Adaptive Sampling Line Search for Simulation Optimization

Prasanna Kumar Ragavan

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Industrial and Systems Engineering

Michael R. Taaffe, Co-Chair
Raghu Pasupathy, Co-Chair
Douglas R. Bish
Christian Wernz

February 17, 2017
Blacksburg, Virginia

Keywords: Simulation Optimization, Adaptive Sampling, Line Search, ADALINE
Copyright 2017, Prasanna Kumar Ragavan
Adaptive Sampling Line Search for Simulation Optimization

Prasanna Kumar Ragavan

(ABSTRACT)

This thesis is concerned with the development of algorithms for simulation optimization (SO), a special case of stochastic optimization where the objective function can only be evaluated through noisy observations from a simulation. Deterministic techniques, when directly applied to simulation optimization problems fail to converge due to their inability to handle randomness thus requiring sophisticated algorithms. However, many existing algorithms dedicated for simulation optimization often show poor performance on implementation as they require extensive parameter tuning.

To overcome these shortfalls with existing SO algorithms, we develop ADALINE, a line search based algorithm that minimizes the need for user defined parameter. ADALINE is designed to identify a local minimum on continuous and integer ordered feasible sets. ADALINE on continuous feasible sets mimics deterministic line search algorithms, while it iterates between a line search and an enumeration procedure on integer ordered feasible sets in its quest to identify a local minimum. ADALINE improves upon many of the existing SO algorithms by determining the sample size adaptively as a trade-off between the error due to estimation and the optimization error, that is, the algorithm expends simulation effort proportional to the quality of the incumbent solution. We also show that ADALINE converges “almost surely” to the set of local minima on integer ordered feasible sets at an exponentially fast rate. Finally, our numerical results suggest that ADALINE converges to a local minimum faster than the best available SO algorithm for the purpose.

To demonstrate the performance of our algorithm on a practical problem, we apply ADALINE in solving a surgery rescheduling problem. In the rescheduling problem, the objective is to minimize the cost of disruptions to an existing schedule shared between multiple surgical specialties while accommodating semi-urgent surgeries that require expedited intervention. The disruptions to the schedule are determined using a threshold based heuristic and ADALINE identifies the best threshold levels for various surgical specialties that minimizes the expected total cost of disruption. A comparison of the solutions obtained using a Sample Average Approximation (SAA) approach, and ADALINE is provided. We find that the adaptive sampling strategy in ADALINE identifies a better solution more quickly than SAA.
Adaptive Sampling Line Search for Simulation Optimization

Prasanna Kumar Ragavan

(GENERAL AUDIENCE ABSTRACT)

This thesis is concerned with the development of algorithms for simulation optimization (SO), where the objective function does not have an analytical form, and can only be estimated through noisy observations from a simulation. Deterministic techniques, when directly applied to simulation optimization problems fail to converge due to their inability to handle randomness thus requiring sophisticated algorithms. However, many existing algorithms dedicated for simulation optimization often show poor performance on implementation as they require extensive parameter tuning.

To overcome these shortfalls with existing SO algorithms, we develop ADALINE, a line search based algorithm that minimizes the need for user defined parameter. ADALINE is designed to identify a local minimum on continuous and integer ordered feasible sets. ADALINE on continuous feasible sets mimics deterministic line search algorithms, while it iterates between a line search and an enumeration procedure on integer ordered feasible sets in its quest to identify a local minimum. ADALINE improves upon many of the existing SO algorithms by determining the sample size adaptively as a trade-off between the error due to estimation and the optimization error, that is, the algorithm expends simulation effort proportional to the quality of the incumbent solution. Finally, our numerical results suggest that ADALINE converges to a local minimum faster than the best available SO algorithm for the purpose.

To demonstrate the performance of our algorithm on a practical problem, we apply ADALINE in solving a surgery rescheduling problem. In the rescheduling problem, the objective is to minimize the cost of disruptions to an existing schedule shared between multiple surgical specialties while accommodating semi-urgent surgeries that require expedited intervention. The disruptions to the schedule are determined using a threshold based heuristic and ADALINE identifies the best threshold levels for various surgical specialties that minimizes the expected total cost of disruption. A comparison of the solutions obtained using traditional optimization techniques, and ADALINE is provided. We find that the adaptive sampling strategy in ADALINE identifies a better solution more quickly than traditional optimization.
To my family, and all my teachers...
Acknowledgments

I would like to express my heartfelt gratitude to Dr. Raghu Pasupathy, and Dr. Michael Taaffe, for, this thesis would have been a dream had it not been for their intervention, and persistent support. I am greatly indebted to Dr. Pasupathy for all his time, and patience in molding me a researcher, and a good human being. All the courses taught by Dr. Pasupathy were instrumental in me pursuing further research in this field. I will always look up to him. I am greatly indebted to Dr. Taaffe for motivating me, for his guidance in writing and for his efforts and care in making sure I completed my PhD.

I would like to sincerely thank Dr. Doug Bish for being a part of my committee and for his valuable insights and encouragement throughout the course of my graduate program. I would like to thank Dr. Christian Wernz for being a part of my committee and for providing inputs on healthcare operations research. I am also thankful to Dr. Ebru Bish for her support during the initial stages of my graduate study, and for evoking interest in me to solve problems in healthcare. I am thankful to Dr. Don Taylor, Dr. Maury Nussbaum, and Dr. Jamie Camelio for providing teaching assistantships in the Grado Department of Industrial and Systems Engineering (ISE). As a teaching assistant at ISE, I enjoyed working with Dr. Laurel Travis and Dr. Natalie Cherbaka.

I am thankful to my friends Krishnan Gopalan, Niloy Mukherjee for their support in personal and professional development through my stay in Blacksburg. Thanks to Mahesh Narayanamurtthy for all his help in the final stages, and for never letting me stranded while in Blacksburg. I also thank Kalyani Nagaraj for helping me during the initial phases of my research. I would also like to take this opportunity to thank Vamshi for his valuable insights in clinical scheduling.

I would also like to thank my cousins Raghavendran and Priya for being there for me through the thick and thin. My brother Guruprasadhi has been a great friend and source of support in times of need. I am blessed to have such a wonderful brother for guidance and advice. Thanks to Jayashree for standing besides me all the time, and for her unconditional love and support throughout. Finally, my sincerest thanks to my parents, Sujatha and Ragavan for all their struggles, and sacrifices in helping me receive a good education.
Contents

1 Introduction 1

1.1 Motivation ................................................................. 2

1.1.1 Example: Continuous Simulation Optimization .......... 2

1.1.2 Example: Integer Simulation Optimization ............. 3

1.2 Simulation Optimization .................................................... 3

1.2.1 Continuous Simulation Optimization ......................... 4

1.2.2 Integer Ordered Simulation Optimization ............... 6

1.3 Organization of the thesis .............................................. 7

2 ADALINE for Integer-Ordered Simulation Optimization 9

2.1 Competitors ................................................................. 10

2.2 Algorithm ADALINE - Integer Ordered ......................... 11

2.2.1 Adaptive Linear Interpolation and Line Search ........... 13

2.2.2 Neighborhood Enumeration (NE) ............................ 15

2.3 Local Convergence ......................................................... 21

2.4 Numerical Results .......................................................... 23

2.4.1 Inventory Optimization with ADALINE ....................... 23

2.4.2 Bus Scheduling Problem .............................................. 24
4.5.2 Threshold Optimization using Integer programming .................................. 66

4.6 Conclusion ........................................................................................................ 67

5 Concluding Remarks ......................................................................................... 70

Bibliography ........................................................................................................ 72

Appendix A ADALINE for Continuous Simulation Optimization ......................... 78

