systems can consist of a single server with multiple CPUs containing many cores, they can be clusters of servers interconnected by high-speed networks, or an HPC system can be built using GPUs with thousands of cores.

Multicore programming leverages multiple processors and cores hosted within the same physical server. The advantage of this paradigm is that the threads can share resources such as memory and disk allowing for simpler designs and faster communication. POSIX threads and OpenMP are two libraries commonly used to accomplish this. Pthreads provides for more granular, low-level control, while OpenMP conceals the basic housekeeping tasks making it easier to develop with.

When working with clusters, MPI is the most frequently used library for applications running across multiple hosts. MPI is a series of APIs that handle process creation, data transfer, and synchronization. MPI libraries are available for Linux and other Unix derivatives, Apple’s MacOS, and Microsoft Windows. Since MPI is standardized, code can be ported with minimal changes.

GPUs offer a low-energy HPC solution. Of the top 500 supercomputers, 25% use GPUs [22]. The two dominant manufacturers are NVIDIA and AMD. GPUs offer dense computational power at a fraction of the power usage for an equivalent CPU configuration, and the acquisition cost is comparably less. This hardware requires a shift in programming paradigm to produce code optimized for the architecture.

CUDA and OpenCL are the two predominant APIs for developing applications to run on GPUs. CUDA offers greater control over the execution on the hardware but is proprietary to NVIDIA’s hardware. OpenCL is an open standard and not only executes on GPUs by AMD, NVIDIA, and Intel but also supports other hardware platforms. The benefits and trade-offs must be considered by the developer.

As highlighted, Monte Carlo simulations are ideal applications for the GPU platform.

It is classified as “embarrassing parallel” due to the fact that each simulation is executed independently, and communication among threads occurs at the beginning when parameters are exchanged and at the end when results are aggregated. Many fields utilize Monte Carlo simulations to estimate results when analytical solutions are not available. Finance is one such field where problems can easily extend into hundreds of dimensions.

## CHAPTER III: MULTILEVEL MONTE CARLO

### 3.1 Introduction

Few physical processes can be modeled by deterministic systems and thus demand other methods of solving. Computers allow the simulation of complex systems that would be otherwise impossible to solve. Large problems such as climate models require computing resources beyond traditional desktops and laptops to simulate. For example, in [38] researchers utilized 8192 nodes to perform a climate simulation with a throughput of 1.22 SYPD (*simulated years per wall clock day*). It would take approximately 82 days to simulate 100 years.

The nature of the problem dictates the implementation of the simulation. Consider the problem of solving a partial differential equation (PDE) in a two-dimensional space over time. The value at a single point on the grid is dependent on the values of its neighbors. Problem decomposition is the process of dividing a problem into smaller subproblems that could potentially be solved in parallel [23]. In the sample problem, as long as the neighboring values are available, the value of each point at the next time step can be solved independently of the others.

Problems can grow in complexity from a higher resolution in spatial discretization, smaller time steps, or from including additional features (e.g., adding an equation to model a new chemical reaction in a chemical Langevin model [39]). By designing the problem in a manner that allows multiple threads to perform chunks of work

independently, the computational hardware can be scaled to allow the simulation to run in an acceptable time. In [23], several techniques for decomposing a parallel problem are discussed:

**Recursive decomposition** Recursively dividing a problem into subproblems that can be solved independently.

**Data decomposition** Dividing datasets into partitions that can be operated on in parallel.

**Exploratory decomposition** For search problems the solution space is divided into separate subspaces and searched concurrently.

**Speculative decomposition** When multiple possible paths could be chosen, one thread performs the calculation that determines the path. Other threads will calculate each of the possible paths. Once the correct path is identified, the value from that calculation is retained, and the others are discarded.

**Hybrid decomposition** Simultaneous application of two or more of the decomposition techniques listed above together.

Certain classes of problems are referred to as “embarrassingly parallel” due to the ease in which they can be parallelized. One such algorithm is the Monte Carlo method which is often used to approximate the solution to stochastic differential equations.

## 3.2 Monte Carlo

Only the simplest of SDEs have closed form solutions; more complex variations are solved through simulation using Monte Carlo methods. Monte Carlo simulations can be computationally expensive due to the number of samples required and the number of steps from the discretization in either space or time [3].

The Monte Carlo estimator [3] can be written as

$$\mathbb{E}[X(T, \omega)] = \frac{1}{M} \sum_{i=1}^M X(T, \omega) \quad (3.1)$$

where  $M$  is the number of Monte Carlo simulations performed,  $T$  is the final time, and  $\omega$  represents the random values taken from  $N(0, \sqrt{dt})$ .  $X(T, \omega)$  is a stochastic differential equation that takes the form of Eqn. 1.2. To better illustrate the Monte Carlo process, Fig. 3.1 shows five random paths of the equation

$$dS = rSdt + \sigma SdW \quad (3.2)$$

using 50 time steps with a step size of 0.02. The average at each step is shown using a bold, black line. The analytical solution for this process is graphed in the bold, red line and is calculated by

$$S_T = S_0 e^{rT} \quad (3.3)$$

The simulation error results from two sources: the error generated due to the discretization of the problem and the error from the Monte Carlo process. These types of error are classified in terms of *weak convergence* and *strong convergence* [3], [40]. Weak error describes the *bias* that comes from the chosen method of discretization while strong error is controlled by the variance resulting from the Monte Carlo simulation.

Let  $E_M[X(T, \omega)]$  be the result from the Monte Carlo simulation at time  $T$  after  $M$  samples,  $\hat{X}(T)$  be the true value at time  $T$ ,  $X(T\omega)$  be a single sample, then the error can be expressed as [3]:

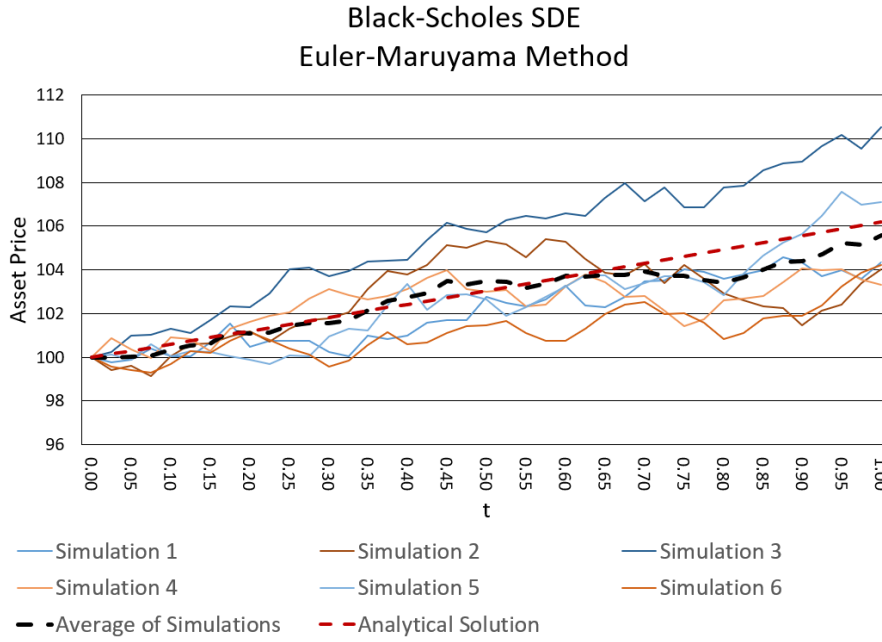


Figure 3.1: Example of five random samples from the equation  $dS = rSdt + \sigma SdW$  for  $0 \leq t \leq 1$  with 50 time steps and  $dt = 0.02$ . The bold black line is the average of the five samples, and the bold red line is the analytical solution.

$$\underbrace{\left| \overbrace{\hat{X}(T)}^{\text{Exact value}} - \overbrace{\mathbb{E}_M[X(T, \omega)]}^{\text{Expected value of simulation}} \right|}_{\text{Error in simulation}} \leq \underbrace{\left| \overbrace{\hat{X}(T)}^{\text{Exact value}} - \overbrace{X(T, \omega)}^{\text{Sample}} \right|}_{\substack{\text{Discretization error} \\ \text{Weak error}}} + \underbrace{\left| \overbrace{X(T, \omega)}^{\text{Sample}} - \overbrace{\mathbb{E}_M[X(T, \omega)]}^{\text{Expected value of simulation}} \right|}_{\substack{\text{Monte Carlo error} \\ \text{Strong error}}} \quad (3.4)$$

The relationship of weak error and Monte Carlo error to the step size of the discretization can be seen in Fig. 3.2. The error is estimated by using a Monte Carlo simulation with Eqn. 3.2 and comparing it to the analytical solution from Eqn. 3.3 through varied values of  $dt$ , the temporal step size.

The variance of the Monte Carlo simulation and the desired level of confidence dictates the number of samples required for the simulation. To calculate the necessary samples, a simulation is performed with a small number of iterations to estimate the



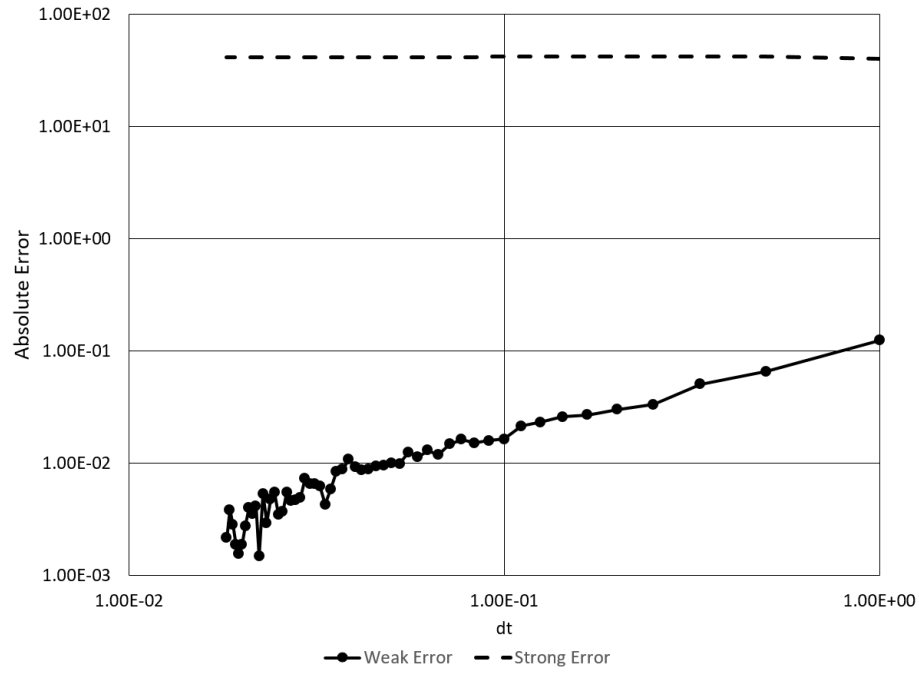


Figure 3.2: Weak error caused by the discretization of the sample problem in relation to the step size  $dt$  using  $10^8$  simulations.

variance. The sample size is determined by leveraging the equation for confidence intervals

$$\left( \bar{x} - z \frac{s}{\sqrt{n}}, \bar{x} + z \frac{s}{\sqrt{n}} \right) \quad (3.5)$$

where  $\bar{x}$  is the sample mean,  $s$  is the sample standard deviation,  $n$  is the number of samples,  $z$  is the z-score for the desired confidence interval, and  $\mu$  is the true mean. The number of samples is obtained by solving for  $n$  in Eqn. 3.5:

$$\begin{aligned}
\bar{x} - z \frac{s}{\sqrt{n}} &< \mu < \bar{x} + z \frac{s}{\sqrt{n}} \\
-z \frac{s}{\sqrt{n}} &< \mu - \bar{x} < z \frac{s}{\sqrt{n}} \\
-z &< \frac{(\mu - \bar{x})\sqrt{n}}{s} < z \\
-\frac{(\mu - \bar{x})\sqrt{n}}{s} &< z < \frac{(\mu - \bar{x})\sqrt{n}}{s}
\end{aligned}$$

The term  $(\mu - \bar{x})$  is the error between the sample mean and the true mean which will be replaced by  $\epsilon$ . The equation can be rewritten to solve for  $n$ .

$$\begin{aligned}
z &= \frac{\epsilon\sqrt{n}}{s} \\
\left(\frac{s \times z}{\epsilon}\right)^2 &= n
\end{aligned}$$

As the desired level of confidence ( $z$ ) increases and the error tolerance ( $\epsilon$ ) diminishes the number of samples increases exponentially. If the system being simulated exhibits a high variance ( $s^2$ ), then it will also demand a greater number of simulations [41]. Numerical methods can be utilized to reduce the variance of the system and thus improve the runtime of the simulation.

### 3.3 Multilevel Monte Carlo

Mike Giles [42] first proposed the multilevel Monte Carlo (MLMC) method as a means to reduce the variability found in Monte Carlo methods. This involves sampling differing *levels* where each level has a different time step that aligns with the level

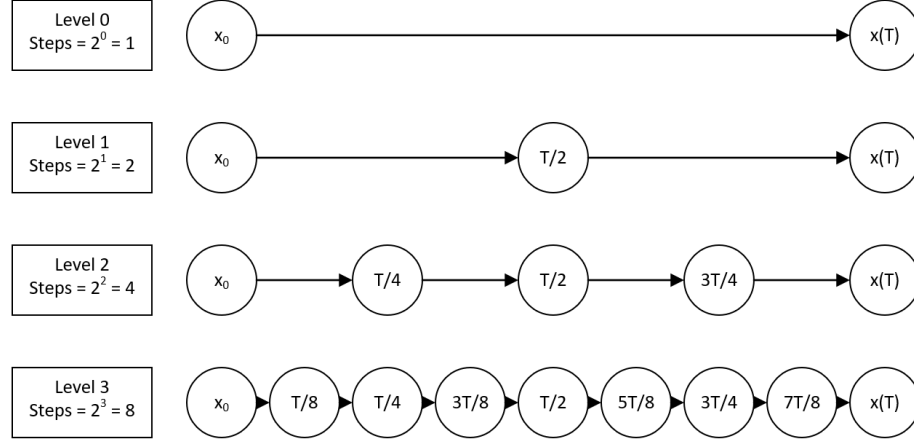


Figure 3.3: Multilevel Monte Carlo

above it (see Fig. 3.3).

The result from the initial level 0 ( $l_0$ ) is used as the starting value. Next the Monte Carlo process is performed on level 0 ( $l_0$ ) and level 1 ( $l_1$ ) such that the same random values from  $N(0, \sqrt{dt_{l_1}})$  are used in both. In this step  $l_0$  is referred to as the *coarse* step and  $l_1$  is the *fine* step. The difference between the results of the two steps is the result of the Monte Carlo for that level.

$$\begin{aligned}
 X_{n+1}^C &= X_n^C + \alpha(X_n^C)dt^C + \beta(X_n^C)(dW_1^f + dW_2^f) \\
 &\quad + \frac{1}{2}\beta(X_n^C)\beta'(X_n^C)(dW_1^f dW_2^f - dt^C) \\
 X_{n+\frac{1}{2}}^f &= X_n^f + \alpha(X_n^f)dt^f + \beta(X_n^f)dW_1^f \\
 &\quad + \frac{1}{2}\beta(X_n^f)\beta'(X_n^f)(dW_1^f - dt^f) \\
 X_{n+1}^f &= X_{n+\frac{1}{2}}^f + \alpha(X_{n+\frac{1}{2}}^f)dt^f + \beta(X_{n+\frac{1}{2}}^f)dW_2^f \\
 &\quad + \frac{1}{2}\beta(X_{n+\frac{1}{2}}^f)\beta'(X_{n+\frac{1}{2}}^f)(dW_2^f - dt^f)
 \end{aligned} \tag{3.6}$$

Let  $\text{MLMC}_{l_N}$  be the MLMC estimation for  $l_N$  (i.e., the difference between the

results of  $X^C$  and  $X^f$ ,  $X^f(T, \omega)$  be the result of the Monte Carlo performed with the fine time step of  $dt = T/2^N$ , and  $X^C(T, \omega)$  be the Monte Carlo result from the coarse level with  $dt = T/2^{N-1}$  but using the same random values taken from  $l_N$ .

$$E[\text{MLMC}_{l_N}(T, \omega)] = \frac{1}{M} \sum_{i=1}^M \left( X^f(T, \omega) - X^C(T, \omega) \right) \quad (3.7)$$

Once the initial value of  $l_0$  is calculated and the corrections from the remaining levels  $l_1 \dots l_{N-1}$ , the final estimation for the MLMC simulation is given by

$$E[\text{MLMC}(T, \omega)] = \sum_{i=0}^{N-1} E[\text{MLMC}_{l_i}(T, \omega)] \quad (3.8)$$

One advantage of multilevel Monte Carlo is the ease at which it can be parallelized. Like Monte Carlo, the samples taken during MLMC are independent of one another, and communication only occurs at the beginning when parameters are distributed and at the end when the results are aggregated. Algorithms 6, 7, and 8 are pseudocode implementations of the MLMC process. Thought must be given to the implementation to avoid an imbalance of work when parallelizing. Lower levels of the MLMC will have fewer time steps but require more samples while the higher levels will have a lower variance and require fewer samples but require more time steps. From Giles [42] the number of samples required for each level is controlled by the variance at each level as well as the total simulation. Let  $V_{l_N}$  is the variance for that level,  $h_{l_N}$  is the step size, and  $\epsilon$  controls the desired accuracy, then the number of samples  $M_{l_N}$  needed at level  $l_N$  is

$$M_{l_N} = \lceil 2\epsilon^{-2} \sqrt{V_{l_N} h_{l_N}} \left( \sum_{i=0}^N \sqrt{V_i / h_i} \right) \rceil \quad (3.9)$$

It is useful to have a definition for the *cost* of each MLMC level. Let  $C_{l_N}$  be the cost of performing level  $N$  of the MLMC where  $N$  is not the initial level ( $l_0$ ), let  $M_{l_N}$

---

**Algorithm 6** Multi-level Monte Carlo

---

```

1: {Define variables}
2:  $L_0$  = Starting level
3:  $N$  = Total number of levels
4:  $s$  = Number of initial samples
5:  $T$  = Ending time
6:
7: {Initial simulation to estimate variance}
8: result[0] = LEVEL0( $L_0, T, s$ ) {See alg 7}
9: for  $m = 1$  ;  $m < N$  ;  $m++$  do
10:   result[ $m$ ] = LEVELN( $L_m, T, s$ ) {See alg 8}
11: end for
12:
13: Estimate samples needed by level using eqn 3.8 and store in samples[ $L_m$ ] for each
    level  $m$ 
14:
15: {Actual simulation}
16: result[0] = LEVEL0( $L_0, T, \text{samples}[0]$ ) {See alg 7}
17: for  $m = 1$  ;  $m < N$  ;  $m++$  do
18:   result[ $m$ ] = LEVELN( $L_m, T, \text{samples}[L_m]$ ) {See alg 8}
19: end for
20:
21: {Calculate final result}
22: for  $m = 0$  ;  $m < N$  ;  $m++$  do
23:   final+ = result[ $m$ ]
24: end for

```

---



---

**Algorithm 7** Multi-level Monte Carlo Level 0 Function

---

```

1: {Initial level simulation}
2: FUNCTION LEVEL0( $L_0, T, s$ ) {
3:   steps =  $2^{L_0}$ 
4:    $dt = T/\text{steps}$ 
5:   result[0] = 0, var[0] = 0
6:
7:   for  $i = 0$  ;  $i < s$  ;  $s++$  do
8:     for  $j = 0$  ;  $j < \text{steps}$  ; steps++ do
9:       Perform standard Monte Carlo simulation with  $dt$ 
10:    end for
11:   end for
12:
13:   Store  $L_0$  results in result[0] and variance in var[0]
14: }
```

---

---

**Algorithm 8** Multi-level Monte Carlo Level N Function
 

---

```

1: FUNCTION LEVELN( $L_N, T, s$ ) {
2:   {Remaining levels}
3:   stepsfine =  $2^{L_N}$ 
4:   stepscoarse =  $2^{(L_N-1)}$ 
5:    $dt_{\text{fine}} = T/\text{steps}_{\text{fine}}$ 
6:    $dt_{\text{coarse}} = T/\text{steps}_{\text{coarse}}$ 
7:   result[ $L_N$ ] = 0, var[ $L_N$ ] = 0
8:
9:   for  $i = 0 ; i < s ; s++$  do
10:    for  $j = 0 ; j < \text{steps}_{\text{coarse}} ; \text{steps}_{\text{coarse}}++$  do
11:      Select two random values  $dW_1 = \sqrt{dt} \times N(0, 1)$  and  $dW_2 = \sqrt{dt} \times N(0, 1)$ 
12:      Perform coarse step with  $dt_{\text{coarse}}$  and  $(dW_1 + dW_2)$ 
13:      Perform two fine steps with  $dt_{\text{fine}}$  and  $dW_1$  and  $dW_2$ 
14:    end for
15:    result[ $L_N$ ] += (resultfine - resultcoarse)
16:  end for
17:
18: Store  $L_N$  results in result[ $L_N$ ] and variance in var[ $L_N$ ]

```

---

be the number of Monte Carlo samples needed at  $l_N$ , and assume that the number of steps at any level is a power of two. The computational cost can then be defined as

$$C_{l_N} = M_{l_N} \times (2^N + 2^{N-1}) \quad (3.10)$$

Work can be divided in such a way that the individual levels are performed by different threads or each thread performs an equal portion of the work for every level. The latter method minimizes the risk of idle resources but makes the estimation of variance in the simulation more difficult as samples are taken on different hosts.