Appendix B ADALINE for Integer Simulation Optimization ............................... 90

Appendix C Surgery Scheduling Problem ........................................................... 103
List of Figures

2.1 Trajectory of ADALINE ............................................. 12

2.2 The figure shows the progression of ADALINE on the \((s, S)\) inventory policy optimization. The solutions visited through iterations, and the cumulative simulation budget utilized are plotted. Earlier iterations prioritize exploration at a smaller sample size, followed by exploitation with a significant increase in sampling. ................................................................. 25

2.3 The figure shows percentiles of the expected wait time corresponding to the solution returned by ADALINE and R-SPLINE at the end of \(b\) simulation calls on the nine-bus scheduling problem................................................................. 26

2.4 The figure shows percentiles of the expected wait time corresponding to the solution returned by ADALINE and R-SPLINE at the end of \(b\) simulation calls on the nineteen-bus scheduling problem................................................................. 27

2.5 The figure shows the percentiles of the expected value corresponding to the solution returned by ADALINE at the end of \(b\) simulation calls on the multi-dimensional quadratic problem. ................................................................. 28

3.1 The figure shows percentiles of the norm of the true gradient corresponding to the solutions returned by all variations of our algorithm on the Aluffi-Pentini problem................................................................. 37

3.2 Performance of the adaptive samplings strategy on the Aluffi-Pentini problem in comparison with the sampling strategy in Krejić and Krklec (2013)................................................................. 38

3.3 Sample growth in the Aluffi-Pentini problem for the Stochastic BFGS method................................................................. 39

3.4 The figure shows percentiles of the norm of the true gradient corresponding to the solutions returned by all variations of our algorithm on the Rosenbrock problem................................................................. 40
3.5 Performance of the adaptive samplings strategy on the Rosenbrock problem in comparison with the sampling strategy in Krejić and Krklec (2013) ........................................ 40

3.6 Sample growth in the Rosenbrock problem for the Stochastic BFGS method .................. 41

4.1 Realization of Random Variables that represent the requests for Semi-urgent surgeries ........ 47

4.2 Sequence of Decisions for the Surgical Rescheduling Problem ................................. 54

4.3 Figure shows the expected total cost of rescheduling as a function of the total simulation calls for the solutions identified by ADALINE at specified budget levels ............................ 65

4.4 Figure shows the expected total cost of rescheduling as a function of elapsed time for the solutions identified by ADALINE at specified time limits ............................................. 66

4.5 Figure shows the expected total cost of rescheduling as a function of elapsed time for the solutions identified by the integer optimization framework with various sample sizes ................. 68

4.6 Figure shows a comparison of the quality of solutions obtained with thresholds determined through ADALINE and the Integer Optimization framework ............................. 68
List of Tables

4.1 Table shows the mean and standard deviation of the negative binomially distributed random variable indicating the number of semi-urgent cases requested on any particular day . . . . . . . . . . . 64

4.2 Table shows the parameters of the distribution of surgery duration . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64

4.3 Average optimality gap of the solutions to the mixed integer programming formulation obtained at various time limits . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
Chapter 1

Introduction

In this research, we develop algorithms for simulation optimization (SO)—a branch of optimization where the objective function, and possibly the constraints, do not have an analytical form and can only be observed through a stochastic simulation. Formally, the simulation optimization problem can be stated as follows:

\[ P_{SO} : \text{minimize} \quad g(x) \]
\[ \text{subject to} \quad x \in X \]

where, the feasible set \( X \subseteq \mathbb{R}^d \) is either continuous, or integer-ordered, \( X \subseteq \mathbb{Z}^d \).

In Problem \( P_{SO} \), the objective function, \( g : X \rightarrow \mathbb{R} \) is real-valued, and in the form of an expectation, such that, \( g(x) = \mathbb{E}_\xi \left[ G(x, \cdot) \right] \). Further, at any decision point, \( x \), only an estimate of the objective function can be obtained through random observations, \( G(x, \xi_j) \), from a simulation. We assume that the observations are independent and identically distributed (i.i.d.), and a point estimator of the objective function, \( \hat{g}_m(x) \), is constructed based on these outcomes. Since \( g(x) = \mathbb{E} \left[ G(x, \cdot) \right] \), the sample mean of the observations, \( \hat{g}_m(x) = m^{-1} \sum_{j=1}^{m} G(x, \xi_j) \), is a suitable estimator. It should be noted that, in SO, obtaining an estimate of the objective function entails either repeating the simulation multiple times, as in the case of terminating simulations or longer simulation times (steady state simulation).
1.1 Motivation

To motivate our research, we present an application of SO in operating room (OR) management. OR management involves three phases of decision making, namely (i) determination of total operating room capacity (strategic), (ii) allocation of capacity between various specialties (tactical), and (iii) scheduling individual surgeries to operating rooms (operational). Later, in Chapter 4, we study in detail the problem of allocating surgical capacity between non-elective and elective surgeries under a constrained operating room environment. However, for illustrative purposes, let us consider the problem of allocating capacity to individual specialties when the total available operating room capacity is unconstrained, a simplified version of the real problem.

1.1.1 Example: Continuous Simulation Optimization

Even when unlimited capacity is available, capacity allocation requires scheduling of OR resources which incurs a fixed cost of $c_r$ dollars per unit time. When sufficient capacity is not allocated to the specialty, resources have to be scheduled with overtime to meet the excess demand, costing $c_o$ dollars for every unit of overtime incurred. The surgical duration every week for any specialty is a random variable, $D$, with cdf $F_D(\cdot)$. Therefore, determining the optimal capacity, $x^*$ can be formulated as a news vendor problem (Hosseini and Taaffe, 2014) as follows:

$$g(x) = \min_x \mathbb{E}_D[G(x, D)] = \mathbb{E}_D[c_r x + (c_o \max\{D - x, 0\})]$$

It is well known that the optimal solution to the one dimensional news vendor problem is to staff for the $(\frac{c_o - c_r}{c_o})$th quantile of the demand distribution. For example, if regular staffing cost, $c_r = $1400/hr, the overtime cost, $c_o = $1600/hr and the surgical volume, $D$ is a random variable uniformly distributed between 160 and 600 hours, then the optimal solution is to plan for the 0.125th quantile which translates to 215 staffing hours with a total expected cost of $570,500 per week. If the solution were to replace the random variable $D$ with its expectation, the optimal solution is to staff for the expectation of the random variable, that is, $x$, is equal to 380. The total expected cost incurred by using such a solution is $620,000, an increase of approximately $50,000 per week from the optimal cost, thereby justifying the need for sophisticated techniques that deal with uncertainty.

In the above example, we assume that the distribution of the random quantity (surgical volume) is known which leads to an analytical solution. Under incomplete information on the demand, a closed form
solution may be impossible necessitating simulation/black-box based optimization techniques. In this case, under incomplete information on demand, the optimal solution can be estimated as the $\left(\frac{c_o - c_r}{c_o}\right)$th quantile by repeatedly drawing samples using a simulation.

A variation of this problem, where allocating labor hours between multiple specialties under a constrained environment (with a constraint on the total available capacity), however, requires a different solution technique. Staffing for the $\left(\frac{c_o - c_r}{c_o}\right)$th quantile for every specialty may no longer be feasible for the multi-dimensional news vendor problem, and when the demand distribution is unknown, simulation optimization methods that exploit the structure of $g(x)$ through random observations, $G(x, D)$ are required (Kim, 2006).

### 1.1.2 Example: Integer Simulation Optimization

For the multi-dimensional news vendor problem discussed in Section 1.1, when the feasible space is continuous, gradient information observed at any decision point can be directly utilized to guide a SO algorithm. However, when the feasible space is integer-ordered, gradient based techniques are not directly applicable. Consider the problem of accommodating surgeries in a hospital where surgical capacity is shared between non-elective and elective surgeries. Elective surgeries that are scheduled at least a week in advance may have to be postponed or canceled to accommodate non-elective surgeries which have an uncertain demand and are realized only a day or two prior to the surgery. A threshold based scheduling policy that exclusively reserves operating room capacity for non-elective surgeries can be devised to solve this problem, the details of which can be found in Chapter 4. However, optimizing the threshold levels, that take values in the integer-ordered space, when the surgical demand can be realized only through a simulation requires sophisticated SO techniques which are developed as a part of this dissertation.

### 1.2 Simulation Optimization

Simulation optimization (SO) is concerned with solving problems where the objective function, and (or) the constraints can only be evaluated through a simulation or as the response of a “black-box” given an input. Such problems find a variety of applications including but not limited to supply chain management (Iravani et al., 2003; Jung et al., 2004), operations and scheduling (Belyi et al., 2009; Buchholz and Thummler, 2005), revenue management (Gosavi et al., 2007; Horng and Yang, 2012), public health services (Eubank et al., 2004; Henderson and Mason, 2005) and hospital operations (Balasubramanian et al., 2007; Lamiri et al., 2009). For a comprehensive list of applications, along with the detailed problem description, please refer to Pasupathy and Henderson (2006) and http://www.simopt.org. In addition to classical SO problems,
the stochastic nature of large scale supervised machine learning models present themselves conducive for exploitation using SO algorithms (Byrd et al., 2012, 2014).

Several algorithms have been proposed to solve simulation optimization problems and a number of review articles have been published on this topic. Amaran et al. (2014) provides a comprehensive overview of the field starting with applications of simulation optimization, and a taxonomy of the algorithms. In SO, the algorithms can be classified based on the feasible space they operate on (continuous or discrete decision variables), cardinality of the feasible set (finite versus uncountable feasible sets), and the search mechanism (random search or gradient search or derivative free methods). Ranking and Selection procedures are designed to select the best solution out of a finite set of feasible solutions with a probabilistic guarantee on choosing the best solution. Conventional ranking and selection methods seek to determine the best solution in a sequential setting where inferior systems are dropped in each phase, followed by an increase in sample size for further evaluations of ‘promising’ systems (Kim, 2005; Kim and Nelson, 2007). In ranking and selection, even if the feasible set is finite, the requirement to completely enumerate the set of all feasible solutions renders them inefficient when evaluating large number of solutions (Fu et al., 2015). Algorithms that iteratively determine the next solution eliminate the need for enumerating the feasible region, and are applicable even when the feasible set is uncountable. In this section, we focus on such SO algorithms that determine the subsequent solutions to evaluate “on the fly” for continuous, and integer-ordered simulation optimization problems.

1.2.1 Continuous Simulation Optimization

Simulation optimization on continuous feasible sets involves visiting a sequence of solutions that are chosen iteratively from an uncountable number of candidates, often by using the estimated gradient at each iteration. One of the first algorithms for continuous simulation optimization is the Stochastic Approximation (SA) algorithm given by Robbins-Monro (Robbins and Monro, 1951). SA is an iterative method, where starting with an initial solution, \( x_0 \), the sequence of iterates, \( \{X_k\} \) are updated using the equation

\[
X_{k+1} := X_k + a_k \nabla G(X_k, \xi), \quad k \geq 0
\]

This method is the stochastic analogue to Newton type search methods in deterministic optimization, where \( \nabla G(X_k, \xi) \) denotes an observation of the ‘noisy’ gradient from a simulation, and \( a_k \) denotes the step length. Unlike deterministic optimization where \( a_k \) is determined at every iteration, a decreasing sequence of positive numbers approaching zero in the limit constitute the step length. For the algorithm to converge, in addition to \( \lim_{k \to \infty} a_k = 0 \), the following conditions should also hold: (i) \( \sum_{k=0}^{\infty} a_k = \infty \) and (ii) \( \sum_{k=0}^{\infty} a_k^2 < \infty \).

Although SA attains the “canonical” rate of convergence, that is, converges at the best possible rate
for a stochastic algorithm \( O(1/\sqrt{n}) \), it often performs poorly on implementation due to a pre-specified step length sequence. Polyak and Juditsky (1992) proposes a variation of the SA algorithm by averaging the iterates, and that takes a longer step at each iteration to obtain a more robust performance while not compromising on the rate of convergence. However, the optimal choice of step length is still unknown for a particular problem, and thus SA requires extensive parameter tuning to achieve better finite time performance. For a thorough treatment of various SA algorithms in solving unconstrained, and constrained SO problems along with a discussion of their theoretical convergence, and rates, please refer to Kushner and Clark (2012).

In order to overcome the issues associated with a pre-specified step length sequence in SA, deterministic line search algorithms can be adapted with appropriate sampling techniques to solve a sample path problem—an approximation of the objective function obtained by i.i.d realizations of the random output. This method, called “Sample Average Approximation” is appropriate for SO problems where the underlying structure of the objective function is known. Since, in SAA, an estimate of the objective function obtained using a fixed number of realizations is optimized, it is readily applicable for solving stochastic optimization problems irrespective of the feasible set, or the constraints. A simple choice on the approximation is to replace the expectation with its sample average obtained using a finite sample size, \( M \), and with a big-enough sample size, it can guarantee convergence. Some of the previous work on SAA, and its application in stochastic optimization along with the convergence theory includes Shapiro (1996), Shapiro and Wardi (1996), Homem-de Mello and Bayraksan (2014), and Kim et al. (2015).

One of the important components in line search based simulation optimization algorithms is the estimation of the gradient at any iterate. In order for SA to converge, it requires that the estimator for the gradient to be unbiased, that is, \( E[\nabla G(X_k, \xi)] = \nabla E[G(X_k, \xi)] \). When the gradient is not directly observable, procedures like finite difference methods, or simultaneous perturbations (Spall, 1992) can be used to estimate the gradient. When finite difference methods are used for gradient estimation in SA, it is called the Kiefer-Wolfowitz type algorithm (Kiefer et al., 1952). When finite difference methods are used for gradient estimation, the addition of another parameter, \( c_k \), further affects the convergence of SA type algorithms. Glasserman (1991) proposes a specialized gradient estimation procedure for discrete event systems called the infinitesimal perturbation analysis that provides unbiased estimates of the gradient. However, this requires special conditions on the structure of the problem.

In solving unconstrained SO problems of the form \( P_{SO} \), two important issues arise: (i) a procedure for estimating the gradient, and (ii) the sample sizes used in the estimation phase. In this thesis, we assume that an unbiased estimate of the gradient is readily available via a simulation oracle and refer interested readers to the aforementioned manuscripts for a detailed overview on gradient estimation emphasizing the sample size.
Notice that both the SA, and SAA described above work within an iterative framework where the number of realizations remains fixed in every iteration. Though, a large simulation effort at any visited solution yields a better estimate of the objective function, the iterative nature of the optimization, along with a limited computational budget for functional evaluation means that fixed sampling schemes are computationally inefficient options for SO. On the other hand, sample size, $m$, when too little, may lead to non-convergence of the algorithm. Later, in Chapter 3, we discuss variable sampling strategies to overcome shortfalls of fixed sample sizes along with the literature, and provide a new adaptive sampling strategy for continuous simulation optimization.

1.2.2 Integer Ordered Simulation Optimization

In this section, we discuss algorithms for simulation optimization involving integer ordered decision variables. When only a handful of candidate solutions are available, ranking and selection procedures are often utilized for selecting the best solution. When it is infeasible to enumerate all candidate solutions, more sophisticated algorithms are utilized. A majority of the search techniques in integer SO are random search methods, where as the name suggests, candidate solutions are chosen from the entire feasible space, either uniformly, oblivious of the solution quality, or based on the quality of the solution in the previous iteration.