### 3.4 Antithetic Multilevel Monte Carlo

Antithetic methods are tools to reduce the variance in Monte Carlo simulations and therefore the number of samples needed. This can result in a significant improvement in simulation time when coupled with multilevel Monte Carlo methods. In addition

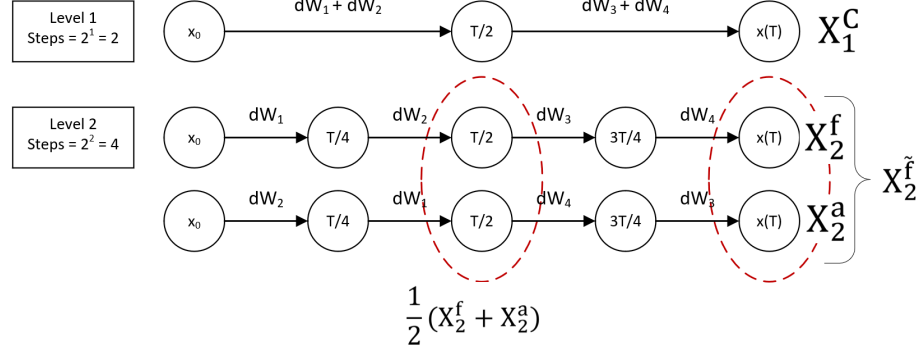


Figure 3.4: Antithetic multilevel Monte Carlo process

to the variance reduction properties, an antithetic approach permits the use of the Milstein method and MLMC without the need to simulate the Lévy areas for simulations containing mixed derivative terms [43], [44].

The antithetic approach to MLMC reduces variance by leveraging antithetic variates which are used in traditional Monte Carlo simulations [43]. The approach involves using alternating Brownian motion values ( $dW$ ) to produce two fine grain paths and using the average as the estimation. Eqn 3.11 is modified such that:

$$\begin{aligned}
 X_{n+1}^C &= X_n^C + \alpha(X_n^C)dt^C + \beta(X_n^C)(dW_1^f + dW_2^f) \\
 &\quad + \frac{1}{2}\beta(X_n^C)\beta'(X_n^C)(dW_1^f dW_2^f - dt^C) \\
 X_{n+\frac{1}{2}}^f &= X_n^f + \alpha(X_n^f)dt^f + \beta(X_n^f)dW_1^f \\
 &\quad + \frac{1}{2}\beta(X_n^f)\beta'(X_n^f)(dW_1^f - dt^f) \\
 X_{n+1}^f &= X_{n+\frac{1}{2}}^f + \alpha(X_{n+\frac{1}{2}}^f)dt^f + \beta(X_{n+\frac{1}{2}}^f)dW_2^f \\
 &\quad + \frac{1}{2}\beta(X_{n+\frac{1}{2}}^f)\beta'(X_{n+\frac{1}{2}}^f)(dW_2^f - dt^f) \\
 X_{n+\frac{1}{2}}^a &= X_n^a + \alpha(X_n^a)dt^f + \beta(X_n^a)dW_2^f \\
 &\quad + \frac{1}{2}\beta(X_n^a)\beta'(X_n^a)(dW_2^f - dt^f)
 \end{aligned} \tag{3.11}$$

$$\begin{aligned}
X_{n+1}^a &= X_{n+\frac{1}{2}}^a + \alpha(X_{n+\frac{1}{2}}^a)dt^f + \beta(X_{n+\frac{1}{2}}^a)dW_1^f \\
&\quad + \frac{1}{2}\beta(X_{n+\frac{1}{2}}^a)\beta'(X_{n+\frac{1}{2}}^a)(dW_1^f - dt^f) \\
\hat{X}_{n+1}^f &= \frac{1}{2}(X_{n+1}^f + X_{n+1}^a)
\end{aligned}$$

---

**Algorithm 9** Antithetic multi-level Monte Carlo

---

```

1: {Define variables}
2:  $L_0$  = Starting level
3:  $N$  = Total number of levels
4:  $s$  = Number of initial samples
5:  $T$  = Ending time
6:
7: {Initial simulation to estimate variance}
8: result[0] = LEVEL0( $L_0, T, s$ ) {See alg 7}
9: for  $m = 1$  ;  $m < N$  ;  $m++$  do
10:   result[ $m$ ] = LEVELN( $L_m, T, s$ ) {See alg 8}
11: end for
12:
13: Estimate samples needed by level using eqn 3.8 and store in samples[ $L_m$ ] for each
    level  $m$ 
14:
15: {Actual simulation}
16: result[0] = LEVEL0( $L_0, T, \text{samples}[0]$ ) {See alg 7}
17: for  $m = 1$  ;  $m < N$  ;  $m++$  do
18:   result[ $m$ ] = LEVELN( $L_m, T, \text{samples}[L_m]$ ) {See alg 8}
19: end for
20:
21: {Calculate final result}
22: for  $m = 0$  ;  $m < N$  ;  $m++$  do
23:   final+ = result[ $m$ ]
24: end for

```

---



---

**Algorithm 10** Antithetic multi-level Monte Carlo Level 0 Function

---

```

1: {Initial level simulation}
2: FUNCTION LEVEL0( $L_0, T, s$ ) {
3:   steps =  $2^{L_0}$ 
4:    $dt = T/\text{steps}$ 
5:   result[0] = 0, var[0] = 0
6:
7:   for  $i = 0 ; i < s ; s++$  do
8:     for  $j = 0 ; j < \text{steps} ; \text{steps}++$  do
9:       Perform standard Monte Carlo simulation with  $dt$ 
10:    end for
11:  end for
12:
13:  Store  $L_0$  results in result[0] and variance in var[0]
14: }
```

---



---

**Algorithm 11** Antithetic multi-level Monte Carlo Level N Function

---

```

1: FUNCTION LEVELN( $L_N, T, s$ ) {
2:   {Remaining levels}
3:   stepsfine =  $2^{L_N}$ 
4:   stepscoarse =  $2^{L_N}$ 
5:    $dt_{\text{fine}} = T/\text{steps}_{\text{fine}}$ 
6:    $dt_{\text{coarse}} = T/\text{steps}_{\text{coarse}}$ 
7:   result[ $L_N$ ] = 0, var[ $L_N$ ] = 0
8:
9:   for  $i = 0 ; i < s ; s++$  do
10:    for  $j = 0 ; j < \text{steps}_{\text{coarse}} ; \text{steps}_{\text{coarse}}++$  do
11:      Select two random values  $dW_1 = \sqrt{dt} \times N(0, 1)$  and  $dW_2 = \sqrt{dt} \times N(0, 1)$ 
12:      Perform coarse step with  $dt_{\text{coarse}}$  and  $(dW_1 + dW_2)$ 
13:       $A =$  Perform two fine steps with  $dt_{\text{fine}}$  and  $dW_1$  and  $dW_2$ 
14:       $B =$  Perform two fine steps with  $dt_{\text{fine}}$  and  $dW_2$  and  $dW_1$ 
15:      resultfinea +=  $(A + B)/2$ 
16:    end for
17:    result[ $L_N$ ] +=  $(\text{result}_{\text{fine}}^a - \text{result}_{\text{coarse}})$ 
18:  end for
19:
20:  Store  $L_N$  results in result[ $L_N$ ] and variance in var[ $L_N$ ]
```

---

## 3.5 Alternative Monte Carlo Methods

Several alternatives to traditional Monte Carlo methods have been developed to improve the rate of convergence. MC generates random values using a pseudorandom number algorithm. Quasi Monte Carlo methods can improve the rate of convergence in certain applications through a deterministic algorithm or quasirandom number generation [45]. This approach can also be applied to MLMC with an improvement in computational cost as shown in [46].

The Monte Carlo method uses independent pseudorandom numbers during the simulation taken from a distribution with known parameters. However, in some models, the parameters are unknown or the distribution cannot be described [47]. Markov Chain Monte Carlo (MCMC) provides a tool for constructing sequences of random values from complex distributions when traditional random sampling will fail - including high dimensionality. In [48], MCMC is combined with MLMC to greatly improve the performance of stochastic chemical reaction networks using tau-leaping by performing the simulation using differing time steps. MCMC requires more samples than Monte Carlo resulting in a longer runtime for simulations that can utilize MC [49].

## 3.6 Conclusion

Complex stochastic systems must be modeled using computer simulations, and the Monte Carlo method is one such popular approach. This consists of performing the simulation repeatedly until the result converges. Depending on the scale of the problem, this convergence can occur in seconds or in months. Much research has been conducted in an attempt to improve performance through modification of numerical and computational methods (see for example [42], [44], [4], [50], [21]).

A stochastic differential equation can be approximated using Monte Carlo simulations by taking small, discrete time steps  $dt$  until the terminal time  $T$ . The solution is calculated using the Monte Carlo estimator

$$\mathbb{E}[X(T, \omega)] = \frac{1}{M} \sum_{i=1}^M X(T, \omega)$$

The simulation is prone to two sources of error: *weak error* caused by the discretization method and *strong error* from the variance from the Monte Carlo simulation. These contribute to the number of samples necessary in order for the result to converge.

One numerical method to reduce variance and thus the number of samples is multilevel Monte Carlo. This method performs a Monte Carlo simulation by discretizing the problem using varying time step sizes. The initial level  $L_0$  is estimated using a large step size with MC. Subsequent levels are estimated by the difference between a *coarse* level and a *fine* level. The result is a faster rate of convergence than MC.

Further performance improvements can be had through antithetic MLMC. During the calculation of the *fine* level of MLMC, two paths are estimated by transposing the order of concurrent pairs of Brownian terms  $dW$  and averaging the result. This leverages properties of antithetic variates from MC to reduce the variance in the simulation and thus the execution time.

Numerical methods can improve the performance of SDE simulations through variance reduction in MC. However, this alone is not always sufficient to produce results in a desired time. High performance computing systems must be employed to gain even further improvements in computational time.

## CHAPTER IV: APPLICATIONS IN FINANCE

### 4.1 Option Pricing

One of the earliest applications of Brownian motion was in the field of finance. The seminal paper by Fischer Black, Myron Scholes, and Robert Merton provided an analytical solution to the problem of pricing financial options. While the original Black-Scholes formula was in the form of a partial differential equation, it is possible to model the same process with an SDE.

This is a challenging problem as the complexity of the problem grows as additional assets are included. For example, a single asset is a one dimensional SDE. In the real-world, portfolios can consist of hundreds of assets quickly increasing the difficulty. The simplest model involves an asset with a stochastic price ( $S_t$ ) while the other parameters, volatility ( $\sigma$ ) and interest rate ( $r$ ), are constants.

$$dS_t = rS_t dt + \sigma S_t dW_t \tag{4.1}$$

#### 4.1.1 Stochastic Volatility Model

The Heston model [51] expands on the Black-Scholes by changing volatility ( $\sigma$ ) from a constant to a stochastic process. This new two equation model has the form of:

$$dS_t = rS_t dt + \sqrt{v_t}S_t dW_t^S \quad (4.2)$$

$$dv_t = \kappa(\theta - v_t)dt + \sigma\sqrt{v_t}dW_t^v \quad (4.3)$$

For financial portfolios consisting of multiple assets, each asset will have a system of two equations as the price and volatility are unique for each asset. For 100 assets, the system would consist of 200 equations.

#### 4.1.2 Stochastic Volatility, Stochastic Interest Rate Model

One strength of the stochastic simulation approach is the ability to expand the size of the problem. In a paper by Medvedev and Scaillet [52] the authors expand the Heston model to model the interest rate as a stochastic process rather than a fixed value. In addition to the third stochastic variable, Medvedev, et al. included correlation between the three Weiner processes ( $dW_t^S$ ,  $dW_t^r$ , and  $dW_t^v$ ) as real-world processes often exhibit a relationship with other processes. The stochastic volatility, stochastic interest rate model is written as:

$$\begin{aligned} dS_t &= r_t S_t dt + \sqrt{v_t} S_t dW_t^S \\ dv_t &= \kappa_v(\theta - v_t)dt + \sigma_v \sqrt{v_t} dW_t^v \\ dr_t &= \kappa_r(\bar{r} - r_t)dt + \sigma_r \sqrt{r_t} dW_t^r \\ dW^{(i)} dW^{(j)} &= \rho_{i,j} dt \end{aligned} \quad (4.4)$$

where  $i, j$  represent the three equations  $S, v, r$ .

The model is discretized using the Milstein scheme [4] from eqn. 1.4 which takes

the form of:

$$dS_t = \mu_t dt + \sigma_t dW_t + \frac{1}{2} \sigma'_t \sigma_t [(dW)^2 - dt] \quad (4.5)$$

Applying 4.5 to the asset price in 4.4 yields:

$$\begin{aligned} \text{Let } \mu_t &= r_t S_t, \sigma_t = \sqrt{v_t} S_t, \sigma'_t = \sqrt{v_t}, \text{ then} \\ dS_t &= r_t S_t dt + \sqrt{v_t} S_t dW_t^S + \frac{1}{2} v_t S_t [(dW_t^S)^2 - dt] \end{aligned}$$

Applying 4.5 to the volatility in 4.4 yields:

$$\begin{aligned} \text{Let } \mu_t &= \kappa_v (\theta - v_t), \sigma_t = \sigma_v \sqrt{v_t}, \sigma'_t = \frac{\sigma_v}{2\sqrt{v_t}}, \text{ then} \\ dv_t &= \kappa_v (\theta - v_t) dt + \sigma_v \sqrt{v_t} dW_t^v + \frac{\sigma_v^2}{2} [(dW_t^v)^2 - dt] \end{aligned}$$

Applying 4.5 to the interest rate in 4.4 yields:

$$\begin{aligned} \text{Let } \mu_t &= \kappa_r (\bar{r} - r_t), \sigma_t = \sigma_r \sqrt{r_t}, \sigma'_t = \frac{\sigma_r}{2\sqrt{r_t}}, \text{ then} \\ dr_t &= \kappa_r (\bar{r} - r_t) dt + \sigma_r \sqrt{r_t} dW_t^r + \frac{\sigma_r^2}{2} [(dW_t^r)^2 - dt] \end{aligned}$$

The Milstein discretizations provide a new system of SDEs:

$$\begin{aligned}
dS_t &= r_t S_t dt + \sqrt{v_t} S_t dW_t^S + \frac{1}{2} v_t S_t [(dW_t^S)^2 - dt] \\
dv_t &= \kappa_v (\theta - v_t) dt + \sigma_v \sqrt{v_t} dW_t^v + \frac{\sigma_v^2}{2} [(dW_t^v)^2 - dt] \\
dr_t &= \kappa_r (\bar{r} - r_t) dt + \sigma_r \sqrt{r_t} dW_t^r + \frac{\sigma_r^2}{2} [(dW_t^r)^2 - dt]
\end{aligned}$$

This model was the subject of research for this dissertation and a recently submitted paper to the journal *Monte Carlo Methods and Applications* [4] that addressed the implementation on a GPU-enabled cluster using antithetic multilevel Monte Carlo.

## 4.2 Implementation

The keystone upon which this research is based is the implementation of the SDE simulation software on a cluster equipped with GPU resources. The research was performed on a cloud-based HPC service provided by Penguin Computing called “Penguin Computing On Demand” or POD. The H30G cluster consists of seven nodes, each with two NVIDIA K40 GPUs. Among the nodes, there are a total of 40320 CUDA cores and 168 GB of GPU RAM.

Upon launch the software developed during this research determines the number of MPI nodes in the cluster. Then it requests each node to provide the number of GPUs it has. Node 0 launches the variance estimation routine on its GPU 0 with one thread per MLMC level. These threads perform a fixed number of simulations (1000 in the current code) and calculate the variance using the Welford online variance algorithm [53] [54]. The CPU uses the variances to estimate the number of samples needed per MLMC level. Next Node 0 broadcasts the simulation parameters to each

node. The nodes request its GPUs to perform an equal number of simulations per level and to report back the sum for each level as well as the number of simulations performed on each. The actual number will be higher than the calculated number due to rounding. Finally the nodes report back to Node 0 their results which are aggregated.

---

**Algorithm 12** Implemented multi-GPU multilevel Monte Carlo algorithm using MPI. Initialization phase.

---

- 1: Initialize MPI
  - 2:
  - 3: {Define variables}
  - 4: Let  $T$  be the number of threads per GPU
  - 5: Let  $G$  be the number of GPUs per MPI node
  - 6: Let  $M$  be the number of MPI nodes
  - 7:
  - 8: Let  $\epsilon$  be the error parameter
  - 9: Let  $L$  be the number of levels in the MLMC
  - 10: Let  $L_0$  be the initial level of the MLMC
  - 11: Let  $S_0$  be the number initial samples to determine the variance for each level
  - 12: Let  $R_l^n$  be the collection of samples at level  $l$  for thread  $n$
- 

## 4.3 Validation

The initial code created in C++ was a simple Monte Carlo application based on [52] to calculate the price of a European option using stochastic volatility and stochastic interest rate. It performed the simulation using each of the provided thirty-six parameter sets and output the result as well as the variance from the author's result. The number of time steps used was 500, and a static  $10^6$  Monte Carlo samples were taken. This was performed single-threaded on an Intel 2.6 GHz Sandy Bridge CPU [55] on Penguin Computing's cloud-based HPC service and used as an initial baseline.

The second stage involved modifying the CPU code to use multilevel Monte Carlo.



---

**Algorithm 13** Implemented multi-GPU multilevel Monte Carlo algorithm using MPI. Variance estimation phase.

---

- 1: {Estimate the variance at each level to determine the number of samples needed}
  - 2: {The variance calculation occurs on MPI node 0}
  - 3: **for** each level  $L_0 \leq l \leq (L_0 + L - 1)$  **do**
  - 4:   Assign  $L$  threads  $S_0$  samples
  - 5:
  - 6:   **if**  $l = L_0$  **then**
  - 7:     Take  $2^l$  time steps to calculate the price
  - 8:     Add the price to  $R_l$
  - 9:   **end if**
  - 10:
  - 11:   **if**  $l > L_0$  **then**
  - 12:     Take  $2^l$  time steps to calculate the price  $P_l^f$  (fine step)
  - 13:     Using the same normal random variables from the previous step, take  $2^l$  time steps to calculate the price  $P_l^a$ . However, each pair of random variables is swapped for the antithetic estimator.
  - 14:     Take  $2^{l-1}$  time steps to calculate the price  $P_{l-1}^c$  (coarse step)
  - 15:     Add the price correction  $(P_l^f + P_l^a)/2 - P_{l-1}^c$  to  $R_l$
  - 16:   **end if**
  - 17:
  - 18:   Each thread estimates its variance  $V_l$  at each level  $l$  using  $R_l$
  - 19: **end for**
  - 20:
  - 21: The number of samples  $N_l$  are calculated at level  $l$  with step size  $\delta t_l$  using the formula
 
$$N_l = \lceil 2\epsilon^{-2} \sqrt{V_l \delta t_l} \sum_{l=L_0}^{L_0+L-1} \sqrt{V_l / \delta t_l} \rceil$$
- 

---

**Algorithm 14** Implemented multi-GPU multilevel Monte Carlo algorithm using MPI. Simulation phase.

---

- 1: {Perform the MLMC simulation}
  - 2: Distribute parameters to nodes
  - 3: Each MPI node is asked to perform  $\lceil N_l/M \rceil$  simulations
  - 4: Each GPU performs  $\lceil N_l^n/(MG) \rceil$  simulations
  - 5: Each GPU thread performs  $\lceil N_l^n/(MGT) \rceil$  simulations
  - 6:
  - 7: Each MPI node reports back the number of actual simulations  $N_M$  performed due to rounding
  - 8: The final price is calculated as  $R = (\sum_{n=1}^M R^n) / (\sum_{n=1}^M N^n)$
-

The MLMC code used the same Euler-Murayama discretization but calculated the number of Monte Carlo samples for each level based on the variance. Again this code was executed single threaded on the same CPU to serve as a comparison. The purpose was not to speed up the code but to validate that the results matched the baseline. In some instances the MLMC code was slower due to the fact that the number of samples required were greater than the fixed number ( $10^6$ ) used initially.

A key algorithm in the MLMC code was the online variance calculation. It is necessary to estimate the variance at each level of the simulation to determine the number of samples needed, however, it would be inefficient or impossible to store each sample in order to use traditional variance calculation methods learned in introductory statistics. The programs use an online variance algorithm first published by Welford in 1962 [54] [53] that eliminates the need to store samples. The algorithm as implemented was tested by generating multiple sets of random numbers from  $N(0, 1)$  and storing them in a text file. The streams were first passed through the algorithm to estimate the variance. The same streams were loaded into *Minitab* to calculate the true variance, and the two results were compared.

With working Monte Carlo and multilevel Monte Carlo code, the next step was to implement GPU versions of both. Both code bases were modified to execute as a single threaded application on an NVIDIA Quadro M2200, and the results were compared to the CPU versions. They were then modified to execute as multi-threaded applications and the output was then validated.

The four applications (CPU Monte Carlo, CPU multilevel Monte Carlo, GPU Monte Carlo, and GPU multilevel Monte Carlo) were modified to perform antithetic Monte Carlo [43]. Once these results were confirmed, the code underwent one further modification to use the Milstein discretization. At this point the code successfully performs the stochastic volatility, stochastic interest rate pricing for European options

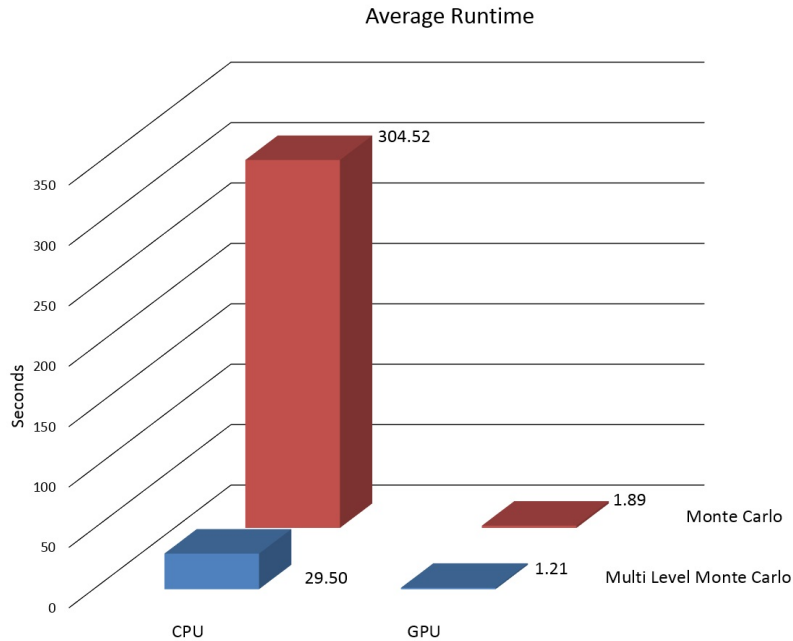


Figure 4.1: Comparison of runtimes for the stochastic volatility, stochastic interest rate model for different platforms and Monte Carlo methods using a single-threaded CPU and a single NVIDIA K20 GPU [4]

	Monte Carlo	Multilevel Monte Carlo
CPU	304.5 sec (100%)	29.5 sec (9.7%)
GPU	1.89 sec (0.6%)	1.21 sec (0.4%)

Figure 4.2: Performance improvement of MLMC and GPU over CPU-based MC [4]

on both a CPU and a GPU platform using either Monte Carlo or multilevel Monte Carlo.

The final modification was extending the code with MPI to execute on a Linux cluster of seven nodes, each equipped with dual NVIDIA K40 GPUs. As before, the results were compared to those of the original single-threaded code to ensure that no bugs were introduced.

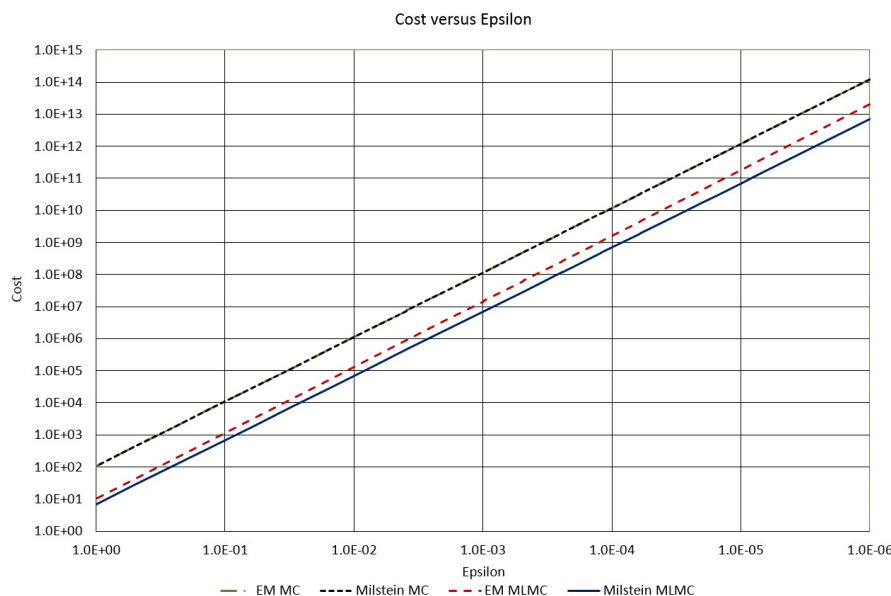


Figure 4.3: Impact of error control variable  $\epsilon$  on the computational cost of the stochastic variable, stochastic interest rate simulation [4]

## 4.4 Results

Fig. 4.1 shows the execution times of the stochastic volatility, stochastic interest rate for CPU and GPU using both a Monte Carlo and a multilevel Monte Carlo simulation. MLMC reduces the variation in the system significantly and yields a substantial improvement over the Monte Carlo. The GPU then provides another level of improvement from the high parallelism associated with the NVIDIA K40. The MLMC implementation on the CPU resulted in a 90.3% improvement over the MC. However, the MLMC GPU implementation is 64.0% of the MC. There is most likely overhead associated with the GPU that places a lower boundary on the speed.

The cost of simulating Eqn. (3.10) is quantified by experimentation and displayed in Fig. 4.3 for both MLMC and MC using the Euler-Murayama and Milstein discretizations. As  $\epsilon$  shrinks, the cost of the simulation increases. The Euler-Murayama and Milstein costs are so close for Monte Carlo that they are indistinguishable on the chart.

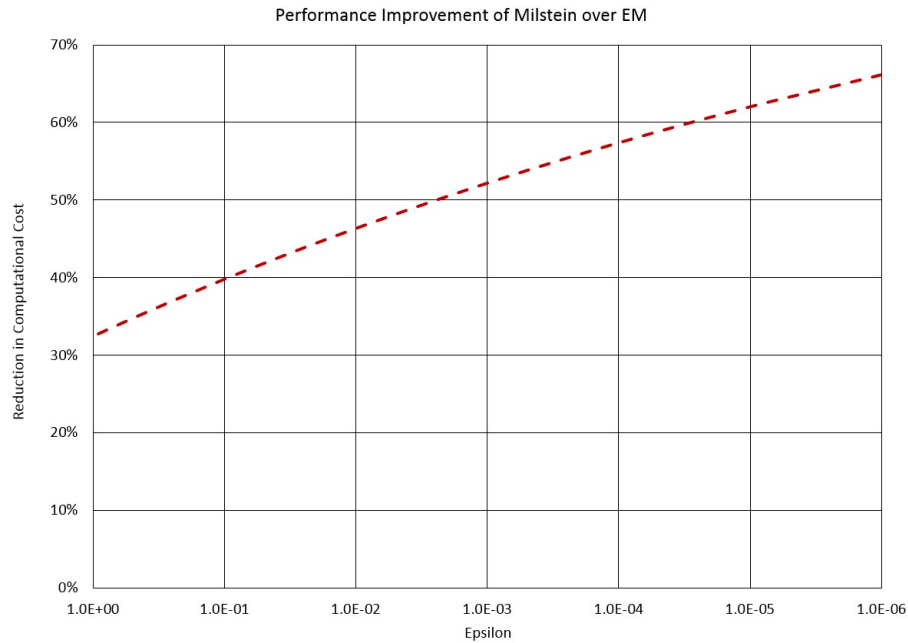


Figure 4.4: Performance improvement of Milstein discretization over Euler-Murayama for multilevel Monte Carlo [4]

The reduction in computational cost made by the antithetic multilevel Monte Carlo algorithm is shown in Fig. 4.4. As  $\epsilon$  decreases, more time steps and samples are required; the antithetic MLMC algorithm significantly lowers the amount of processing needed.

The relationship between the error control value  $\epsilon$  and the runtime of the simulation is found in Fig. 4.5. For larger values of  $\epsilon$ , the number of simulations decreases, but the runtime eventually plateaus. This is due to a combination of the work being insufficient to fully utilize the GPU and the overhead of transferring data to the GPU acting as a lower boundary to execution time.

## 4.5 Conclusion

Pricing of options is a challenging problem for the Financial industry and a very active field of research. Analytical solutions are available only for the smallest systems, and

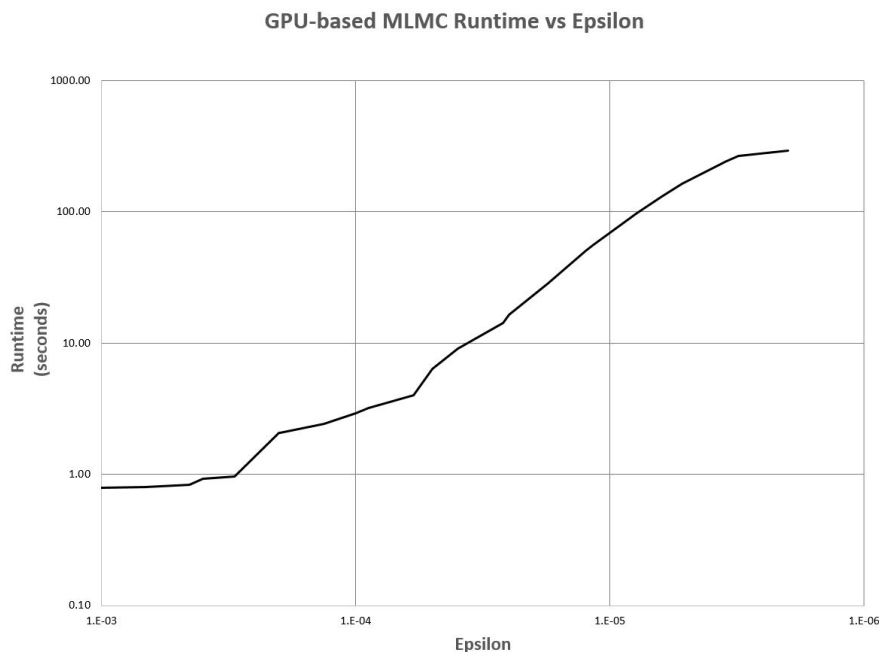


Figure 4.5: Runtime of the multilevel Monte Carlo for varying values of the error control variable  $\epsilon$

larger problems require simulations to estimate their results. The number of samples needed during a simulation is controlled by the variance which in turn controls the runtime.

Runtime can be reduced through the use of numerical and computational methods. Multilevel Monte Carlo accomplishes this by performing the Monte Carlo simulation using varying time steps. The initial level estimates the result using a large  $dt$  with many fast samples; subsequent levels further refine the result with more samples of smaller steps. The addition of the antithetic approach decreases variance by leveraging the well-known antithetic variate used with Monte Carlo.

The results obtained show that the antithetic multilevel Monte Carlo algorithm provides a significant improvement over Monte Carlo by reducing the variance in the simulation. Further improvement is gained through the use of the Milstein discretization rather than Euler-Maruyama.

Additional performance benefits are found by addressing the computational hardware upon which the simulations were performed. One major advantage of Monte Carlo is the independence of the samples. Since each sample does not depend on the results of other simulations, they can be performed in parallel. Currently Intel's Xeon Platinum 9282 processor has 56 cores, and AMD's EPYC 7002 processor has 64 cores. The price of more computing capability is heat, space, and cost. Researchers have found a solution by using GPUs. GPUs have the advantage of being denser, more energy efficient, and optimized for performing the same task in parallel over different data (SIMD). The fourteen NVIDIA K40s used in this research had 2880 cores each.

Applying numerical methods in a high performance computing environment can address the challenge of simulating the difficult problem of option pricing in the Financial sector. These tools have applications in other areas of science as well. The following chapter demonstrates this by applying GPU computing to the challenge of inventory management.

## CHAPTER V: APPLICATIONS IN INVENTORY MANAGEMENT

### 5.1 Introduction

Data are being generated today at phenomenal speeds, and transforming this torrent into actionable information is a challenge for many organizations. The volume of data is growing annually by 40%, and by 2020, there will be as many bits of data as there are stars in the universe [56]. With the introduction of intelligent devices able to transmit data over networks including the Internet, traditional systems may be unable to process the continuous stream of data. This Internet of Things (IoT) requires a new approach to data processing, especially in computing intensive areas such as machine learning, and has helped give birth to platforms such as Apache Hadoop [57] and Spark [58] and algorithms such as MapReduce [59] [60].

#### 5.1.1 Motivation

Inventory represents a significant investment for manufacturing and retail companies. The challenge is having the optimal levels of the needed products at the time the customer demands them, and there are many risks associated with inventory management. One such risk is the obsolescence of products due to evolving technology or shifting market needs. Unused inventory must be disposed of at a loss which in turn increases



the company's operating costs and makes it more difficult to compete in price sensitive markets [61].

Heavy equipment manufacturers and distributors face these challenges despite selling products designed to withstand decades of use. Changing market demands and economic conditions can leave manufacturers and distributors with fleets of equipment ranging in value from a hundred-thousand USD to over a million USD per unit. In addition to the equipment, distributors will stock the parts needed for maintenance and repairs; this inventory can be as substantial an investment as the equipment fleet. Finally, distributors will invest in hiring and training service personnel, purchase tooling, and build facilities to maintain and repair the customers' equipment.

Given the capital-intensive nature of the heavy equipment industry, developing accurate forecasts that can be generated in a timely means is of critical importance. If the manufacturer does not have the needed machine available in inventory, the time from ordering a machine, through its assembly, and to its delivery can span months (and often over a year for large mining machines). For these types of forecasts, the quantity of each model must be accurate months in advance to avoid unnecessary inventory and to ensure that the correct products are available when demanded.

The lead time for replenishing parts is significantly less (often less than a week), and the cost is a fraction of a single machine. However, stocking parts introduces new challenges. A single model of machine can consist of thousands of parts making it impractical in terms of space and capital to carry every part for every model of machine. Parts forecasting involves exponentially more items - over 100000 products versus 100 models of equipment. In addition, the frequency of the forecasts for parts is much higher - days or weeks.

Machines manufactured currently are equipped with sensors that monitor the operation of the internal systems and store this data for diagnostics. Becoming even

more common is near realtime connectivity between the machine, the distributors, and manufacturers using cellular and satellite communications. This stream of data provides the opportunity not only for identifying equipment failures before they occur but also for improving forecasting of parts and rental fleet machines. By predicting when a customer will likely bring a machine in for preventative maintenance or repair, a distributor could have a similar machine available for rent to minimize the customer's downtime. Since the parts necessary for preventative maintenance are known for each interval by model, forecasting the customer's equipment usage would allow the distributor to ensure the parts needed are available at the time the customer needs them along with a technician to perform the repair.

## 5.2 Exponential Smoothing Models

Time series data often exhibits three distinct components: (1) a trend, (2) seasonality, and (3) a random fluctuation [62]. Consider the time series data in Fig. 5.1 which shows the number of heavy equipment units delivered in a geographic region from January 2000 through April 2019. There are distinct trends in the data: from 2002 through 2006 there was a positive growth, from 2006 through 2009 the market contracted, and from 2009 through 2019 the market expanded.

Consider the time period from January 2010 through December 2012 shown in Fig. 5.2. The data displays an upward trend as highlighted by the linear regression described by

$$\hat{y}_t = 0.0057t + 1.0788, \text{ where } t = 0 \text{ for January 2010, } t = 1 \text{ for February 2010, etc.}$$

Dividing each actual data point ( $y_t$ ) by the predicted value ( $\hat{y}_t$ ) yields Fig. 5.3. From the chart of seasonal factors, it can be noticed that December has the highest factor for each of the three years while January has the lowest. The seasonality is most prominent when averaging the three years as seen in Fig. 5.4.

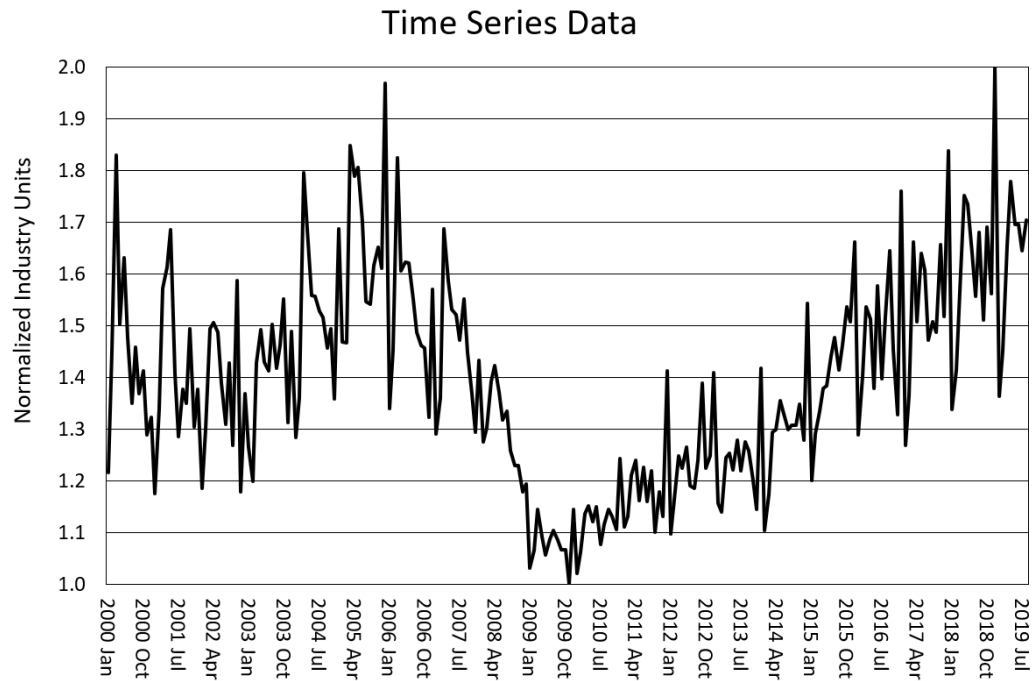


Figure 5.1: Time series data exhibiting trend, seasonality, and error.