One of the first random search methods for solving SO problems on integer ordered feasible sets is the Stochastic Ruler Method proposed by Yan and Mukai (1992). In this method, the subsequent solution to be visited is determined based on the objective function estimate of the current solution and a uniform random variable. Andradóttir (1999) improves the performance of random search methods with focus on further exploitation of visited solutions. Prudius (2007) proposes a class of adaptive random search methods that balance the exploration, and exploitation based on partitioning the feasible set into promising, and not promising regions, and choosing the subsequent set of solutions to visit by constructing a global, and a local sampling distribution.

Shi and Ólafsson (2000) propose the nested partitions method where subsequent solutions are chosen adaptively from nested sets that are pre-determined. Hong and Nelson (2006) propose COMPASS, an improvement over the nested partitions method to identify a local solution, where the promising region is updated at the end of every iteration by solving an optimization problem. However, the requirement to solve an optimization problem increases the computational overhead, and affects scalability of the algorithm to higher dimensions. The Industrial Strength - COMPASS (ISC) (Xu et al., 2010), an enhancement to COMPASS, improves the quality of solution obtained by restarting COMPASS multiple times, and returning the best solution out of the resulting local solutions using a ranking and selection procedure.
R-SPLINE (Wang, 2012) breaks from the tradition of random search methods, and exploits structure within the objective function by repeatedly performing a line search, and enumeration procedure to identify a local solution. R-SPLINE works within a retrospective framework—an optimization technique where sample path problems are solved sequentially starting with the optimal solution identified in the previous iteration. The sample size increases in the number of retrospective iterations so that the resulting sequence of approximate solutions would eventually converge to a local minimum. However, in R-SPLINE, and the random search methods discussed above (Yan and Mukai, 1992; Shi and Ólafsson, 2000; Hong and Nelson, 2006), the sampling scheme at any iteration is fixed, oblivious of the quality of the visited solution, often affecting finite time performance. In this thesis, we address this gap on choosing the sample size for integer ordered simulation optimization by developing ADALINE, an algorithm that adaptively determines the sample size without the requirement for any parameter tuning.

1.3 Organization of the thesis

The theme of this dissertation centers around developing algorithms for continuous and integer-ordered simulation optimization problems. In Chapter 2, we present ADALINE (Adaptive Piecewise Linear Interpolation with line search and Neighborhood Enumeration), an iterative method for determining a local minimum for integer-ordered SO problems. In ADALINE, a linear interpolation phase is first performed to identify a descent direction, followed by a neighborhood enumeration procedure to confirm the presence of a local minimum. As the name suggests, the sample size used for interpolation, and the neighborhood enumeration are chosen adaptively at every iteration. ADALINE is designed to be a non-terminating algorithm, that is, it will expend all available simulation budget on convergence to a local minimum. However, numerical results suggest that ADALINE converges to a local minimum much faster than a competing algorithm, R-SPLINE (Wang et al., 2013), utilizing only a fraction of the total simulation budget. In Chapter 3, we discuss the basic principles of line search techniques that are widely used in deterministic optimization, and extend these algorithms to work within an adaptive sampling framework for simulation optimization on continuous feasible sets. We present empirical results on the performance of the algorithm on the stochastic Aluffi Pentini and Rosenbrock functions.

An application of ADALINE to operating room rescheduling is discussed in Chapter 4. A threshold based scheduling heuristic is developed to accommodate semi-urgent surgeries, and elective surgeries under an uncertain operating room environment. The threshold levels are optimized using ADALINE, and a comparison of the solutions obtained using a Sample Average Approximation (SAA) approach and ADALINE is provided. We find that the adaptive sampling strategy in ADALINE converges to a better threshold level,
and is faster than the traditional SAA approach where the sample size is fixed.

Finally, MATLAB codes for the continuous simulation optimization algorithm, followed by Python implementations of ADALINE and the codes for the surgery rescheduling problem are listed in Appendix A.
Chapter 2

ADALINE for Integer-Ordered Simulation Optimization

In this chapter, we consider simulation optimization (SO) problems where the decision variables are integer-ordered, and the objective function is of the form of an expectation that cannot be analytically computed, but only “noisy” observations can be observed through simulation experiments. Our objective is to develop an algorithm that will identify a local minimum for such problems. Formally, the integer-ordered SO problem can be stated as:

$$P_I : \min g(x)$$
subject to \( x \in X \subseteq \mathbb{Z}^d \)

where \( X \) denotes the feasible set. The objective function, \( g : X \subseteq \mathbb{Z}^d \rightarrow \mathbb{R} \), is an expectation, given by \( g(x) = \mathbb{E}_{\xi}[G(x, \xi)] \), where \( \xi \) is a random vector defined on the probability space \( (\Omega, \mathcal{A}, P) \). At any decision point, \( x \), the analytical form of \( g(x) \) is unavailable and can only be estimated by using random observations, \( G(x, \cdot) \), by calling a Monte Carlo simulation oracle. We assume that an unbiased and strongly consistent estimator of the objective function is available, that is, \( \hat{g}_m(x) = m^{-1} \sum_{j=1}^{m} G(x, \xi_j) \) such that \( E[\hat{g}_m(x)] = g(x) \), and \( \lim_{m \rightarrow \infty} \hat{g}_m(x) = g(x) \), respectively. The feasibility is implicitly determined by the simulation oracle, that is, in addition to returning the random objective function, the oracle also flags an infeasible solution.

SO with integer-ordered decision variables find applications in a variety of settings, including, but not
limited to, inventory replenishment policy optimization (Jalali and Nieuwenhuyse, 2015), revenue management (Gosavi et al., 2007), surgery planning and scheduling (Lamiri et al., 2009), and determining vaccine allocation strategies to prevent an epidemic outbreak (Eubank et al., 2004).

Similar to continuous simulation optimization, the majority of algorithms that have so far been proposed to solve $P_I$ (Hong and Nelson, 2006; Wang et al., 2013; Xu et al., 2010) require that the sample size for estimating the objective function be a parameter specified by the user, and a majority of them operate on an explicitly defined feasible set. Recall from Chapter 1, that predetermined sample sizes can lead to inefficiency in the SO algorithm by either sampling too little at points close to a locally optimal solution or too much far away from a locally optimal solution. Thus, our objective in this chapter is to focus on integer-ordered feasible sets.

In the context of integer-ordered simulation optimization, a feasible solution $\mathbf{x}^*$ is a local minimum if and only if all other feasible solutions in the $N^1$ neighborhood are inferior to $\mathbf{x}^*$ as measured by the objective function $g$; thus, $\mathbf{x}^*$ is a local minimum if $g(\mathbf{x}^*) \leq g(\mathbf{x})$, $\forall \mathbf{x} \in N^1(\mathbf{x}^*) \cap X$. We define a $N^1$ neighborhood as follows: $N^1(\mathbf{x}^*) = \{ \mathbf{x} : ||\mathbf{x}^* - \mathbf{x}|| = 1 \}$. In this chapter, we propose ADALINE- Integer Ordered (Adaptive Piecewise Linear Interpolation with line search and Neighborhood Enumeration), an integer-ordered SO algorithm, that adaptively determines the sample size in a line search framework and that can guarantee convergence to a local minimum.