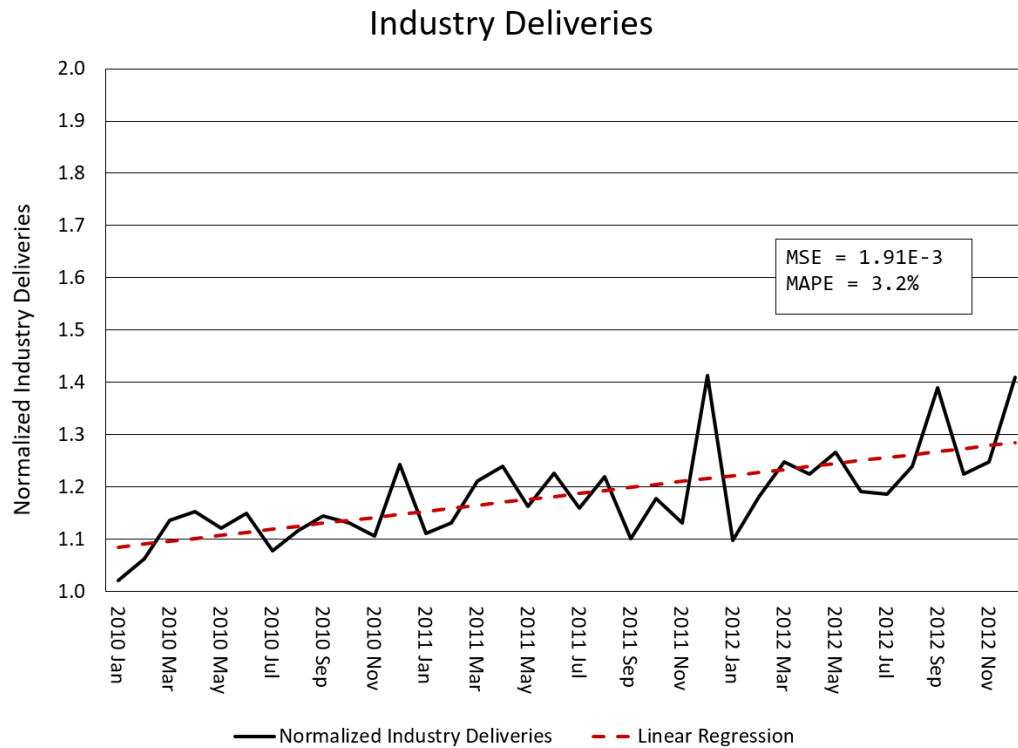


Figure 5.2: Normalized industry deliveries from January 2010 through December 2012 with linear regression.

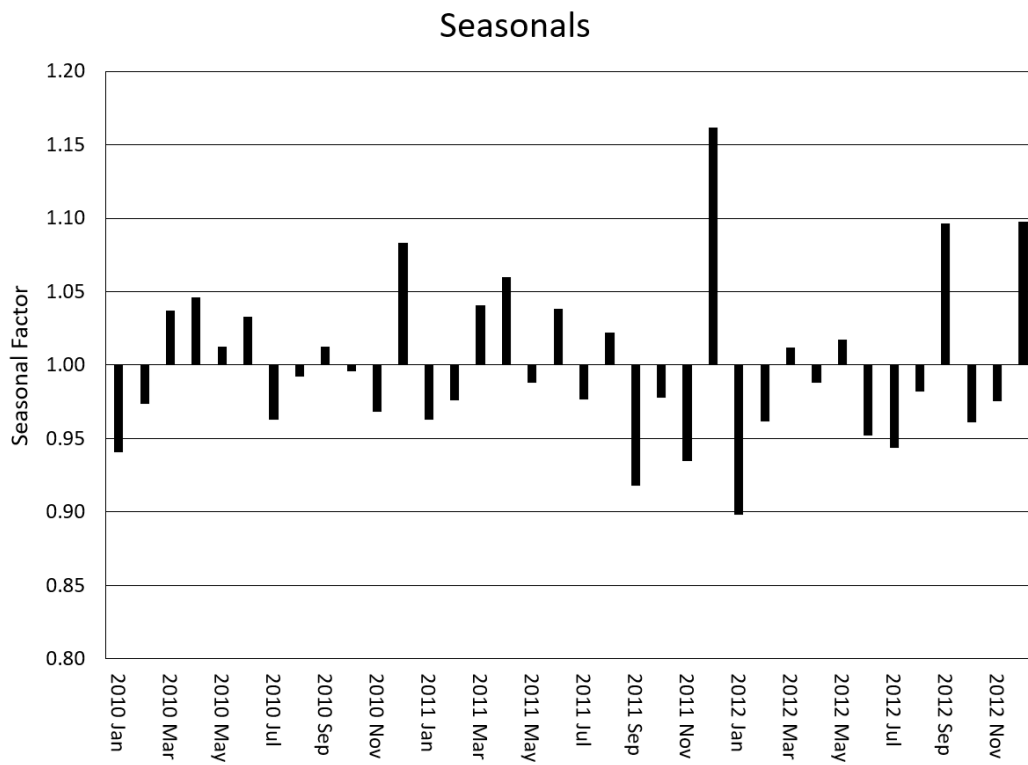


Figure 5.3: Seasonality factor for each period shown in Figure 5.2.

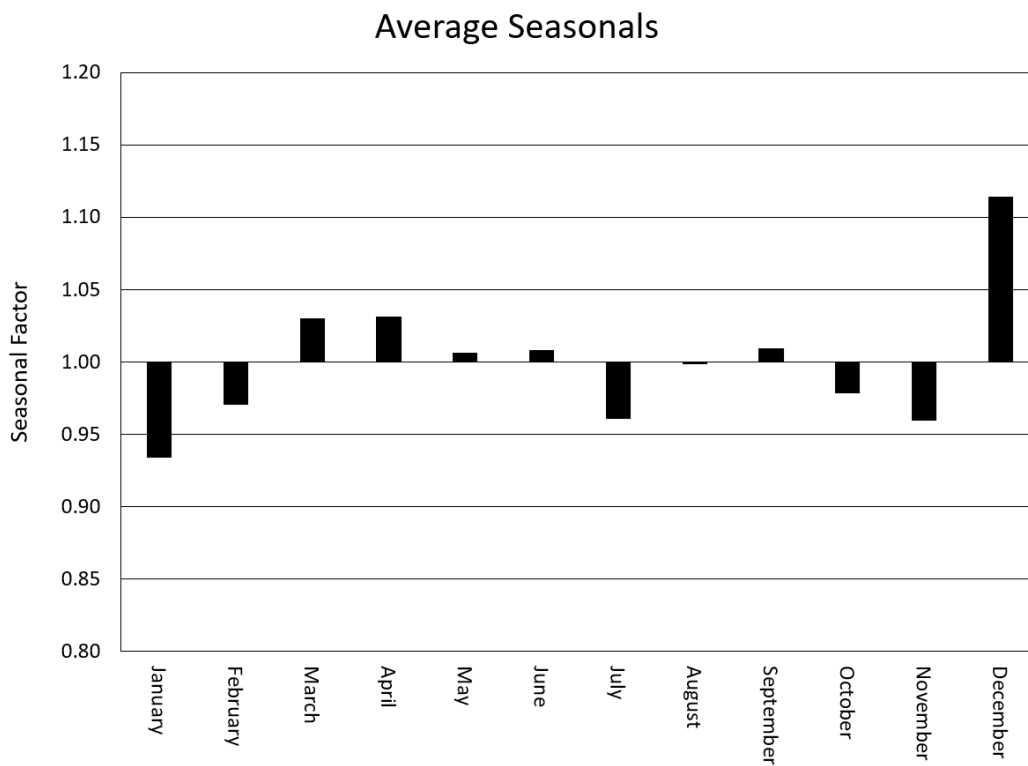


Figure 5.4: Average seasonality factor by period shown in Figure 5.3.

Simple linear regression can describe the trends but lacks the ability to express the seasonality that is often found in time series data. In addition, if the trend changes over time, linear regression will fail completely. More complex forms of regression such as nonlinear regression [63] [64] and deep learning [65] [66] can be applied to a time series to address this behavior.

David Wolpert is attributed with the metaphor of “no free lunch” in association with computing algorithms [67]. The more complex models becomes the greater is the need for additional computing resources. The quote “Remember that all models are wrong; the practical question is how wrong do they have to be to not be useful.” by George Box highlights the challenge of developing a model and the inherent error of using a mathematical model to describe a natural phenomenon [68]. The optimal balance between computing resources and model accuracy must be determined based on the error tolerance, the performance of the algorithm, and the cost of implementation.

### 5.2.1 Single Exponential Smoothing

Exponential smoothing was first presented by Robert Brown in 1957 [69]. Data is smoothed using a single weight ( $\alpha$ ) that blends the previously estimated value ( $\hat{y}_{t-1}$ ) and the current actual value ( $y_t$ ) to create a prediction at the current period ( $\hat{y}_t$ ):

$$\hat{y}_t = \alpha y_t + (1 - \alpha)\hat{y}_{t-1} \quad (5.1)$$

where  $y_t$  is the actual value,  $\hat{y}_t$  is the smoothed value,  $\alpha$  is the weight, and  $0 < \alpha < 1$ ,  $\hat{y}_0 = y_0$ .

Consider the data in Fig. 5.1. Using  $\alpha = 0.1$  yields the fit shown in Fig. 5.5 [70]. The choice of  $\alpha$  in Eqn. 5.1 determines the weight historical data has on current predictions. The calculations for fitting existing data is detailed in Table 5.1. Higher

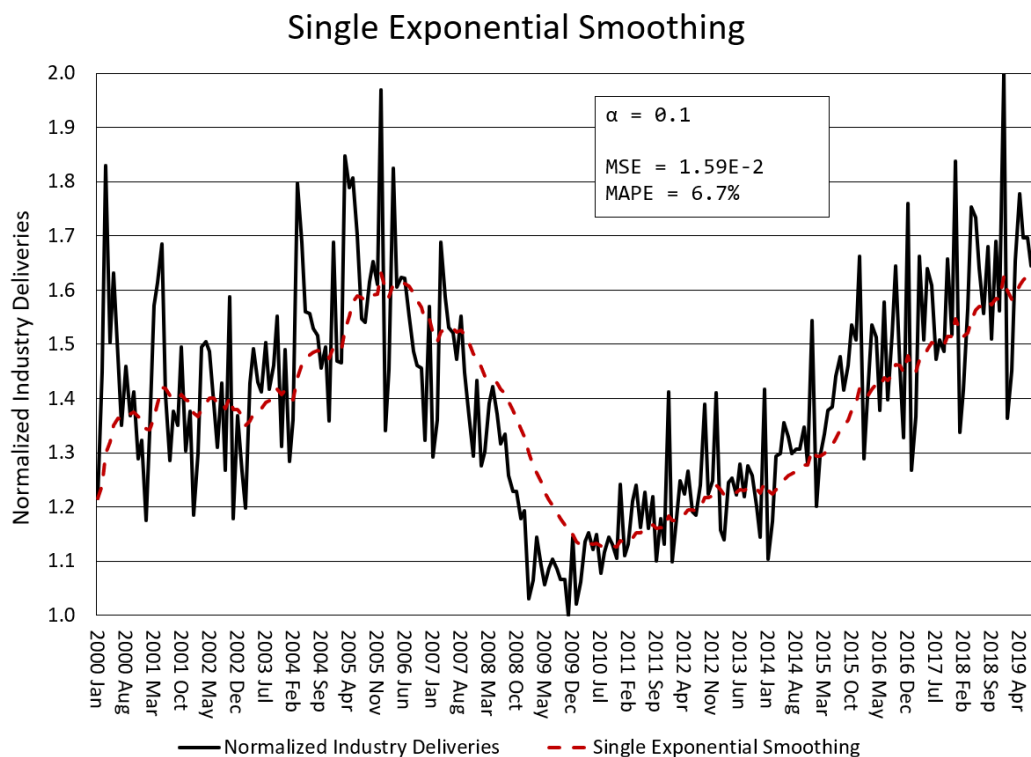


Figure 5.5: Fit of single exponential algorithm to actual data with  $\alpha = 0.1$

values of  $\alpha$  place greater emphasis on current values and reduces the importance of prior predictions. Fig. 5.6 demonstrates the relationship between  $\alpha$  and two error calculations: mean-squared error (MSE) and mean absolute percentage error (MAPE) [71]. As  $\alpha$  increases, more weight is placed on the actual current value rather than the prior prediction resulting in overfitting.

The prior example used the single exponential smoothing to fit a model to known

Table 5.1: An example of single exponential smoothing for a subset of the data shown in Fig. 5.2.

Period ( $t$ )	Units ( $y_t$ )	Prediction ( $\hat{y}_t$ )
2016 Jan	1.289	1.289
2016 Feb	1.402	$0.1 \times 1.402 + (1 - 0.1) \times 1.289 = 1.300$
2016 Mar	1.536	$0.1 \times 1.536 + (1 - 0.1) \times 1.300 = 1.324$
2016 Apr	1.513	$0.1 \times 1.513 + (1 - 0.1) \times 1.324 = 1.343$
2016 May	1.379	$0.1 \times 1.379 + (1 - 0.1) \times 1.343 = 1.346$
2016 Jun	1.577	$0.1 \times 1.577 + (1 - 0.1) \times 1.346 = 1.369$

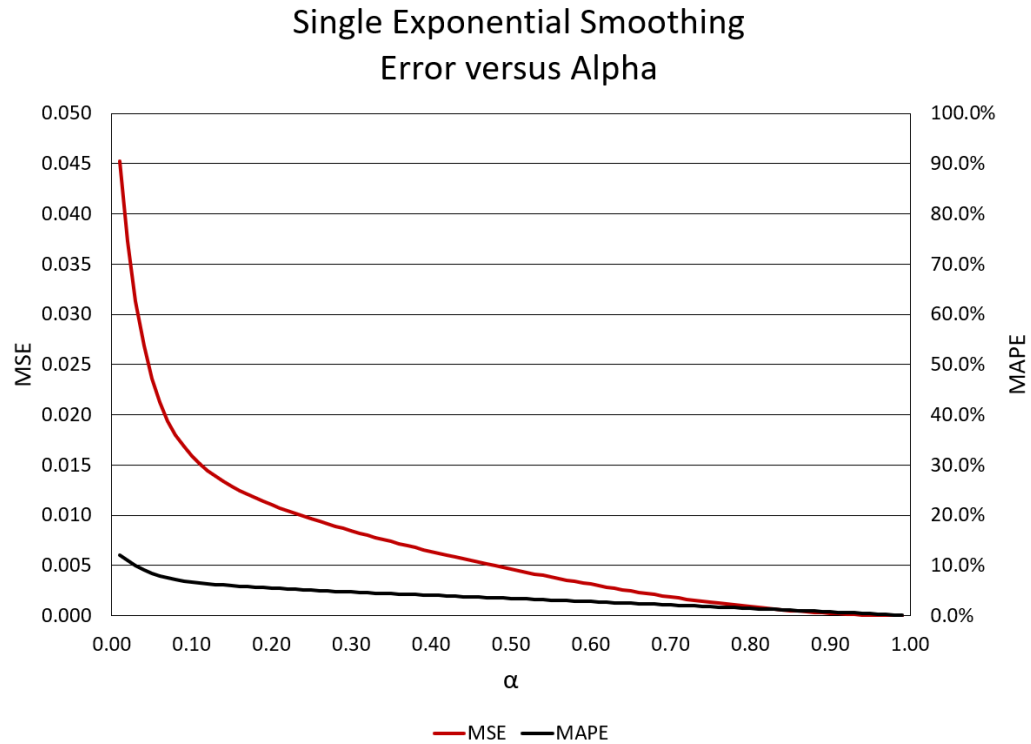


Figure 5.6: Error versus  $\alpha$  for single exponential smoothing

Table 5.2: Forecasting using the single exponential smoothing

Period ( $t$ )	Units ( $y_t$ )	Forecast ( $\hat{y}_t$ )	Calculation
2004 Dec	1.688	1.495	
2005 Jan	1.469	1.514	$0.1 \times 1.688 + (1 - 0.1) \times 1.495 = 1.514$
2005 Feb	1.466	1.532	$0.1 \times 1.688 + (1 - 0.1) \times 1.514 = 1.532$
2005 Mar	1.848	1.547	$0.1 \times 1.688 + (1 - 0.1) \times 1.532 = 1.547$

data. The model uses prior prediction and current actual values to generate the current prediction. When creating forecasts, the future values are unknown. This problem is solved by using the last known actual value during the forecast process.

In Fig. 5.7 the data has an upward trend. The last actual data point known by the algorithm was 1.688 units in December 2004. The calculations for the first three forecasts are performed in Table 5.2 and result in a MSE of  $2.88 \times 10^{-2}$  and MAPE of 7.0% over the twelve months of forecasts.

The problem of using the single exponential smoothing method with data exhibiting

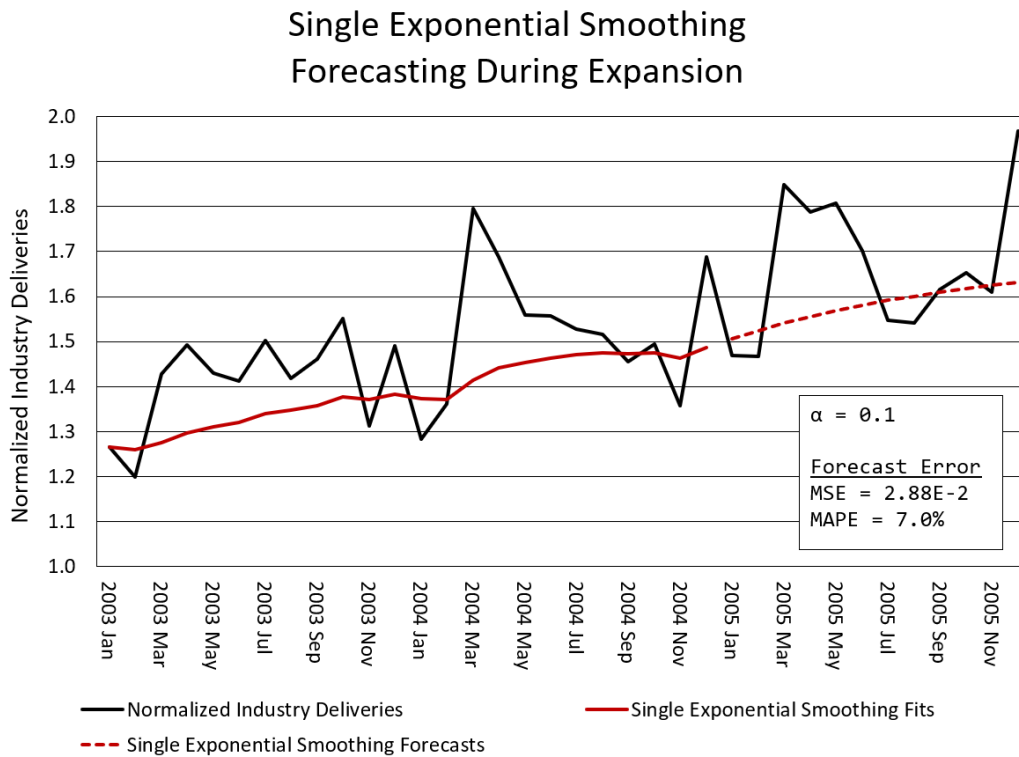


Figure 5.7: The single exponential smoothing forecast during expansion period.

a trend can best be observed during a contraction period. In the prior example, the asymptotic behavior caused the system to increase and plateau which approximately mimicked the data.

Consider Fig. 5.8 in which the trend is negative. The forecast increases slightly before converging on the asymptote of 1.43299 units, the last actual value seen by the algorithm. However, the actual data continues to decrease. The *generalized reduced gradient* (GRG) solver finds the minimum error at  $\alpha = 0.39$ ; the MSE is  $2.86 \times 10^{-2}$  and the MAPE is 12.2%, both significantly higher than the expansion example.

Single exponential smoothing works well when the data lacks a trend. For time series data with more complex behavior, the model must be expanded to include additional features.



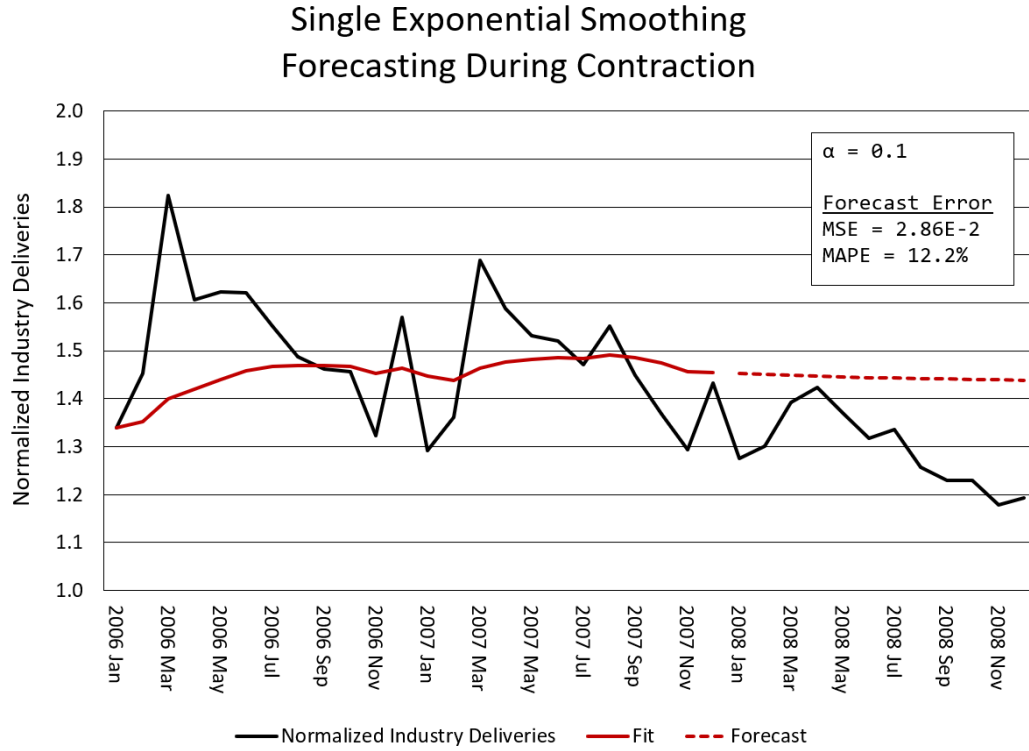


Figure 5.8: The single exponential smoothing forecast during contraction period.

## 5.2.2 Double Exponential Smoothing

Holt extended the original single exponential smoothing model to introduce a method to address trend in a time series [72–74]. The new model requires two weights  $\alpha$  and  $\gamma$ :

$$\hat{y}_t = l_{t-1} + b_{t-1}$$

$$l_t = \alpha y_t + (1 - \alpha)(l_{t-1} + b_{t-1})$$

$$b_t = \gamma(l_t - l_{t-1}) + (1 - \gamma)b_{t-1}$$

The initial values of  $l_0$  and  $b_0$  are found by performing a linear regression through a

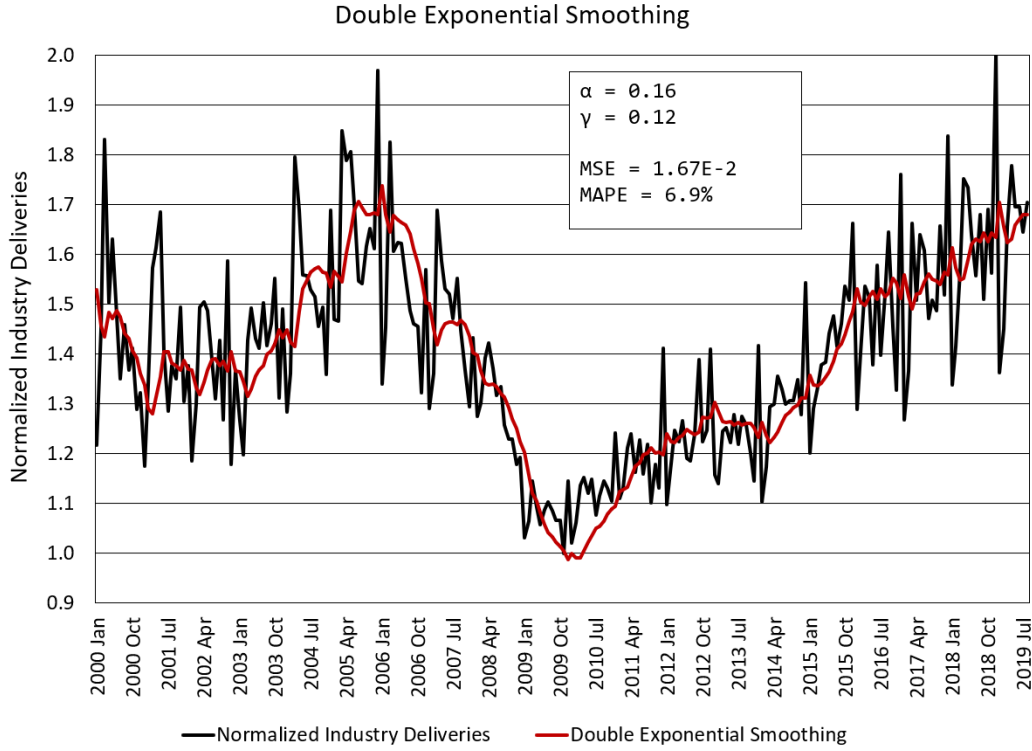


Figure 5.9: Double exponential smoothing.

sample of the data. For Fig. 5.9, the first twelve data points were used. The intercept is used for the initial level  $l_0$ ; the slope is used for the initial trend  $b_0$ . A comparison of the fits with those of the single exponential smoothing fits in Fig. 5.5 reveal that the double exponential smoothing model follows the actual data closer.

The calculations for the period from January 2016 to June 2016 are shown in Table 5.3;  $l_0$  and  $b_0$  are estimated by using linear regression through the data from January 2015 through December 2015. The level for each subsequent step is a blend between the current actual value and the prediction controlled by the parameter  $\alpha$ . The trend at each time step is controlled by the smoothing parameter  $\beta$ .

Fig. 5.10 demonstrates the performance of the double exponential smoothing method during a period of expansion using  $\alpha = 0.1$  and  $\gamma = 0.1$ . The MSE of  $2.46 \times 10^{-2}$  is slightly better than that of the single exponential method, but the MAPE of the double exponential smoothing is slightly worse at 7.2%. The

Table 5.3: An example of double exponential smoothing for a subset of the data shown in Fig. 5.2.

Period	$y_t$	$\hat{y}_t$	Calculation
			$l_0 = 1.373$
			$b_0 = 0.017$
2016 Jan	1.289	1.390	$l_1 = 0.1 \times 1.289 + (1 - 0.1) \times 1.390 = 1.380$
			$b_1 = 0.1 \times (1.380 - 1.373) + (1 - 0.1) \times 0.017 = 0.016$
2016 Feb	1.402	1.396	$l_2 = 0.1 \times 1.302 + (1 - 0.1) \times 1.396 = 1.396$
			$b_2 = 0.1 \times (1.396 - 1.380) + (1 - 0.1) \times 0.016 = 0.016$
2016 Mar	1.536	1.412	$l_3 = 0.1 \times 1.536 + (1 - 0.1) \times 1.412 = 1.425$
			$b_3 = 0.1 \times (1.425 - 1.396) + (1 - 0.1) \times 0.016 = 0.017$
2016 Apr	1.513	1.442	$l_4 = 0.1 \times 1.513 + (1 - 0.1) \times 1.442 = 1.449$
			$b_4 = 0.1 \times (1.449 - 1.425) + (1 - 0.1) \times 0.017 = 0.018$
2016 May	1.379	1.467	$l_5 = 0.1 \times 1.379 + (1 - 0.1) \times 1.467 = 1.458$
			$b_5 = 0.1 \times (1.458 - 1.449) + (1 - 0.1) \times 0.018 = 0.017$
2016 Jun	1.577	1.475	$l_6 = 0.1 \times 1.577 + (1 - 0.1) \times 1.475 = 1.485$
			$b_6 = 0.1 \times (1.485 - 1.458) + (1 - 0.1) \times 0.017 = 0.018$

Table 5.4: Forecasting using the double exponential smoothing

Period	$y_t$	$\hat{y}_t$	Calculation
2004 Dec	1.688	1.534	$l_0 = 1.558$
			$b_0 = 0.007$
2005 Jan	1.469	1.566	$l_1 = 1.566$
			$b_1 = 0.007$
2005 Feb	1.466	1.573	$l_2 = 1.573$
			$b_2 = 0.007$
2005 Mar	1.848	1.580	$l_3 = 1.580$
			$b_3 = 0.007$

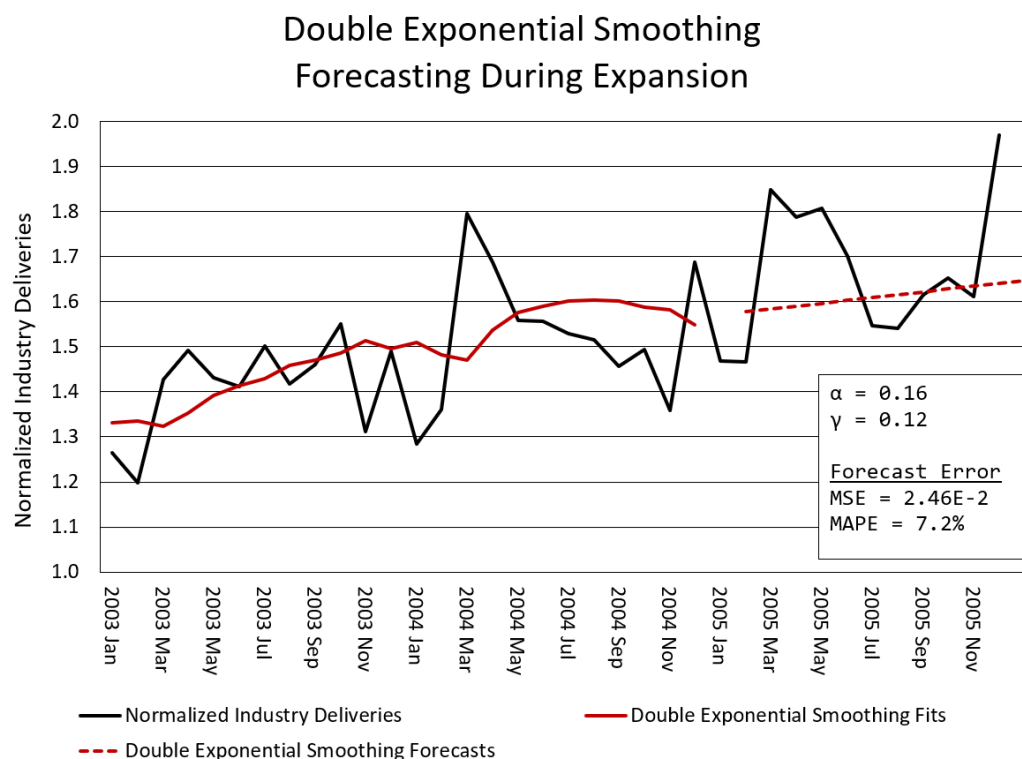


Figure 5.10: The double exponential smoothing forecast during expansion period.

calculations for the first three months are performed in Table 5.4.

Of more interest is the difference in performance of the single and double exponential methods during the contraction period. Since the double exponential method includes a trend term, it can be surmised that it will be better at forecasting. Fig. 5.11 confirms this. The MSE is  $1.06 \times 10^{-2}$  and the MAPE is 7.1% compared to the single exponential smoothing results of  $2.86 \times 10^{-2}$  and 12.2%.

The relationship between the choice of  $\alpha$  and  $\gamma$  is plotted in Fig. 5.12 and Fig. 5.13. The parameters are varied over the domain  $0 < \alpha < 1$  and  $0 < \gamma < 1$ , and the  $\log(\text{MSE})$  of the fits over Fig. 5.1 is calculated. The minimal error is found at  $\alpha = 0.16$  and  $\gamma = 0.12$ .

Closer inspection of the data exposes that certain months are always higher than others while other months are consistently less. The double exponential smoothing

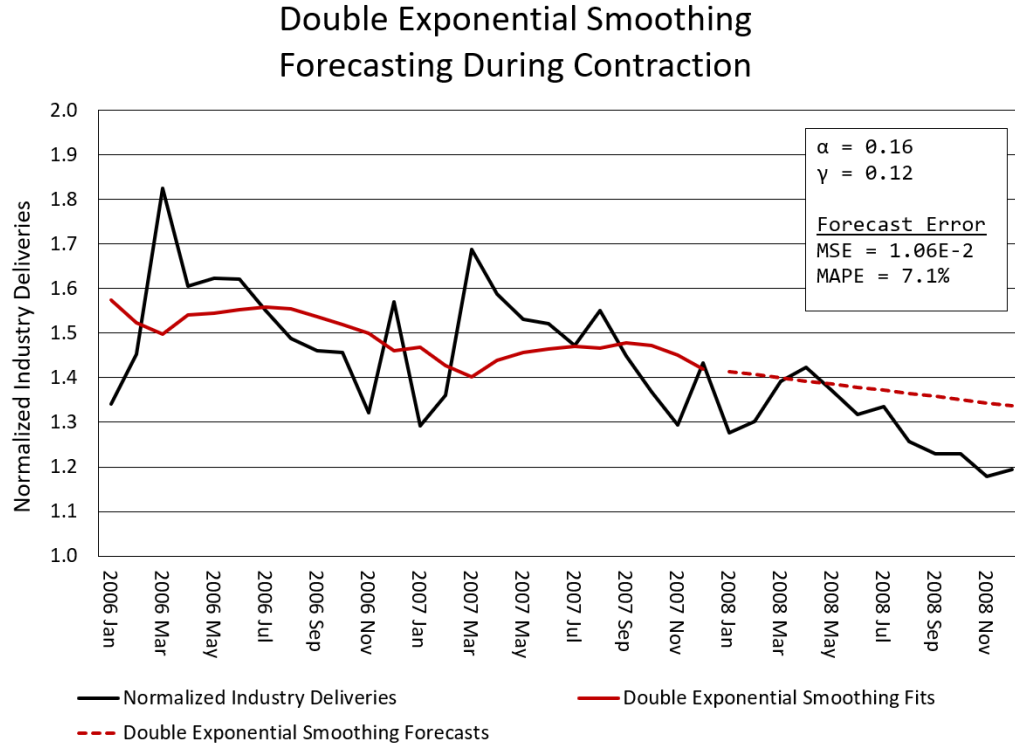


Figure 5.11: The double exponential smoothing forecast during contraction period.

model does not describe this regular *seasonality*.

### 5.2.3 Triple Exponential Smoothing

The double exponential smoothing model was modified in [75] and [72] to include a weighted trend component and a weighted seasonal component. A model utilizing multiplicative seasonals can be described as:

$$\begin{aligned}
 l_t &= \alpha \left( \frac{x_t}{s_t} \right) + (1 - \alpha)(l_{t-1} + b_{t-1}) \\
 s_{t+L} &= \beta \left( \frac{x_t}{l_t} \right) + (1 - \beta)s_t \\
 b_t &= \gamma(l_t - l_{t-1}) + (1 - \gamma)b_{t-1} \\
 \hat{x}_t &= (l_t + b_t)s_t
 \end{aligned}$$

## Double Exponential Smoothing Error versus Alpha, Gamma

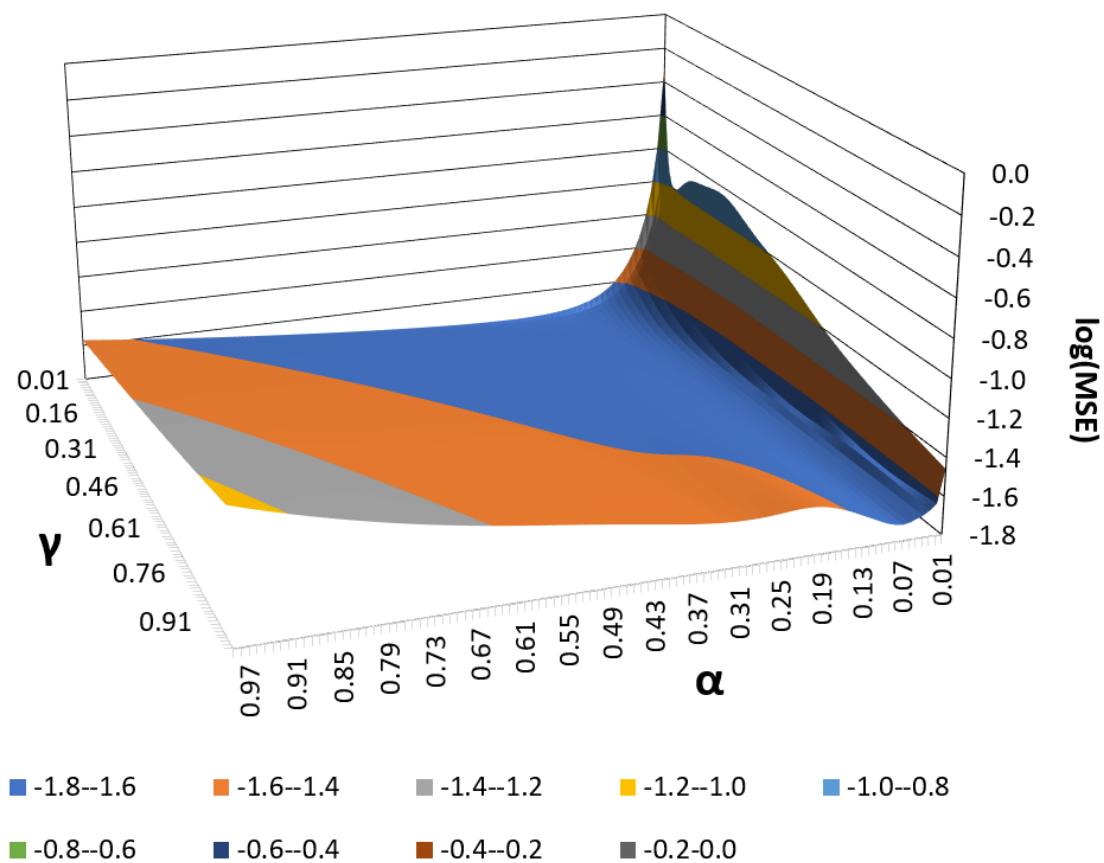


Figure 5.12: Double exponential smoothing error versus  $\alpha$ ,  $\gamma$

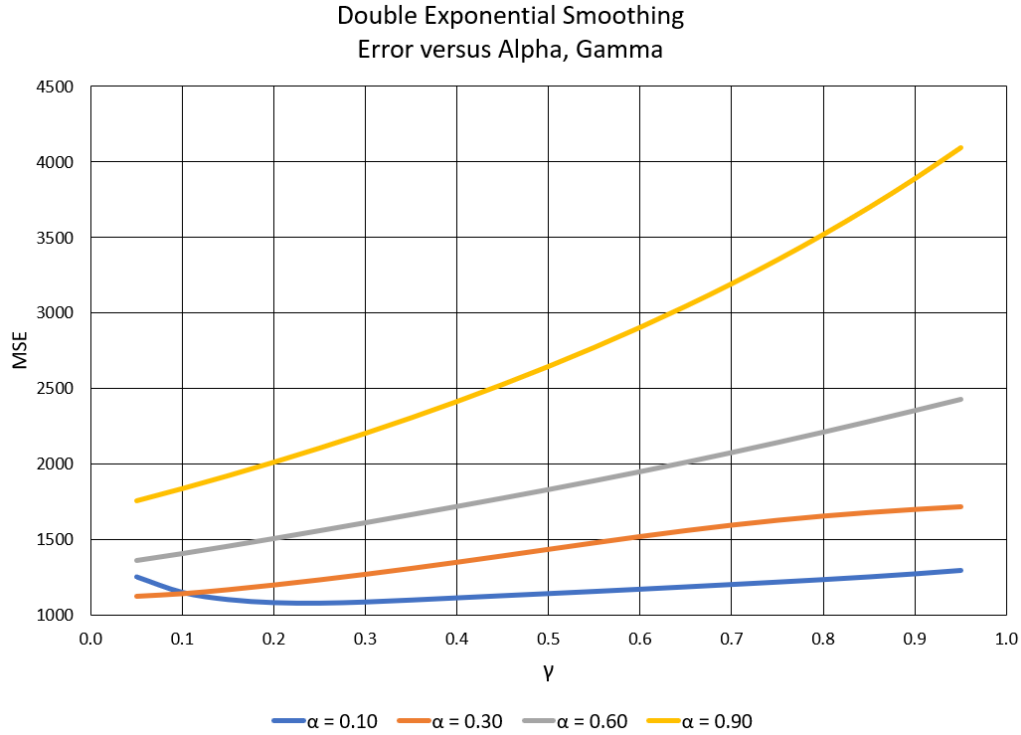


Figure 5.13: Error versus  $\alpha$ ,  $\gamma$

where  $l_t$  is the level,  $b_t$  is the trend,  $s_t$  is the seasonal factor, and  $L$  is the periodicity of the seasonals.  $\alpha$ ,  $\beta$ , and  $\gamma$  are the weights with  $0 < \alpha < 1$ ,  $0 < \beta < 1$ , and  $0 < \gamma < 1$ . This model is applied to the time series data and plotted in Fig. 5.14. Of the three models, triple exponential smoothing has the lowest error for MSE and MAPE.