### 2.1 Competitors

Major competitors for ADALINE include the related line search based algorithm, R-SPLINE (Wang et al., 2013); popular partitioning based algorithms – COMPASS (Hong and Nelson, 2006), the enhanced Industrial Strength COMPASS (ISC) (Xu et al., 2010); and the recent Gaussian Process-based Search (GPS) (Sun et al., 2014). Operating on an explicitly defined feasible space, COMPASS iteratively partitions and expends simulation effort to solutions from the “promising regions” of the feasible space to guarantee convergence to a local solution. The promising region is updated at the end of every iteration by solving an optimization problem after identifying and grouping solutions that are “close” to the best solution. COMPASS implicitly exploits the structure of the objective function, though the requirement to solve an optimization problem affects scalability to higher dimensions.

The Industrial Strength COMPASS (ISC), an enhancement over COMPASS, returns a good quality local solution by restarting COMPASS several times starting with promising initial solutions obtained using a genetic algorithm (GA). The best solution out of the resulting local solutions is returned by using a ranking
and selection (R & S) algorithm. However, ISC still suffers from the issue of scaling well to higher dimensions. Both COMPASS and ISC exploit the structure of the objective function implicitly by sampling more at points which are promising, whereas, R-SPLINE and our adaptive search algorithm rely on phantom gradients, evaluated by differencing the objective function values through an interpolation procedure, to exploit the structure of the objective function. GPS is a random search method that balances the exploration and exploitation trade-off by generating the sampling distribution that allocates further simulation effort using a Gaussian process. Of all the above algorithms, GPS guarantees convergence to a global solution as the sample size, \( M \), goes to infinity, whereas, the other algorithms guarantee convergence to a local solution.

R-SPLINE, a line search based algorithm identifies a local minimum by operating within a retrospective framework, where a sequence of sample path problems are solved with increasing sample sizes at each phase. Each retrospective phase involves an interpolation and a neighborhood enumeration procedure, where interpolation with line search exploits the structural information on the objective function by estimating a pseudogradient, and the neighborhood enumeration (NE) acting as a confirmatory procedure for a local minimum. In addition to acting as a confirmatory test, the NE procedure signals an increase in sample size for the subsequent sample path problem (Wang et al., 2013).

One of the key differentiations between R-SPLINE and ADALINE is how the neighborhood enumeration procedure is performed. In R-SPLINE, once a sample path problem is solved to optimality, an increase in sample size is signaled, making it impossible to revert back to a smaller sample size. Though such a framework to increase sample size would guarantee convergence to a local minimum, improving finite time performance may require extensive parameter tuning.

### 2.2 Algorithm ADALINE - Integer Ordered

We provide a broad outline of the algorithm, ADALINE (Adaptive Piecewise Linear Interpolation with line search and Neighborhood Enumeration - Integer Ordered), which we see as an improvement over an existing integer-ordered algorithm R-SPLINE. As the name suggests, ADALINE is an iterative procedure that incorporates an interpolation phase with a gradient based search (LI) and a neighborhood enumeration (NE) procedure. The choice of sample size for estimating the gradient, performing the line search and the neighborhood enumeration are determined adaptively at each iteration. Specifically, each iteration of ADALINE performs the following steps.

S.1 Perturb an integer solution and estimate the pseudogradient within an adaptive sampling framework;

S.2 Perform a line search along the negative gradient direction identified in S.1 to identify a better solution;
S.3 Perform a strategic neighborhood enumeration of the $N^1$ neighborhood to confirm a local minimum.

At each one of the $2d$ neighbors in the $N^1$ neighborhood, an estimate of the objective function is obtained by expending a simulation effort of $M^I$. If a trivial search in the neighborhood fails to yield a better solution, it indicates that $M^I$ is insufficient, and an adaptive neighborhood enumeration is performed. In Section 2.2, we will explain in more detail, the adaptive neighborhood enumeration procedure. The trajectory of the algorithm is illustrated in Figure 2.1. Starting with an initial solution, $X_0$, ADALINE performs an adaptive interpolation procedure, followed by a line search that terminates at $X_0^{NE}$. At $X_0^{NE}$, an adaptive NE procedure is performed such that $X_1$ is deemed the best solution at the end of iteration 1. A similar sequence of adaptive interpolation, followed by a neighborhood enumeration, results in ADALINE converging to $X^*$, a local minimum in the $N^1$ neighborhood, in the subsequent iterations. ADALINE

Figure 2.1: Trajectory of ADALINE

is a non-terminating algorithm, that is, on convergence to a local minimum, the algorithm is designed to expend infinite simulation effort. For all practical purposes, the algorithm can be stopped upon expending a maximum simulation budget, denoted by $B$. In the following section, we provide a detailed description of the adaptive linear interpolation and the NE procedure.
Algorithm 1 ADALINE

Require: Initial solution, $x_0$; Minimum sample size, $M_{min}$; Confidence level, $\alpha$; Total simulation budget, $B$; $s_0 = 2$, $c = 2$, $j_{max} = 25$

Ensure: Solution, $X^* \in X$ such that $\hat{g}(X^*) < \hat{g}(X), \forall X \in N^R(X^*), R = 1$

1: Set $k \leftarrow 0$; $X_k \leftarrow x_0$; $ncalls \leftarrow 0$
2: repeat Piecewise-linear Interpolation (PLI) and Neighborhood Enumeration (NE)
3: \[ [X_{PLI}, M^I, \hat{\gamma}] = PLI(X_k, M_{min}) \]
4: Set $Y^0 \leftarrow X_{PLI}$; $j \leftarrow 0$. Update $ncalls$ after PLI
5: repeat Line Search
6: \[ j \leftarrow j + 1; s = c^{j-1} \times s_0; Y^j = Y^0 - s \times \hat{\gamma}/||\hat{\gamma}|| \]
7: Estimate $\hat{g}_{MI}(Y^j)$; $ncalls \leftarrow ncalls + M^I$
8: until $\hat{g}_{MI}(Y^j) > \hat{g}_{MI}(Y^{j-1})$ or $Y^j$ is infeasible or $j < j_{max}$
9: $X_{best} \leftarrow Y^{j-1}$
10: \[ [X_{best}, M^E] = NE(X_{best}, M^I) \]
11: $ncalls \leftarrow ncalls + \sum_{i=1}^{2d} M^E_i$
12: $k \leftarrow k + 1$; $X_k \leftarrow X_{best}$
13: until $ncalls > B$

2.2.1 Adaptive Linear Interpolation and Line Search

Adaptive piecewise-linear interpolation, LI, marks the beginning of an iteration in ADALINE, where an interpolation followed by a line search is sequentially employed starting with an initial feasible solution, $X_k$.

Similar to other line search techniques, it involves two phases, namely, (i) a gradient estimation phase, and (ii) a line search phase. However, unlike continuous optimization, it should be noted that a gradient based search strategy in integer-ordered space presents two major challenges, (i) function is not differentiable, and hence its gradient and descent direction is undefined; and (ii) points visited during the line search may be infeasible, and additional rules to guarantee feasibility needs to be enforced.

In order to ensure that feasible solutions are returned from a line search, we can project an infeasible solution onto the convex hull of the feasible set, $X$, by rounding non-integer points to the closest feasible integer point. Concerning the descent direction, finite difference methods are commonly used to approximate the gradient when it is undefined. However, they are computationally intensive and fail to yield a reasonable approximation under certain scenarios, as observed in Atlason et al. (2003), when all the points are within unit distance from the current solution. Given the iterative nature of the algorithm, the search procedure may be performed multiple times, making it imperative that it is computationally efficient. In addition to the above, the question of, “What should be the sample size for simulation?,” adds to the complexity of the problem.