The algorithm for this model is explained in detail in section 5.3; the calculations for the months of January 2016 through March 2016 are detailed in Table 5.5. For the same period of contraction sampled earlier, the triple exponential model outperforms the others with an MSE of  $9.06 \times 10^{-3}$  and an MAPE of 4.4% as shown in Fig. 5.17. The first three forecasts are demonstrated in Table 5.6. However, the contraction period in Fig. 5.18 is worse than the double exponential mode with an MSE of  $1.51 \times 10^{-2}$  and an MAPE of 8.1% due to the dampened seasonality when compared to the prior two years.

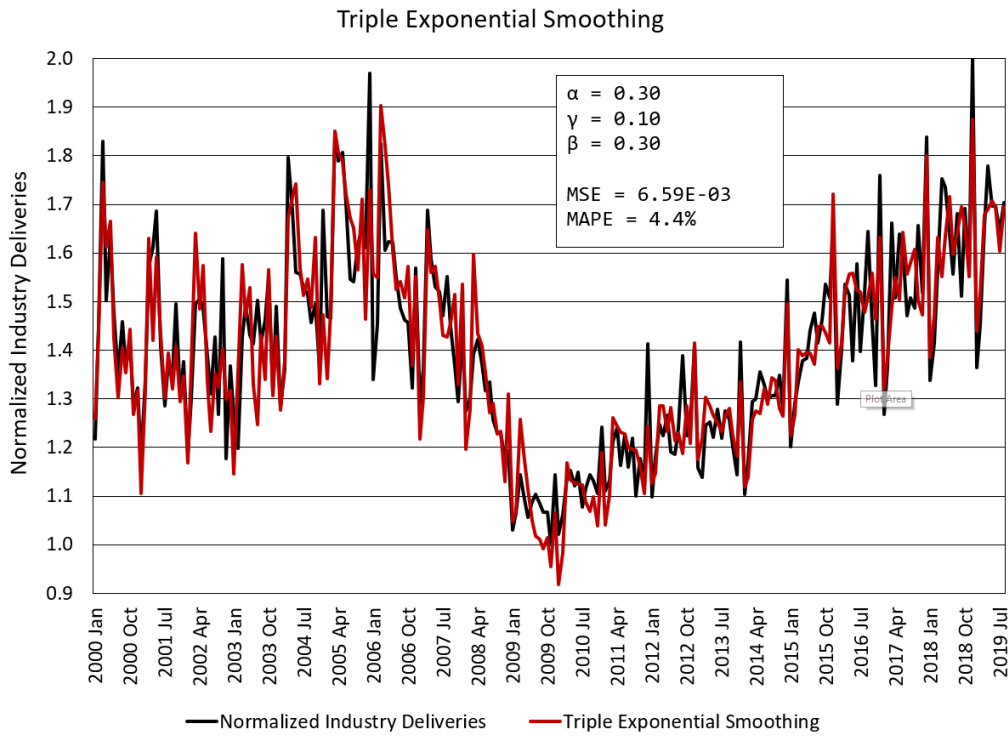


Figure 5.14: Triple exponential smoothing

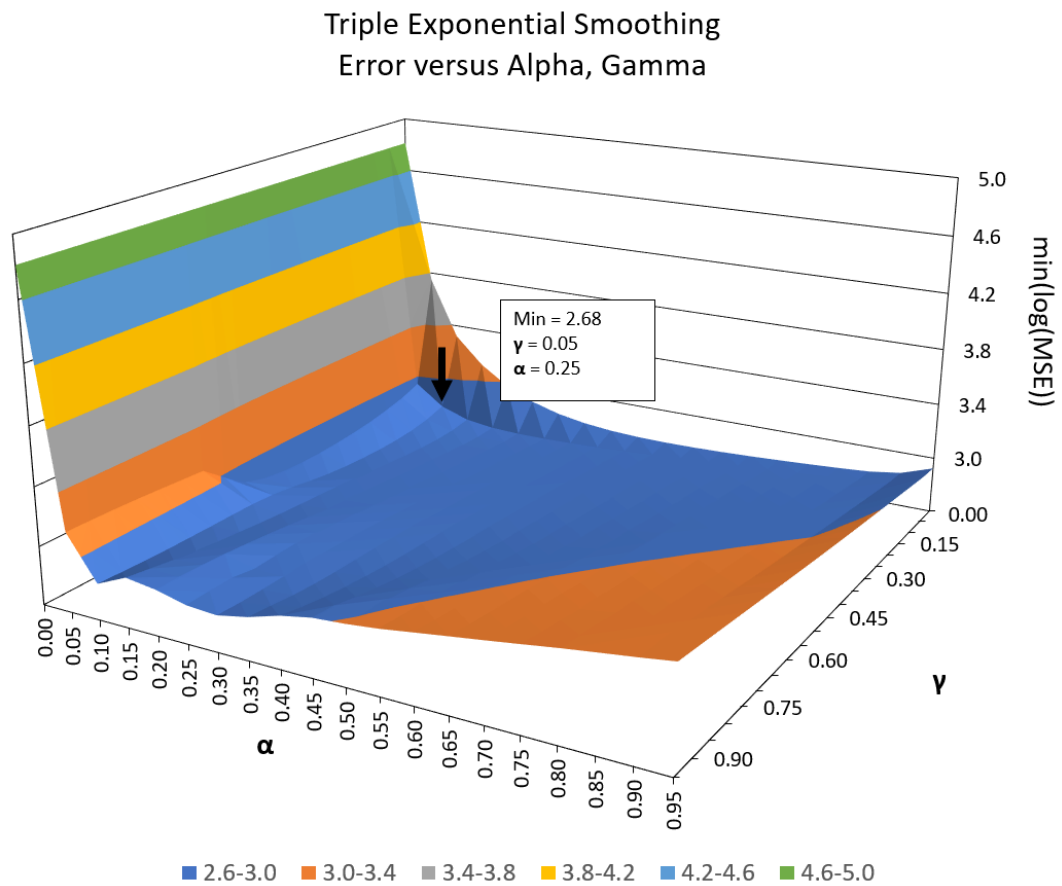
Table 5.5: An example of triple exponential smoothing for a subset of the data shown in Fig. 5.2.

Period	$y_t$	$\hat{y}_t$	Calculation
2016 Jan	1.289	1.211	$l_0 = 1.373$
			$b_0 = 0.017$
			$s_1 = 0.871$
			$\hat{y}_1 = 0.871(1.373 + 0.017) = 1.211$
			$l_1 = 0.1(1.289/0.871) + 0.9(1.373 + 0.017) = 1.399$
2016 Feb	1.402	1.331	$b_1 = 0.1(1.399 - 1.373) + 0.9(0.017) = 0.018$
			$s_2 = 0.939$
			$l_2 = 0.1(1.402/0.939) + 0.9(1.339 + 0.018) = 1.424$
2016 Mar	1.536	1.557	$b_2 = 0.1(1.396 - 1.380) + (1 - 0.1) \times 0.016 = 0.016$
			$s_3 = 1.079$
			$l_3 = 0.1(1.536/1.079) + 0.9(1.424 + 0.016) = 1.441$
			$b_3 = 0.1(1.462 - 1.424) + 0.9(0.016) = 0.018$



Table 5.6: Forecasting using triple exponential smoothing

Period	$y_t$	$\hat{y}_t$	Calculation
2004 Dec	1.688		$l_0 = 1.488$
			$b_0 = 0.001$
2005 Jan	1.469		$s_1 = 0.858$
			$l_1 = 1.490$
			$b_1 = 0.001$
2005 Feb	1.466		$s_2 = 0.914$
			$l_2 = 1.491$
			$b_2 = 0.001$
2005 Mar	1.848		$s_3 = 1.137$
			$l_3 = 1.493$
			$b_3 = 0.001$

Figure 5.15:  $\min(\log(\text{MSE}))$  for  $\alpha$  and  $\gamma$

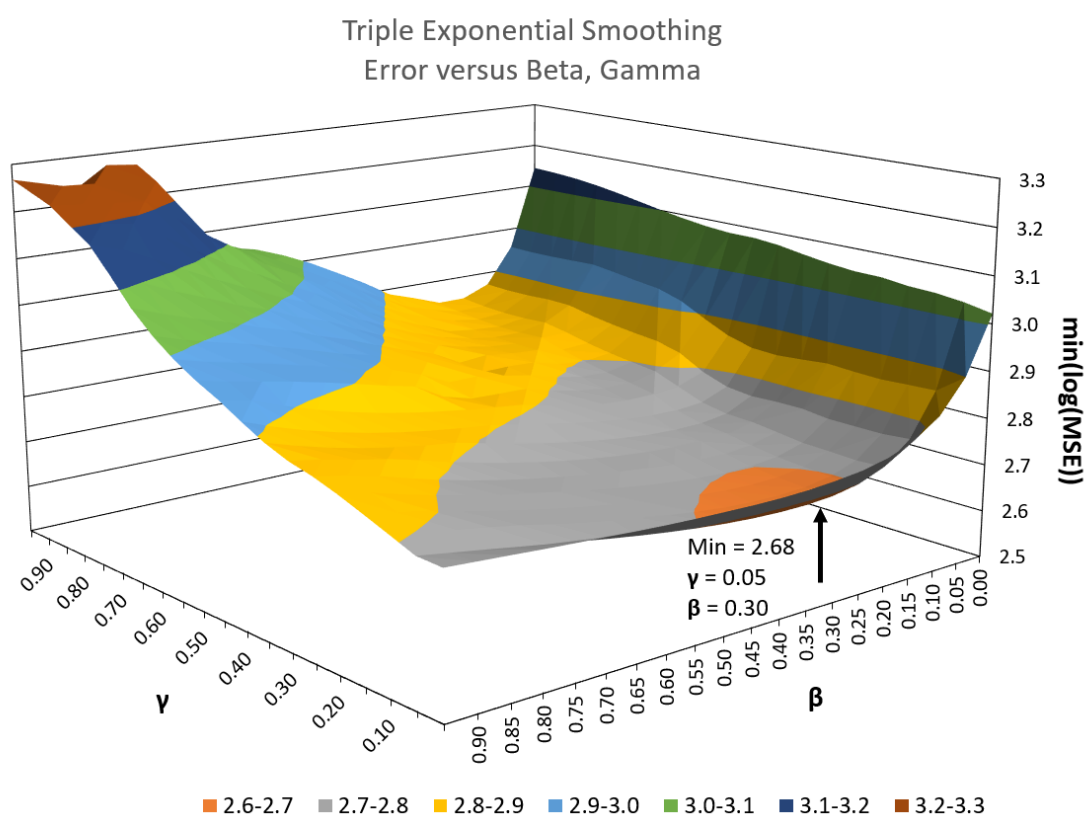


Figure 5.16:  $\min(\log(\text{MSE}))$  for  $\beta$  and  $\gamma$

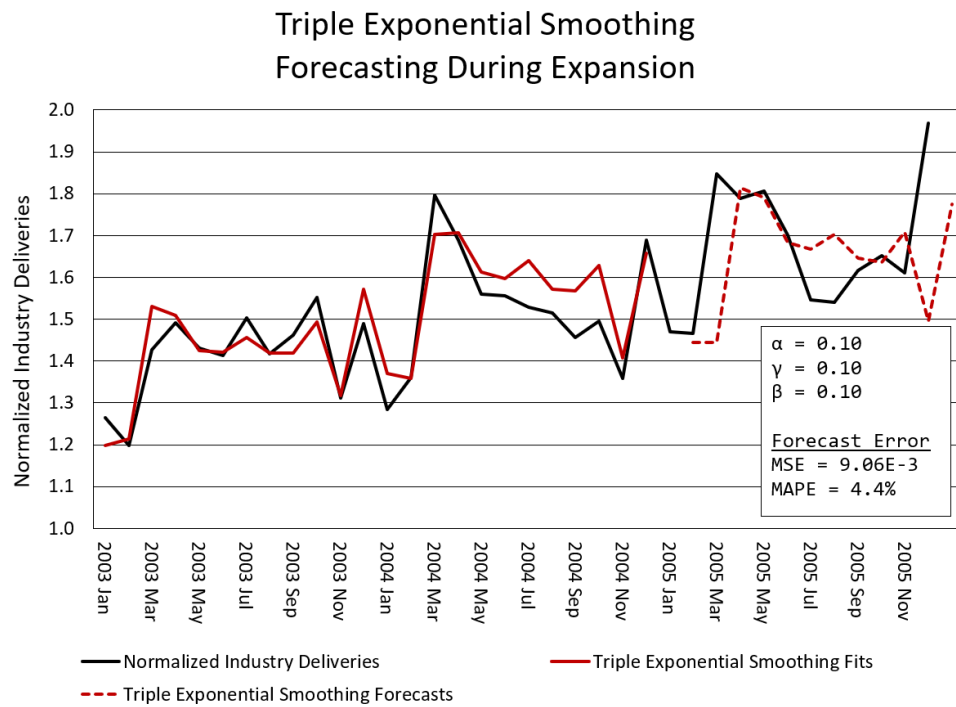


Figure 5.17: Triple exponential smoothing forecast during expansion period

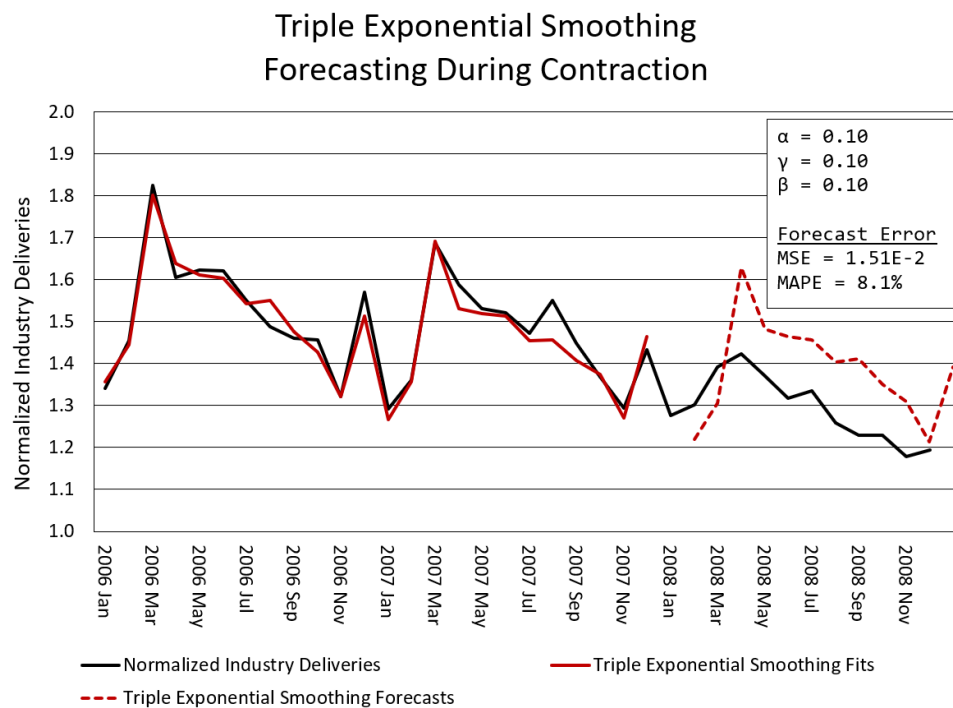


Figure 5.18: Triple exponential smoothing forecast during contraction period

The relationship between the parameters  $\alpha$ ,  $\gamma$ , and  $\beta$  are explored in Fig. 5.15 and Fig. 5.16. For the pair  $\alpha$  and  $\gamma$ , the minimal error is found at  $\alpha = 0.25$  and  $\gamma = 0.05$ , while the pair  $(\gamma, \beta)$  has a minimal error at  $\gamma = 0.05$  and  $\beta = 0.30$ .

## 5.3 Implementation

The algorithms presented above share one common trait - the inherent lack of efficient parallelism. Future values depend on the calculations for previous time periods, and the operations within the algorithm offer only minimal opportunities. However, if multiple items must be forecast simultaneously, then multiple threads can be exploited to improve performance.

The time series forecasting code uses a triple exponential smoothing algorithm outlined in Algorithm 15. To illustrate the core algorithm, consider a subset of actual data shown in Fig. 5.19. It assumes that the data contains regular cycles of  $N$  data points and at least two full cycles ( $2N$ ) are present.

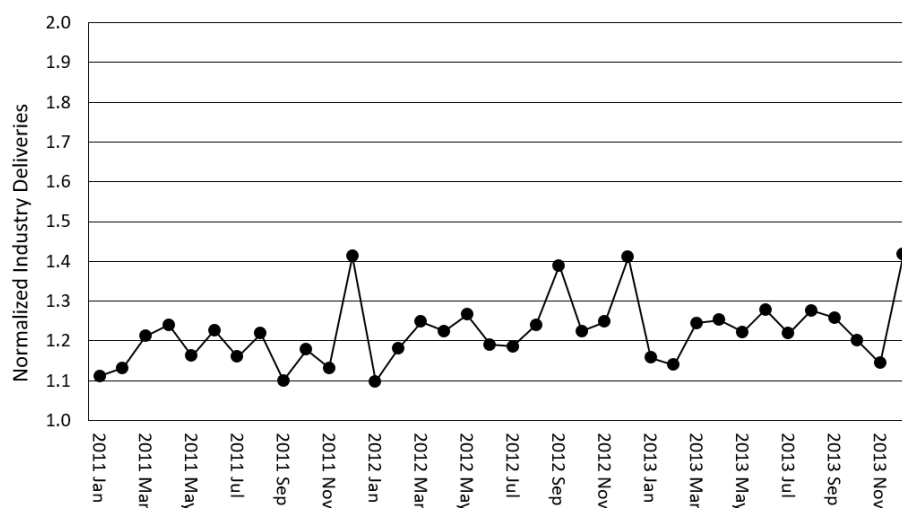


Figure 5.19: Actual data used in the example taken with permission from Thompson Machinery's industry activity

The first  $N$  data points are taken from the data set, and linear regression is

performed through them to estimate the initial trend (slope) and level (intercept). Next linear regression is performed through the first  $2N$  data points. The difference between the regression and the actual data becomes the initial seasonal; there will be  $N$  seasonal values. Because there are two complete cycles, the initial value is the average of the two occurrences. In the example data, the cycle is twelve months long. Therefore, the initial seasonal estimate for period 1 (January in the test data) is  $s_1 = (p_1 + p_{13})/2$ . In a multiplicative model, the seasonals are calculated by dividing the actual data by the estimated value from the regression; in an additive model the seasonals are the difference between the actual value and the estimated value. The selection of the correct approach is often gleaned from knowledge of the process.

With the initial estimate for the level, trend, and seasonals, the algorithm performs the smoothing function by adding the level to the trend and multiplying by the seasonal to generate a *fit*. The algorithm uses three control variables:  $\alpha$ ,  $\beta$ , and  $\gamma$ . The variable  $\alpha$  controls how much of the new level is taken from the calculated level and how much is taken from the actual level of the next step (line 22 in Algorithm 15).  $\beta$  controls the balance between calculated values and actual values for seasonality (line 23). Finally  $\gamma$  sets the balance between the calculated trend and actual trend (line 24).

Once the exponential smoothing algorithm has fitted the “training data,” the result is a final trend, level, and set of seasonals. This can be used to forecast the next  $N$  time periods using a modified slope-intercept equation  $y = s_p(mx + b)$ . The results of applying this to the sample data are shown in Fig. 5.21.