Therefore, the search procedure we utilize should serve two major purposes: (i) provide a reasonable approximation to the gradient, and the descent direction, and (ii) find the optimal simulation effort to expend for gradient estimation, and subsequently the line search. One such technique that is efficient, and that
provides a good approximation to the gradient is the piecewise-linear interpolation (PLI) procedure utilized in R-SPLINE (Wang et al., 2013). It involves the approximation of descent direction by systematically simulating at points that are sufficiently far apart in the feasible set.

In PLI, otherwise called simplex interpolation, the feasible point is first extended to the continuous space, followed by the construction of a $d$-simplex that encloses the perturbed point. To be more precise, consider the perturbation of $X_k \in \mathbb{Z}^d$ to the continuous space, such that the resulting point, $X \in \mathbb{R}^d \setminus \mathbb{Z}^d$. Now, of the $2^d$ vertices that enclose the perturbed point, simplex interpolation requires that $d + 1$ vertices be chosen. The objective is to choose the vertices carefully so that the perturbed point, $X$, lies in the convex hull of the simplex, and the objective function at the perturbed point, $g(X)$, can be written as a convex combination of the functional value at each one of the $d + 1$ vertices.

We retain the same procedure from R-SPLINE to construct the $d$-simplex, where the first vertex of the simplex is chosen to be the integer floor of all elements in $X$, denoted by $X^0$. Subsequent vertices, $X^1, \ldots, X^d$ are chosen to be exactly a unit distance from each other in the order of elements of $X$ with the greatest fractional part. Since this is no different than the one described in the PLI phase of R-SPLINE, we refer to Section 4 of Wang et al. (2013) for further information and only provide an example of how the simplex is constructed.

Suppose, $X = (2.2, 5.1, 8.3)$, the $d + 1$ vertices of the simplex would be $X^0 = ((2.2), (5.1), (8.3)) = (2, 5, 8)$, $X^1 = (2, 5, 9)$, $X^2 = (3, 5, 9)$ and $X^3 = (3, 6, 9)$, respectively. Once the vertices of the simplex are identified, functional estimates are obtained to interpolate the objective function and also to estimate the direction of descent. However, it should be noted, that in R-SPLINE, the sample size at each of these $d + 1$ vertices is specified at the beginning of an iteration, whereas in the adaptive version, a sampling rule that determines the simulation effort to be expended is enforced.

Recall that the estimator of $g(X)$ is strongly consistent, that is, $\lim_{M \to \infty} \hat{g}_M(X) = g(X)$. If the minimum sample size, $M_{min}$ is ‘big enough,’ the obtained estimates and the resulting approximation to the descent direction would be reliable. However, what constitutes a ‘big enough’ sample size depends on the problem and a large sample size to estimate the pseudogradient leads to inefficiency, necessitating an optimal allocation rule. We build upon existing sequential sampling schemes described in Hashemi et al. (2014); Pasupathy and Schmeiser (2010) and allocate simulation effort as a trade-off between the optimization error (norm of the pseudogradient) and the standard error. Mathematically, the sampling rule can be written as:

$$M^I = \inf \{ m : \kappa(\alpha, m) \times \hat{se}_m(\hat{\gamma}) \leq \| \hat{\gamma} \| \},$$

where $\hat{se}$ denotes the standard error on the estimated pseudogradient, $\| \hat{\gamma} \|$ is the estimated norm, and $\kappa$ is a constant. $\kappa(\alpha, m)$ can be chosen to be the $\alpha$th percentile of a Student’s $t$ distribution with $m - 1$ degrees
of freedom. User specified parameters at the start of an iteration include the minimum sample size, \( M_{\text{min}} \), and the confidence level, \( \alpha \). The sample size for line search and neighborhood enumeration that follows the interpolation phase is set to be \( M^I \) and the best integer solution at the end of interpolation, \( X_{PLI} \), is updated.

A line search is performed only when all \( d + 1 \) vertices in the simplex interpolation phase are feasible; otherwise, only the best solution from interpolation, \( X_{PLI} \) is updated. Parameters used in the line search include, (i) the initial step size, \( s_0 \), and (ii) the step size increment, \( c \). The search identifies a sequence of solutions starting with the initial point, \( X_{PLI} \), and along the descent direction, \(-\hat{\gamma} \). If \( Y \) denotes a solution visited during the search, probability of \( Y \) being feasible is arbitrarily small. Under such circumstances, rather than performing an interpolation at every visited solution, we conserve simulation effort by rounding every element of \( Y \) to the nearest integer and estimate \( \hat{g}_{M^I}(Y) \).

The line search is terminated at one of the following events: (i) the line search results in an infeasible solution, or (ii) the line search results in a solution that is inferior a prior solution in the sequence, or (iii) the number of simulation calls (\( n_{\text{calls}} \)) exceeds the total simulation budget, \( B \). On termination of the line search, the best available integer solution is updated to be \( X_{\text{best}} \).

### 2.2.2 Neighborhood Enumeration (NE)

The NE procedure in ADALINE is a confirmatory test for a local minimum, that is, it verifies if any better solution exists in the neighborhood of \( X_{\text{best}} \). However, in addition to exploring solutions in the neighborhood, ADALINE also includes an optional co-ordinate search, when there is sufficient confidence that a search along a sub-set of coordinates may improve the objective function. The three critical procedures in NE are given as follows: (i) Enumeration and initial estimation, (ii) Confirmatory check for a local minimum, (iii a) Identification of \( X_{\text{best}} \) (iii b) Adaptive sampling. Only on failing to identify a better solution in the neighborhood, an adaptive neighborhood enumeration is performed.

**Enumeration and Initial Estimation**

This is the exploratory phase of neighborhood enumeration where all feasible points in the \( N^R \) neighborhood of the iterate, \( X_{\text{best}} \) are identified. Let \( F \) denote the set of all feasible solutions. An initial estimate obtained using simulation effort, \( M^I \). If \( R = 1 \), the total number of neighboring points that are exactly one unit away from \( X_{\text{best}} \) is \( 2d \). For example, if \( d = 2 \) and \( X_{\text{best}} = (3, 5) \), then \( N^1(X_{\text{best}}) = \{(4, 5), (2, 5), (3, 6), (3, 4)\} \).

Once all feasible solutions in the neighborhood are identified, at every solution, denoted by \( X^i \), \( \forall i \in N^1(X_{\text{best}}) \), and the incumbent solution, \( X_{\text{best}} \), estimates of the sample mean, \( \hat{g}_{M^I}(X^i) \), and the sample
Algorithm 2 Adaptive Piecewise Linear Interpolation

Require: \(X = (X_1, \cdots, X_d) \in \mathbb{R}^d \setminus \mathbb{Z}^d\); Minimum sample size \(M_{\min}\); Confidence level \(\alpha\); Sample size increment, \(\delta\)

Ensure: Function value \(\hat{g}_{M^t}(X)\); Pseudo-gradient \(\hat{\gamma}\) of the function \(\hat{g}_{M^t}(\cdot)\) at \(X\); Sample size \(M^t\)