It is evident that the algorithm used for triple exponential smoothing can not be parallelized due to the fact that the next calculation is dependent on the prior calculation. However, when there are numerous data sets to be processed, the algorithm can be executed in parallel over different sets of data. This holds true

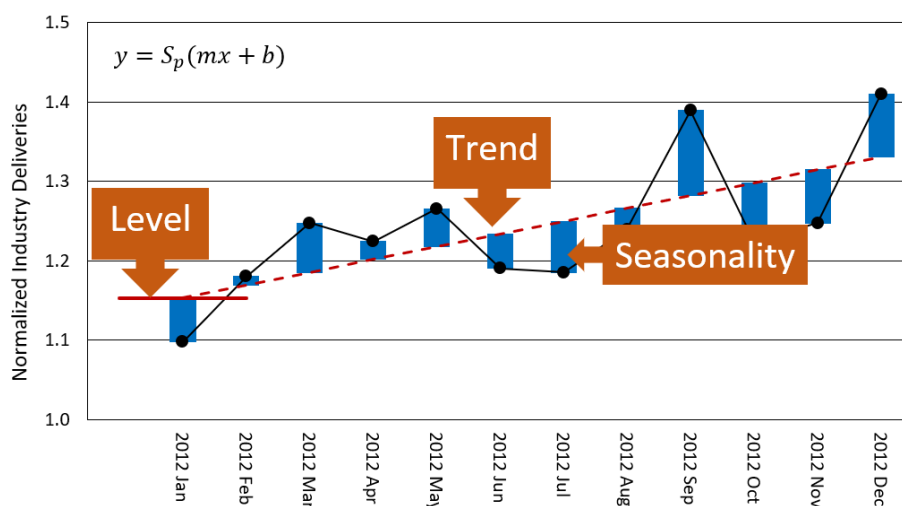


Figure 5.20: Components of triple exponential smoothing

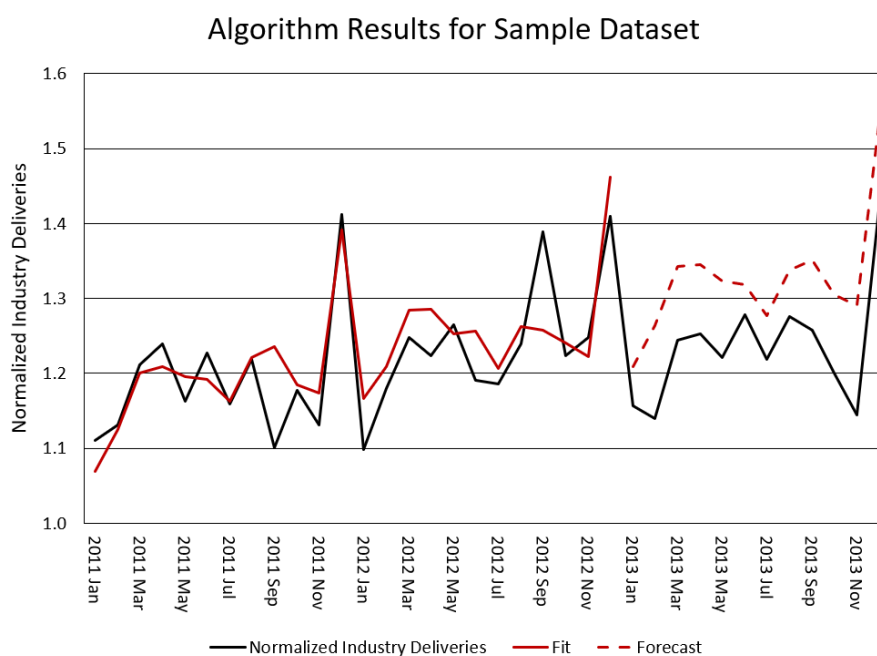


Figure 5.21: Results of triple exponential smoothing on subset of data

when testing the algorithm by sliding it through a series of actual data as shown in Fig. 5.22.

Using the MPI implementation of the forecast tool, the entire dataset is made available to all nodes of the cluster. Node 0 interrogates every other node for the number of GPU's to calculate the capacity of the cluster. If there are a total of  $M$

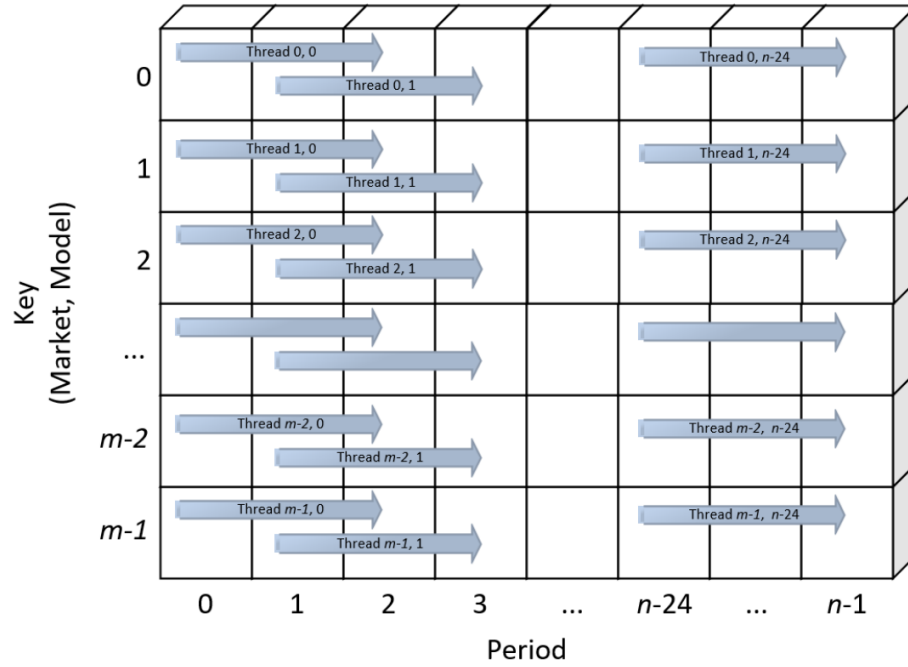


Figure 5.22: GPU Launch for triple exponential smoothing showing different threads performing the algorithm over different periods of time and data sets

nodes in the cluster and a total of  $G$  GPU's (assume that each node has the same number of GPU's for this example), then each node must work on  $G/M$  of the total problem. In Fig. 5.22 there are  $m$  rows of data that are independent. So each node will process  $m/(G/M)$  rows. However, it is important to understand the accuracy of the forecast model which can be attained by forecasting earlier data and comparing it to data later in the set.

This is illustrated in Fig. 5.22. Thread 0 will process the first  $2N$  periods of data which generates forecasts for  $2N + 1$  through  $2N + 12$ . If the time period of the data set is larger than the required  $2N$  periods, these forecasts can be compared to the actual data. Thread 1 will process  $N + 1$  through  $N + 12$  to generate forecasts for  $2N + 2$  through  $2N + 13$ , etc.

---

**Algorithm 15** Implemented triple exponential smoothing for generating forecasts
 

---

```

    { $N$  is the number of periods in a cycle}
2: {Assume  $2N$  actual values}
    {Calculate initial trend and level}
4:  $x = 1 \dots N$ 
     $y =$  first  $N$  actual values
6: Perform linear regression for  $x$  and  $y$ 
     $L_0 =$  intercept from regression {Initial level}
8:  $T_0 =$  slope from regression {Initial trend}
    {Calculate initial seasonals}
10:  $x = 1 \dots 2N$ 
     $y =$  All  $2N$  actual values
12: Perform linear regression for  $x$  and  $y$ 
    { $\hat{y}$  are the fits from the regression}
14:  $S_p = [(\hat{y}_p/y_p) + (\hat{y}_{p+N}/y_{p+N})]/2$  for  $1 \leq p \leq N$ 

16: {Perform smoothing of actuals}
     $L_p = L_0$ 
18:  $T_p = T_0$ 
    for  $p = 1 \dots 2N$  do
20:    { $\hat{s}$  are the smoothed  $y$ 's}
         $\hat{s}_p = S_p(L_p + T_p)$ 
22:     $L_{p+1} = \alpha(y_p/S_p) + (1 - \alpha)(L_p + T_p)$ 
         $S_{p+N} = \beta(y_p/L_p) + (1 - \beta)S_p$ 
24:     $T_{p+1} = \gamma(L_{p+1} - L_p) + (1 - \gamma)T_p$ 
    end for
26:    {Perform forecasts}
28: for  $p = 2N + 1 \dots 2N + 12$  do
         $\hat{s}_p = S_p(L_p + T_p)$ 
30:     $L_{p+1} = L_p + T_p$ 
    end for
  
```

---

## 5.4 Validation

The high performance forecasting algorithm developed during this research is currently in production at Thompson Machinery and is used for inventory forecasting of heavy equipment. Given the cost of the equipment being ordered for inventory and the importance of the project, the algorithm was validated through two processes.



The first step involved creating a prototype in *Microsoft Excel* [76] that performed the calculations using formulas in cells. Using samples from the overall data set, the fits and forecasts were created. Next the commercial package *Minitab* [77] created fits and forecasts using the same data, and the results were compared.

A single threaded application was built in C++ using the *Excel* code as a blueprint. The forecast process was created as a function that accepted actual data as an array and returned an array of forecasts. The prototype aided in debugging since intermediate results from the C++ code could be compared to the equivalent step in *Excel*. The final results from the single threaded code was verified by comparing with the *Minitab* and Excel results.

The migration from the single threaded CPU code to a GPU-based application utilized the same validated code which eliminated potential errors from modifications. Once the GPU code was implemented, samples of the final result were processed by the single threaded code and validated.

The procedure of randomly sampling results from the GPU application and validating them with the single-threaded code, *Microsoft Excel*, and *Minitab* was repeated for six months when new data sets were released.

## 5.5 Results

The two dimensional GPU thread launch returns a three dimensional result set as shown in Fig. 5.23. As in Fig. 5.22, the rows represent products and aggregated categories, and the columns represent the period being forecast. Two complete cycles of data (24 points) are required for the exponential smoothing algorithm which reduces the number of columns by 24 when compared to the source data.

The slices of the cube are the number of months in the future of the forecast. The first slice stores the forecast for one month in advance from the period represented by

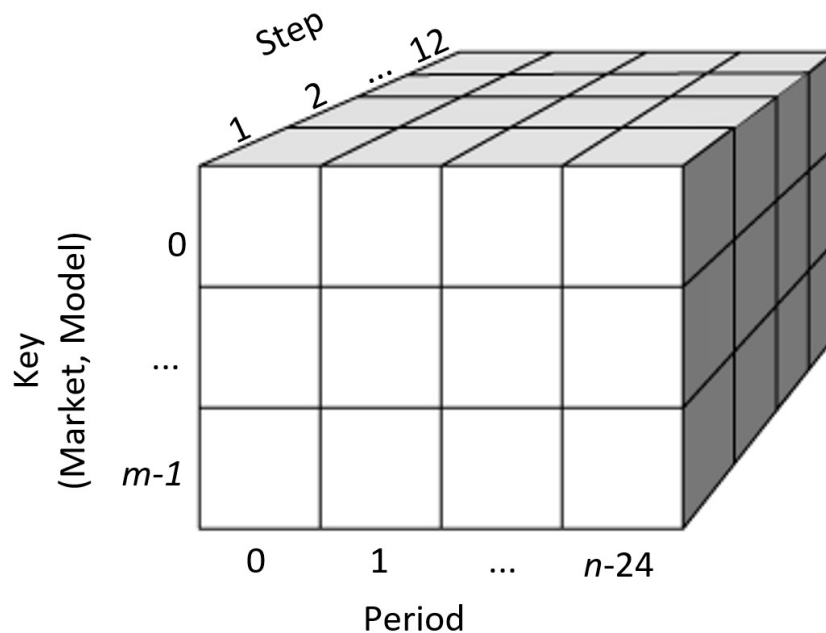


Figure 5.23: Data structure for forecast results. The rows are products, the columns are the last period of the  $2N$  cycles used to create the forecast, and the slices are the number of periods forecast in advance.

the column. For example, if column 0 was December 2002, then the first slice would be the forecast for January 2003.

The forecast results are imported into Microsoft Power BI [78], a data visualization and exploration tool, for analysis and reporting. Fig. 5.24 shows the dashboard created to analyze forecasts by specific products and product hierarchies. The filters in the top row allow the analyst to select the products of interest. There are two date filters on the left of the second row to select the forecast creation date and the time period desired. The chart at the center of the second row displays the actual product deliveries and the average forecast for that period to give a visual indication of the fit. The right chart shows the MAPE by lead time for the forecasts in the select time frame. The first column chart of the third row plots the median MAPE by lead time, while the second column chart retains the sign of the forecast error. This second chart

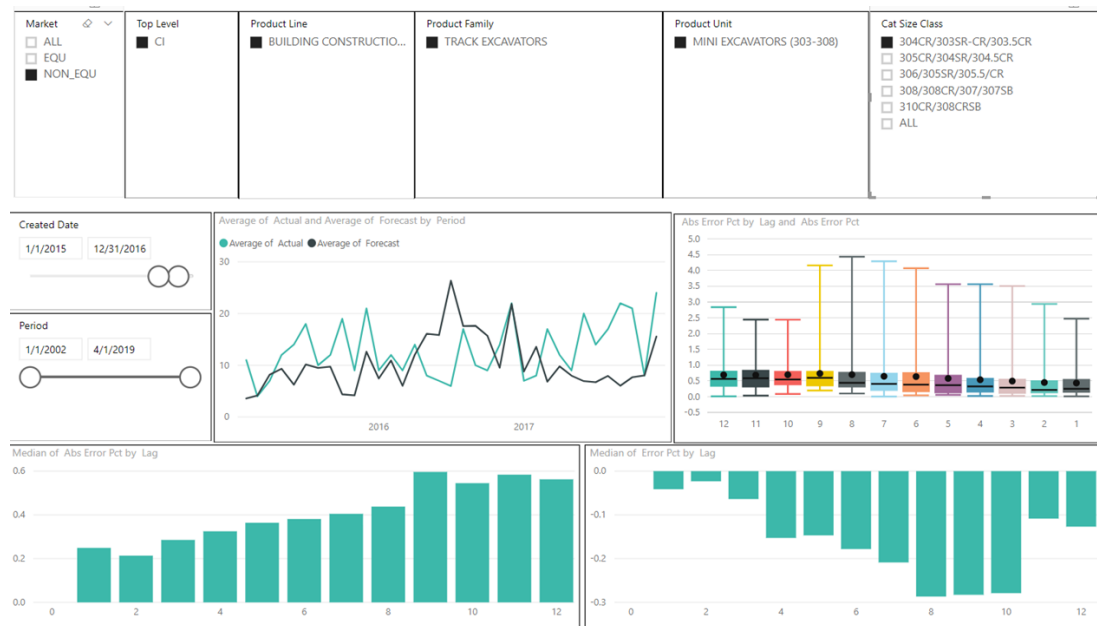


Figure 5.24: The forecast analysis dashboard created with Microsoft Power BI for analyzing product forecasts

is of particular interest as it highlights if the forecasts have a tendency to be biased below or above actual values.

The second dashboard shown in Fig. 5.25 is used to analyze how the forecast evolved of a single time period. The top row contains the product filters seen in the first dashboard. However, the single date filter on the second rows allows only for the selection of one period. The column chart on the second row displays the actual deliveries along with the forecast for each lead time. The third row plots the MAPE for each lead time. This dashboard is of interest for diagnosing forecasts that have an unusual MAPE.

The aggregated forecast results for all time periods shown in Fig. 5.2 are displayed in Fig. 5.26. The chart includes both product-level forecasts as well as aggregated product categories and reports the error during contraction and expansion periods.

Table 5.7 highlights the significant improvement in performance of the multi-GPU code over the CPU version. The single threaded CPU implementation was able to

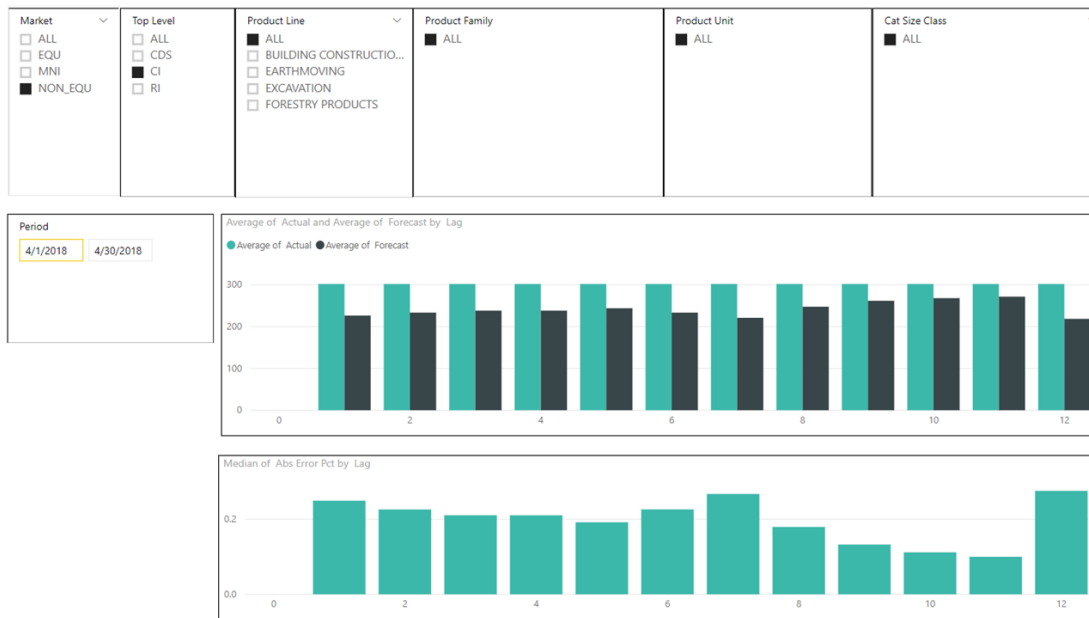


Figure 5.25: The forecast analysis dashboard created with Microsoft Power BI for analyzing accuracy for specific time periods

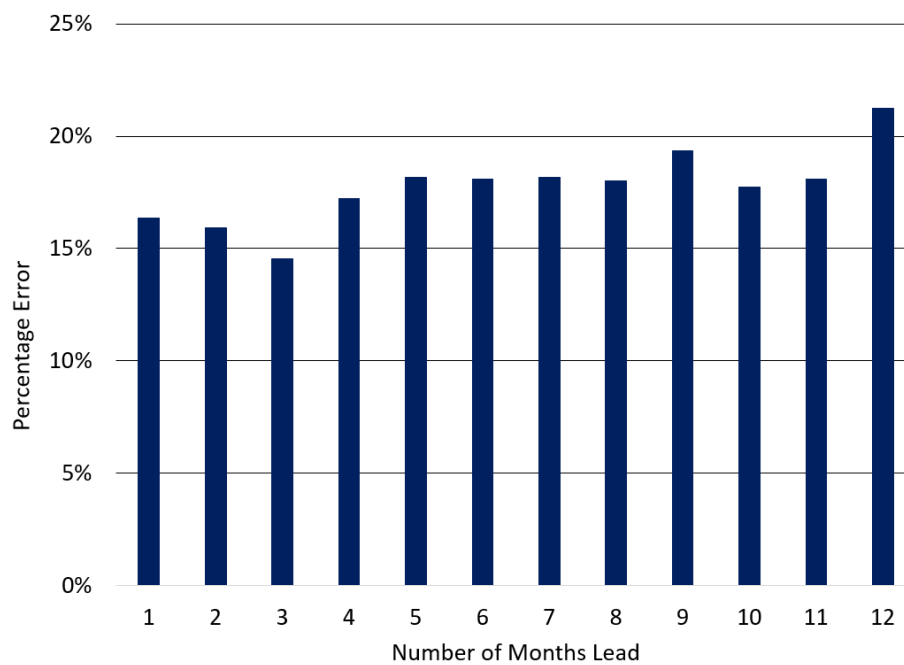


Figure 5.26: MAPE for forecasts based on data from January 2000 through June 2018 for product level and aggregated categories.

produce  $10^6$  forecasts per second. Even with the overhead of data movement among nodes and GPUs, the multi-GPU implementation generated  $10^8$  forecasts per second.

Table 5.7: Performance results for double precision forecasting

<b>Platform</b>	<b>Seconds</b>	<b>Forecasts</b>	<b>Forecasts / Sec</b>
Single threaded CPU	86.2844	84 082 803	974 484
multiGPU (14 NVIDIA Tesla K40s)	0.7864	84 082 803	106 921 164

## 5.6 Conclusion

Forecasting is a crucial process for manufacturing and retail organizations. Companies must invest in inventory to meet customer demand or risk losing those customers to competitors. Much research is performed in developing models to produce accurate forecasts in an acceptable time frame with available computational resources.

The original single exponential smoothing model was developed in 1957 by Robert Brown [69]. Since then, the algorithm has been extended to incorporate trends and seasonality to improve its accuracy and the variety of applications to which it can be applied. It has the dual advantages of producing accurate forecasts and ease of implementation which makes it a popular algorithm in the inventory management field [72], [75].

With the exponential rate of growth in data, high performance forecasting implementations will be required. The Internet of Things (IoT) is producing data from sensors in realtime which gives decision makers an option to change policies and procedures at a faster rate. However, this data must be processed in realtime as well, and insights must be provided to users of the information. The parallelism presented in this research offers a high performance solution that addresses these concerns. NVIDIA offers a low-cost GPU platform that can perform these forecasts at the edge of the

network where the data creation occurs to provide insights to both human users and machine learning applications.

The high performance algorithm developed in this research was validated using a dual step approach. The algorithm was first implemented manually using *Microsoft Excel* [76] and compared to results from the third-party commercial software package *Minitab* [77]. Next a single threaded CPU version was created in C++, and the results were validated with the *Excel* and *Minitab* output. This C++ function was then implemented in the GPU application.

While forecasting is a fundamental process in manufacturing and retail, it has applications in other fields such as energy production [79] and oil prices [80]. It can be utilized in any process in which future values must be predicted to make decisions based on changes to the process's parameters.

## CHAPTER VI: CONCLUSION

### 6.1 Introduction

Stochastic simulations are a challenging problem and are an active field of research. As the system of SDEs grows, the complexity increases. Variance is the controlling factor since a higher variance demands more samples. By taking smaller discretization steps in time and space, the variance can be reduced, but the trade off is more steps. Numerical methods and high-performance computing can provide additional tools to reduce the computational time of simulations.

Traditional Monte Carlo simulations take uniform steps over the dimensions. This approach is simple to implement and easily parallelized. In fact, MC simulations are categorized as “embarrassingly parallel” due to their inherent parallel nature. They can be implemented in high-performance environments such as clusters or GPUs to gain a speedup as communication is required at the beginning of the simulation to distribute parameters and at the end to aggregate the results. Each simulation is independent of past results and future results.

Originally designed to render graphics which are stored in memory as matrices, GPUs are now utilized in scientific computing. GPUs are optimized to perform a single operation across multiple data elements (SIMD) in parallel. A well-constructed simulation that avoids or minimizes branching is an ideal candidate for GPUs. Many of the TOP 500 supercomputers include GPUs on their cluster nodes as the computational

density (number of processors in a volume) and lower power consumption (compared to a cluster node) make them appealing.

High-performance computing can improve the computational time of simulations by distributing the work over a greater number of processors. However, modifications of the existing numerical methods can also reduce processing. Monte Carlo simulations are one method of solving stochastic differential equations by taking small steps through time or space while generating random values from a normal distribution for the Brownian motion [2]. The number of simulations is controlled by the variance in the estimation; the higher the variance, the more samples needed. Smaller step sizes can improve variance, but at the cost of more steps. Antithetic multilevel Monte Carlo leverages the variance reducing nature of MLMC and classical antithetic methods [43]. Use of the Milstein discretization rather than Euler-Maruyama further reduces variance.

The field of finance is one of the largest for the use of stochastic simulations. Pricing financial assets is a difficult problem, and as a portfolio grows in assets, the problem grows exponentially. The early Black-Scholes models involved a single equation for the price of an asset and used a constant volatility and interest rate [5]. The Heston model introduced a stochastic model for volatility [51]. This dissertation addressed the challenging problem of stochastic interest rates; the results having been published in [4]. The simulation employed a cluster of seven nodes, each equipped with dual NVIDIA Tesla K40 GPUs. The K40 has 2880 cores and 12 GB of memory, providing the simulation with 40320 CUDA cores.

GPU computing can be applied to other problems in finance. Inventory management is one such problem of concern to companies that must stock products to meet customer demand. The heavy equipment industry is particularly interesting due to the nature of the products. Each unit of equipment can range in price from under



\$100K to over \$1 million. In addition, the number of units sold in each month can range from 100 for smaller machines used on construction sites to only one in a year for larger mining machines. This dissertation utilized high-performance computing to significantly improve forecasting times for two advantages. First, new algorithms can be tested on historical data quickly to determine their capability to detect market shifts such expansion and contraction. Second, it provides the capability for realtime forecasts for data from IoT devices such as sensors in equipment.

Simulations are powerful tool for estimating systems with no analytical solutions. The complexity of the model and variance of the results controls how many samples are necessary and thus the length of time needed to complete. Numerical and computational approaches can reduce the compute time. High-performance environments distribute the workload across multiple processors, and GPUs are a commonly used tool in these systems. They can provide thousands of cores in a fraction of space used by servers with less power consumption. Numerical methods such as antithetic multilevel Monte Carlo and Milstein can further improve performance by variance reduction.

## 6.2 Significant Contribution

This dissertation has provided contributions to the financial field by blending computational and numerical methods to improve the simulation of stochastic systems. In addition, high-performance approaches were applied to the area of forecasting.

The difficulty of simulating complex stochastic simulations was discussed, and a novel approach of combining antithetic multilevel Monte Carlo using Milstein discretization [43] with a cluster of GPU-equipped nodes was presented. This provides a framework for simulating larger and more difficult problems faster by reducing variance with numerical algorithms with HPC resources [4].

The GPU framework was modified to address the separate but equally challenging real world problem of forecasting demand in the heavy equipment industry. The solution presented allows new forecasting algorithms to be tested rapidly on historical data by performing forecasts and comparing the accuracy to known values. *Backtesting* is a common approach when testing new algorithms, especially in the financial field [81].

## 6.3 Future Work

The research presented provides opportunities for future work in a number of areas. The improvements to stochastic simulations can be applied to larger systems. This is subject of an upcoming paper submission to *ACM Transactions on Modeling and Computer Simulation* and presentation submission to the *SC20* conference. This problem is extremely challenging in that it must simulate a system of 196608 SDEs and maintain a correlation matrix of  $65536 \times 65536$  (32 GB in storage). It could create new approaches for detecting cybersecurity attacks by analyzing aggregated network activity and identifying traffic that is abnormal.

The work in forecasting can be further improved by the use of deep learning to aid in detecting shifts in trends, rare events, and anomalous data. It could also be applied to areas needing real time forecasting such as sensor data from IoT devices in order to detect potential failures before they occur.

## REFERENCES

- [1] X. Mao, *Stochastic Differential Equations and Applications*, 2nd ed. Chichester: Horwood Publishing Ltd., 2008.
- [2] P. E. Kloeden, E. Platen, and H. Schurz, *Numerical solution of SDE through computer experiments*, 3rd ed. Germany: Springer, 2003.
- [3] G. J. Lord, C. E. Powell, and T. Shardlow, *Introduction to Computational Stochastic PDEs*, 1st ed., ser. Cambridge Texts in Applied Mathematics. New York, U.S.A.: Cambridge University Press, 2014, no. 50.
- [4] H. Lay, Z. Colgin, V. Reshniak, and A. Khaliq, “On the implementation of multilevel Monte Carlo simulation of the stochastic volatility and interest rate model using multi-GPU clusters,” *Monte Carlo Methods and Applications*, vol. 24, no. 4, 2018.
- [5] F. Black and M. Scholes, “The pricing of options and corporate liabilities,” *The Journal of Political Economy*, vol. 81, no. 3, pp. 637–654, 1973.
- [6] J. Sirignano and K. Spiliopoulos, “DGM: A deep learning algorithm for solving partial differential equations,” *Journal of Computational Physics*, vol. 375, pp. 1339–1364, 2018.

- [7] E. B. Iversen, J. M. Morales, J. K. Moller, and H. Madsen, “Short-term probabilistic forecasting of wind speed using stochastic differential equations,” *International Journal of Forecasting*, vol. 32, no. 3, pp. 981–990, Jul 1, 2016.
- [8] A. Baldwin, I. Gheysa, C. Ioannidis, D. J. Pym, and J. Williams, “Contagion in cybersecurity attacks.” in *WEIS*, 2012.
- [9] M. Tenenbaum and H. Pollard, *Ordinary Differential Equations*. Dover Publications, Oct. 1985.
- [10] S. Zhang, X. Meng, T. Feng, and T. Zhang, “Dynamics analysis and numerical simulations of a stochastic non-autonomous predator–prey system with impulsive effects,” *Nonlinear Analysis: Hybrid Systems*, vol. 26, pp. 19–37, 2017.
- [11] S. Ditlevsen and A. Samson, “Introduction to stochastic models in biology,” 2010. [Online]. Available: <https://hal.archives-ouvertes.fr/hal-00534445>
- [12] R. Srivastava, L. You, J. Summers, and J. Yin, “Stochastic vs. deterministic modeling of intracellular viral kinetics,” *Journal of Theoretical Biology*, vol. 218, no. 3, pp. 309–321, 2002.
- [13] D. Schnoerr, G. Sanguinetti, and R. Grima, “The complex chemical Langevin equation,” *The Journal of Chemical Physics*, vol. 141, no. 2, p. 024103, 2014.
- [14] E. J. Allen and C. Huff, “Derivation of stochastic differential equations for sunspot activity,” *A&A*, vol. 516, 2010.
- [15] Y. Aït-Sahalia, J. Cacho-Diaz, and R. J. A. Laeven, “Modeling financial contagion using mutually exciting jump processes,” *Journal of Financial Economics*, vol. 117, no. 3, pp. 585–606, 2015.

- [16] A. Baldwin, I. Gheyas, C. Ioannidis, D. Pym, and J. Williams, “Contagion in cyber security attacks,” *Journal of the Operational Research Society*, vol. 68, no. 7, pp. 780–791, 2017.
- [17] E. B. Iversen, J. M. Morales, J. K. Møller, P. Trombe, and H. Madsen, “Leveraging stochastic differential equations for probabilistic forecasting of wind power using a dynamic power curve,” *Wind Energy*, vol. 20, no. 1, pp. 33–44, Jan 2017.
- [18] T. V. Ta and L. T. H. Nguyen, “A stochastic differential equation model for the foraging behavior of fish schools,” *Physical Biology*, vol. 15, no. 3, 2018.
- [19] T. Dudok de Wit, L. Lefèvre, and F. Clette, “Uncertainties in the sunspot numbers: Estimation and implications,” *Solar Physics*, vol. 291, no. 9, pp. 2709–2731, Nov 2016.
- [20] G. Maruyama, “Continuous markov processes and stochastic equations,” *Rendiconti del Circolo Matematico di Palermo*, vol. 4, no. 1, pp. 48–90, 1955.
- [21] Z. Colgin, “Computational improvements for stochastic simulation with multilevel Monte Carlo,” Ph.D. dissertation, Middle Tennessee State University, 2016.
- [22] (2019) Top500 supercomputer sites. [Online]. Available: <https://www.top500.org/>
- [23] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, 2nd ed. Pearson Education Limited, 2003.
- [24] Intel Xeon processors. [Online]. Available: <https://www.intel.com/content/www/us/en/products/processors/xeon/view-all.html>

- [25] AMD EPYC 7000 series. [Online]. Available:  
<https://www.amd.com/en/products/epyc-7000-series>
  
- [26] B. Barney. Posix threads programming. Lawrence Livermore National Laboratory. [Online]. Available: <https://computing.llnl.gov/tutorials/pthreads/>
  
- [27] OpenMP Specifications. OpenMP Architecture Review Board. [Online]. Available: <https://www.openmp.org/specifications/>
  
- [28] G. Bosilca, J. Squyres, E. Gabriel, and J. Ladd, “Open MPI State of the Union XI Community Meeting SC18.” SC’18, November 13, 2018, Dallas, TX, USA, 2018.
  
- [29] Intel® Core™ i9-9980XE Extreme Edition Processor. [Online]. Available:  
<https://ark.intel.com/content/www/us/en/ark/products/189126/intel-core-i9-9980xe-extreme-edition-processor-24-75m-cache-up-to-4-50-ghz.html>
  
- [30] “CUDA Toolkit Archive,” Mar 27, 2018. [Online]. Available:  
<https://developer.nvidia.com/cuda-toolkit-archive>
  
- [31] S. P. Jammy, C. T. Jacobs, D. J. Lusher, and N. D. Sandham, “Energy efficiency of finite difference algorithms on multicore CPUs, GPUs, and Intel Xeon Phi processors,” 27 Sep 2017. [Online]. Available:  
<https://arxiv.org/abs/1709.09713>
  
- [32] L. Tassev. Report: Crypto miners bought 3 million GPUs last year. Saint Bitts LLC. [Online]. Available: <https://news.bitcoin.com/report-crypto-miners-bought-3-million-gpus-last-year/>

- [33] ——. AMD increases GPU production to match crypto mining demand. Saint Bitts LLC. [Online]. Available: <https://news.bitcoin.com/amd-increases-gpu-production-to-match-crypto-mining-demand/>
- [34] CUDA Toolkit v10.1.243 Best Practices Guide. NVIDIA. [Online]. Available: <https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html>
- [35] J. Peddie and R. Dow, “Add-in board report (quarterly),” 2019.
- [36] Khronos group. [Online]. Available: <https://www.khronos.org/>
- [37] OpenCL overview. Khronos Group. [Online]. Available: <https://www.khronos.org/opencl/>
- [38] M. R. Norman, A. Mametjanov, and M. Taylor, “Exascale programming approaches for the accelerated model for climate and energy,” *Exascale scientific applications: Scalability and performance portability*, 2017.
- [39] D. T. Gillespie, “The chemical Langevin equation,” *The Journal of Chemical Physics*, vol. 113, no. 1, pp. 297–306, 2000.
- [40] P. Kloeden and A. Neuenkirch, “Convergence of numerical methods for stochastic differential equations in mathematical finance,” Apr 30, 2012.
- [41] M. Liu, “Optimal number of trials for Monte Carlo simulation,” *VRC–Valuation Research Report*, 2017.
- [42] M. B. Giles, “Multilevel Monte Carlo path simulation,” *Operations Research*, vol. 56, no. 3, pp. 607–617, 2008.
- [43] M. B. Giles and L. Szpruch, “Antithetic multilevel Monte Carlo estimation for multi-dimensional SDEs without Lévy area simulation,” *The Annals of Applied Probability*, vol. 24, no. 4, pp. 1585–1620, Aug 1, 2014.

- [44] L. Ricketson, “Three improvements to multi-level Monte Carlo simulation of SDE systems,” *arXiv preprint arXiv:1309.1922*, 2013.
- [45] H. Niederreiter, “Quasi-Monte Carlo methods in computational finance.” *COSMOS*, vol. 1, no. 1, pp. 113 – 125, 2005.
- [46] M. B. Giles and B. J. Waterhouse, “Multilevel quasi-Monte Carlo path simulation,” *Advanced Financial Modelling, Radon Series on Computational and Applied Mathematics*, vol. 8, pp. 165–181, 2009.
- [47] D. van Ravenzwaaij, P. Cassey, and S. D. Brown, “A simple introduction to Markov Chain Monte-Carlo sampling,” *Psychonomic Bulletin & Review*, vol. 25, no. 1, pp. 143–154, Feb 2018.
- [48] D. F. Anderson and D. J. Higham, “Multilevel Monte Carlo for continuous time Markov chains, with applications in biochemical kinetics,” *Multiscale Modeling & Simulation*, vol. 10, no. 1, pp. 146–179, 2012.
- [49] C. J. Geyer, “Practical Markov Chain Monte Carlo,” *Statistical Science*, vol. 7, no. 4, pp. 473–483, 1992.
- [50] V. Reshniak, “Reducing computational cost of the multilevel monte carlo method by construction of suitable pathwise integrators,” Ph.D. dissertation, Middle Tennessee State University, 2017.
- [51] S. L. Heston, “A closed-form solution for options with stochastic volatility with applications to bond and currency options,” *The Review of Financial Studies*, vol. 6, no. 2, pp. 327–343, April 1993.
- [52] A. Medvedev and O. Scaillet, “Pricing american options under stochastic volatility and stochastic interest rates,” *Journal of Financial Economics*, vol. 98, no. 1, pp. 145–159, October 2010.



- [53] D. E. Knuth, *The Art of Computer Programming, Volume 2 Seminumerical Algorithms*, 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.
- [54] B. P. Welford, "Note on a method for calculating corrected sums of squares and products," *Technometrics*, vol. 4, no. 3, pp. 419–420, 1962.
- [55] "Penguin computing on-demand." [Online]. Available: <https://www.penguincomputing.com/pod-hpc-cloud/>
- [56] M. Zwolenski and L. Weatherill, "The Digital Universe Rich Data and the Increasing Value of the Internet of Things," *Australian Journal of Telecommunications and the Digital Economy*, vol. 2, no. 3, pp. 1–47, 2014.
- [57] J. Nandimath, E. Banerjee, A. Patil, P. Kakade, S. Vaidya, and D. Chaturvedi, "Big data analysis using Apache Hadoop," in *2013 IEEE 14th International Conference on Information Reuse Integration (IRI)*, 8 2013, pp. 700–703.
- [58] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache Spark: A Unified Engine for Big Data Processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, 10 2016.
- [59] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 1 2008.
- [60] U. Sivarajah, M. M. Kamal, Z. Irani, and V. Weerakkody, "Critical analysis of Big Data challenges and analytical methods," *Journal of Business Research*, vol. 70, pp. 263–286, 2017.
- [61] M. Muller, *Essentials of inventory management*. HarperCollins Leadership, 2019.

- [62] S. Clarke and others, “The Little (Illustrated) Book of Operational Forecasting,” *Foresight: The International Journal of Applied Forecasting*, no. 52, 2019.
- [63] P. A. W. Lewis and J. G. Stevens, “Nonlinear modeling of time series using multivariate adaptive regression splines (MARS),” *Journal of the American Statistical Association*, vol. 86, no. 416, pp. 864–877, 1991.
- [64] A. C. Harvey and A. C. Harvey, *Time series models*. Harvester Wheatsheaf London, 1993, vol. 2.
- [65] A. S. Weigend, *Time series prediction: forecasting the future and understanding the past*. Routledge, 2018.
- [66] R. S. Tsay and R. Chen, *Nonlinear time series analysis*. Wiley, 2018.
- [67] D. H. Wolpert, W. G. Macready, and others, “No free lunch theorems for optimization,” *IEEE transactions on evolutionary computation*, vol. 1, no. 1, pp. 67–82, 1997.
- [68] G. E. P. Box and N. R. Draper, *Empirical model-building and response surfaces*. John Wiley & Sons, 1987.
- [69] R. G. Brown, “Exponential smoothing for predicting demand,” in *Operations Research*, vol. 5, no. 1. Institute for Operations Research and the Management Sciences, 1957, p. 145.
- [70] L. S. Lasdon, A. D. Waren, A. Jain, and M. Ratner, “Design and Testing of a Generalized Reduced Gradient Code for Nonlinear Programming,” *ACM Trans. Math. Softw.*, vol. 4, no. 1, pp. 34–50, 3 1978.

- [71] R. J. Hyndman and others, “Another look at forecast-accuracy metrics for intermittent demand,” *Foresight: The International Journal of Applied Forecasting*, vol. 4, no. 4, pp. 43–46, 2006.
- [72] C. C. Holt, “Forecasting seasonals and trends by exponentially weighted moving averages,” *International Journal of Forecasting*, vol. 20, no. 1, pp. 5–10, 2004.
- [73] P. Goodwin and others, “The Holt-Winters approach to exponential smoothing: 50 years old and going strong,” *Foresight*, vol. 19, pp. 30–33, 2010.
- [74] R. J. Hyndman and G. Athanasopoulos, *Forecasting: principles and practice*. OTexts, 2018.
- [75] P. R. Winters, “Forecasting sales by exponentially weighted moving averages,” *Management science*, vol. 6, no. 3, pp. 324–342, 1960.
- [76] Microsoft, “Excel for office 365.” [Online]. Available: <https://products.office.com/en-us/excel>
- [77] Minitab LLC, “Minitab.” [Online]. Available: <http://www.minitab.com/>
- [78] Microsoft, “Powerbi.” [Online]. Available: <https://powerbi.microsoft.com/>
- [79] J. Bedi and D. Toshniwal, “Deep learning framework to forecast electricity demand,” *Applied energy*, vol. 238, pp. 1312–1326, 2019.
- [80] A. Garratt, S. P. Vahey, and Y. Zhang, “Real-time forecast combinations for the oil price,” *Journal of Applied Econometrics*, vol. 34, no. 3, pp. 456–462, 2019.
- [81] C. Fry and M. Brundage, “The M4 forecasting competition - A practitioner’s view,” *International Journal of Forecasting*, vol. July, p. 5, 2019, none.