1: Set \(X^0 \leftarrow [X]\)
2: Set \(Z_i \leftarrow X_i - X_i^0\) for \(i = 1, \cdots, d\)
3: Set \(Z \leftarrow (Z_1, \cdots, Z_d)\) to get \(1 = Z_{p(0)} > Z_{p(1)} \geq Z_{p(2)} \geq \cdots \geq Z_{p(d+1)} = 0\)
4: Set \(X^t \leftarrow X^{t-1} + \epsilon_{p(t)}\) for \(i = 1, \cdots, d\)
5: Set \(w_i \leftarrow Z_{p(i)} - Z_{p(i+1)}\) for \(i = 0, 1, \cdots, d\)
6: Set \(M^t \leftarrow M_{\min}\)
7: Set \(feas \leftarrow 0; t \leftarrow 0; \hat{g}_{M^t}(X) \leftarrow 0\)
8: for \(i \leftarrow 0, d\) do
9: \hspace{1em} if \(X^t\) is feasible then
10: \hspace{2em} Estimate \(\hat{g}_{M^t}(X^t) = \frac{1}{M^t} \sum_{j=1}^{M^t} G_j(X^t)\)
11: \hspace{2em} feas \leftarrow feas + 1; t \leftarrow t + w_i; \text{ and } \hat{g}_{M^t}(X) \leftarrow \hat{g}_{M^t}(X) + w_i \hat{g}_{M^t}(X^t)\)
12: end if
13: end for
14: if \(t > 0\) \(\hat{g}_{M^t}(X) \leftarrow \frac{\hat{g}_{M^t}(X)}{t}\) else \(\hat{g}_{M^t}(X) \leftarrow \infty\) end if
15: if \(feas < d + 1\) return \(\hat{g}_{M^t}(X)\)
16: if \(feas = d + 1\) then
17: \(\hat{\gamma}_{p(i)} \leftarrow \hat{g}_{M^t}(X^t) - \hat{g}_{M^t}(X^{t-1})\) for \(i = 1, \cdots, d\)
18: \(\hat{\sigma}^2_{M^t}(\hat{\gamma}_{p(i)}) \leftarrow \frac{\hat{\sigma}^2_{M^t}(G(X^t)) + \hat{\sigma}^2_{M^t}(G(X^{t-1})) - 2\hat{\sigma}(G(X^t), G(X^{t-1}))}{M^t}\) for \(i = 1, \cdots, d\)
19: \(\hat{\sigma}^2_{M^t}(\hat{\gamma}) \leftarrow \sqrt{\sum_{i=1}^{d} \hat{\sigma}^2_{M^t}(\hat{\gamma}_{p(i)})}\)
20: while \(\kappa(\alpha, M^t) \times \hat{\sigma}^2_{M^t}(\hat{\gamma}) > \| \hat{\gamma} \|\) do
21: \(M^t \leftarrow M^t + \delta\)
22: Sample \(\delta\) more observations from \(X^t\), \(\forall i \in 0, \cdots, d\)
23: Update \(\hat{g}_{M^t}(X^t)\) for \(i = 0, 1, \cdots, d\)
24: Update \(\hat{g}_{M^t}(P_i); \hat{\gamma}; \hat{\sigma}^2_{M^t}(\hat{\gamma})\)
25: end while
26: return \(\hat{g}_{M^t}(P_i); \hat{\gamma}; M^t\)
27: end if
Prasanna Kumar Ragavan  

Chapter 2. Integer-Ordered Simulation Optimization  

17

variance, \( \hat{\sigma}^2_{M^I}(X^i) \) are obtained. The difference in sample mean between the incumbent solution and a neighbor, denoted by \( \hat{\mu}^i \), is:

\[
\hat{\mu}^i = \hat{g}_{M^I}(X^i) - \hat{g}_{M^I}(X^i), \forall i \in 1, \cdots, 2d,
\]

We write the standard error of the difference in sample mean as follows:

\[
\hat{se}(\hat{\mu}^i) = \sqrt{\hat{\sigma}^2_{M^I}(G(X^i)) + \hat{\sigma}^2_{M^I}(G(X^i)) - 2\text{cov}(G(X^i), G(X^i)) \over M^I}
\]

**Confirmatory Check for a local minimum**

When \( \hat{\mu}^i > 0, \forall i \in N^I(X_{best}) \), it indicates that a unit step along that coordinate direction results in a better solution than \( X_{best} \). Let \( Q \) denote the set of all neighbors better than the incumbent,

\[
Q = \{ i : i \in N^I(X_{best}), \hat{\mu}^i > 0 \},
\]

and thus \( Q^C = F \setminus Q \). Conditional on the cardinality of \( Q \), two different trajectories for the algorithm are possible:

**CASE A** (\( |Q| > 0 \)) : Update \( X_{best} \) based on the best solution observed in the neighborhood.

**CASE B** (\( |Q| = 0 \)) : If none of the solutions are better than the incumbent solution, either \( X_{best} \) is a local minimum or the sample size from interpolation, \( M^I \) is insufficient to distinguish between the neighbors and the incumbent solution. This condition triggers the adaptive neighborhood enumeration, where the sample size is strategically increased across solutions in the neighborhood.

**Identify a better solution - CASE A**

As the name suggests, in this phase, \( X_{best} \) is updated to be the best solution in \( Q \). However, if the sample size, \( M^I \), is sufficient to establish that the difference in means is significant, a coordinate search will improve finite time performance. A neighboring solution is significantly different if the standard error on the difference in sample means is less than the absolute value of the difference. Thus, to establish significant difference in sample mean, following our previous definition of optimal sampling, the condition, \( \kappa(\alpha, M^I) \hat{se}_{M^I}(\hat{\mu}^i) \leq |\hat{\mu}^i| \) should hold, and we let \( S \) denote the set of all neighbors satisfying the same:

\[
S = \{ i : i \in N^I(X_{best}), \hat{se}_{M^I}(\hat{\mu}^i) \leq |\hat{\mu}^i| \}
\]
Adaptive Neighborhood Enumeration - CASE B

Adaptive neighborhood enumeration forms the exploitative phase of ADALINE, where the objective is to confirm a local minimum by repeated sampling, when we fail to identify a better solution in the neighborhood based on the initial estimates. Sampling is strategically repeated across feasible solutions in the neighborhood, and the incumbent solution, $X_{best}$, until the budget is expended, or a significantly better solution is identified. Any solution in the set, $Q \cap S$ should be deemed “significantly better”. We follow a three step procedure described below in allocating simulation effort across all feasible solutions in the neighborhood:

1. **Stratify**: Based on the initial estimates of $\hat{\mu}_i, \hat{se}(\hat{\mu}_i)$ obtained by expending $M^i$, we classify all feasible neighbors into one of three categories as follows:

   (i) Set $Q \cap S$, the set of all neighbors that satisfy sampling criterion in (2.1), and are also promising enough to be a descent direction; that is,
   
   $$Q \cap S \leftrightarrow \{i : i \in N^1(X), \kappa(\alpha, M_i) \times \hat{se}(\hat{\mu}_i) \leq |\hat{\mu}_i|, \text{and } \hat{\mu}_i > 0\}. $$

   (ii) Set $Q^c \cap S$, the set of all neighbors that satisfy sampling criterion in (2.1), but are not promising enough to be a descent direction; that is,
   
   $$Q^c \cap S \leftrightarrow \{i : i \in N^1(X), \kappa(\alpha, M_i) \times \hat{se}(\hat{\mu}_i) \leq |\hat{\mu}_i|, \text{and } \hat{\mu}_i \leq 0\}. $$

   (iii) $S^c$, the set of all neighbors that do not satisfy sampling criterion in (2.1).

   Note that adaptive neighborhood enumeration is performed only when the set $Q = \emptyset$.

2. **Prioritize** Upon classification of neighbors, the following actions are possible depending on the state of the neighboring solutions:

   (i) When $|Q \cap S| > 0$, it indicates that a subset of significantly better solutions have been identified and hence the neighborhood enumeration procedure is terminated.

   (ii) When $|Q \cap S| = 0$, it indicates that further sampling is needed to identify a significantly better solution. Instead of allocating effort randomly across the neighbors, we use the following ordering to determine where further simulation effort should be expended:

   $$S^c \prec Q^c \cap S, $$
where $\prec$ indicates the precedence constraint, that is, the non-empty set to the left of $\prec$ has higher priority over the set to the right. Thus, sampling effort is uniformly allocated only to neighbors where the difference in sample means is not significant.

3. **Simulate and Update**: Once the set of neighbors where further simulation effort should be allocated are identified, we further collect $\delta$ independent samples at each of those neighboring solutions. Based on the new sample information, $\hat{\mu}^i$, $\hat{\sigma}(\mu^i)$ and all other relevant sample statistics are updated and the stratification rule is applied again to check if the termination criterion for NE is satisfied. Else, the above three steps are repeated until $|Q \cap S| > 0$.

If $X_{best}$ is indeed a local minimum, all the remaining simulation budget is expended between the incumbent and its neighbors, increasing the confidence on the estimated objective.

**Remark.** With the availability of new information, a neighboring solution may repeatedly transition between $Q^c \cap S$, $Q \cap S^c$, and $Q^c \cap S^c$. Under such circumstances, after a set has been chosen for further simulation, we equalize the sample size at all neighbors before expending $\delta$ more simulations.

**Coordinate Search**

Once the set of all significantly better solutions are identified, estimates of the partial gradients in the neighborhood is used to perform a block coordinate search. The gradient direction is a $d$ dimensional vector, thus we first resolve the neighbors in $A$ to the corresponding co-ordinate axis ($j = 1, \cdots, d$), and then estimate the gradient along each coordinate axis as follows:

$$
\hat{g}^j = \begin{cases} 
(-1)^{i+1} \left( \hat{g}_{M^i}(X^i) - \hat{g}_{M}(X) \right), & j = \left\lfloor \frac{i}{2} \right\rfloor \text{ and } \forall i \in A \\
0, & \text{otherwise.}
\end{cases}
$$

The coordinate search is similar to line search described in Section 2.2.1 and does not require much elaboration. In the coordinate search, instead of searching along the entire decision vector, we restrict our search to a subspace.

ADALINE continues to alternate between the enumeration procedure and the coordinate search until all the simulation budget is expended. The solution with the lowest estimated objective function is chosen to be $X_{best}$ and returned to the user as the solution to Problem (1).
Algorithm 3 Neighborhood Enumeration

Require: Best integer solution, \(X_{\text{best}} = (X_1, \ldots, X_d) \in \mathbb{X} \cap \mathbb{Z}^d\); Interpolation sample size, \(M_t^E\); Sample size increment, \(\delta\); Confidence level \(\alpha\), Simulation calls, \(n_{\text{calls}}\); Simulation budget, \(B\)

Ensure: Best solution, \(X_{\text{best}}\); Functional estimate, \(\hat{g}(X_{\text{best}})\); Coordinate(s) to descend, \(\hat{\Gamma}\); Sample size vector, \(M_t^E\)

1: Set \(X^0 \leftarrow X_{\text{best}}\). Find the \(2d\) neighborhood points of \(X^0\) in the first-order neighborhood, i.e., determine \(N^1(X^0)\). Set it equal to \(X^1\), for \(i = 1, \ldots, 2d\)
2: Set \(\hat{\Gamma} = 0\), for \(j = 1, \ldots, d\)
3: for \(i = 0, \ldots, 2d\) do
4: if \(X^i\) is feasible then \(\hat{g}_{M_{\min}}(X^i)\); \(M_t^E \leftarrow M_t^E\) end if
5: if \(i > 0\) then
6: Set \(\hat{\mu}^i \leftarrow \hat{g}_{M_t^E}(X^0) - \hat{g}_{M_t^E}(X^i)\)
7: Set \(\hat{\sigma}_t^2(\hat{\mu}^i) \leftarrow \hat{\sigma}_{M_t^E}^2(G(X^i)) + \hat{\sigma}_{M_t^E}^2(G(X^0)) - 2 \hat{\text{cov}}(G(X^i), G(X^0))\)
8: Set \(\hat{\text{se}}(\hat{\mu}^i) \leftarrow \frac{\hat{\sigma}_{M_t^E}(\hat{\mu}^i)}{\sqrt{M_t^E}}, \beta_t^i \leftarrow \frac{\hat{\mu}^i}{\hat{\text{se}}(\hat{\mu}^i)}\)
9: end if
10: end for
11: Set \(Q \leftarrow \{i : i \in N^1(X^0), \hat{\mu}^i > 0\}\); \(S \leftarrow \{i : i \in N^1(X^0), \kappa(\alpha, M_t^E) \times \hat{\text{se}}(\hat{\mu}^i) \leq |\hat{\mu}^i|\}\)
12: if \(|Q| > 0\) then
13: CASE A
14: if \(|Q \cap S| > 0\) then
15: for \(j \in Q \cap S\) do
16: \(\hat{\Gamma}_{t+1} \leftarrow (-1)^{j+1} \left(\hat{g}_{M_t^E}(X^j) - \hat{g}_{M_t^E}(X^0)\right)\)
17: end for
18: Perform line search along \(\hat{\Gamma}\) starting with \(X_{\text{best}}\). Update \(X_{\text{best}}\).
19: else
20: Set \(X_{\text{best}} \leftarrow X_q\), where \(q = \arg \min_{i \in 1,\ldots,2d} \hat{g}_{M_t^E}(X^i)\)
21: end if
22: Terminate algorithm
23: else
24: CASE B
25: while \(n_{\text{calls}} < B\) and \(|Q \cap S| = 0\) do
26: if \(|S^C| > 0\) then
27: \(K \leftarrow S^C\)
28: else if \(|Q^C \cap S| > 0\) then
29: \(K \leftarrow Q^C \cap S\)
30: end if
31: if For all \(i \in K\), \(M_{t+1}^E = M_t^E\) then

for $i \in \mathcal{K} \cup \{0\}$ do
  $M_i^E \leftarrow M_i^E + \delta$
  Estimate $\hat{g}_{M_i^E}(X^i), \hat{\sigma}_{M_i^E}^2(X^i), \text{cov}(G(X^i), G(X^0))$
end for

else
  for $i = \{i : i \in \mathcal{K}, M_i^E < M_0^E\}$ do
    $M_i^E \leftarrow M_i^E + (M_0^E - M_i^E)$
    Estimate $\hat{g}_{M_i^E}(X^i), \hat{\sigma}_{M_i^E}^2(X^i), \text{cov}(G(X^i), G(X^0))$
  end for
end if

Update $\hat{\mu}_i, \hat{\sigma}_i^2(\hat{\mu}_i), \text{se}(\hat{\mu}_i), n_{calls}$

Update $Q, S$ using conditions in line 11.

end while

for $i \in Q \cap S$ do
  $\hat{\Gamma}^{|\frac{1}{2}} \leftarrow (-1)^{|i+1|} (\hat{g}_{M_i}(X^i) - \hat{g}_M(X^0))$
end for

Terminate algorithm

2.3 Local Convergence

Let $\mathcal{F}$ denote the set of all feasible points, and $\mathcal{L}(\mathcal{N}^1)$ denote the set of local minima. Under certain assumptions on the feasible set, we show that the sequence of points visited by ADALINE, denoted by $\{X_k\}$, converges with probability one (wp1) to the set of local minima, $\mathcal{L}(\mathcal{N}^1)$. In order to prove convergence of ADALINE, we use the following assumptions:

Assumption 1. The set of all feasible points, denoted by $\mathcal{F}$, is finite, that is, $|\mathcal{F}| < \infty$.

Assumption 2. The objective function is of the form of an expectation, $g(x) = \mathbb{E}[G(x, \xi)]$. An unbiased, and strongly consistent estimator of the objective function, $\hat{g}_m(x)$ is available such that, $\mathbb{E}[^{\hat{\mu}_i}, \hat{\sigma}_i^2(\hat{\mu}_i), \text{se}(\hat{\mu}_i), n_{calls}$

Assumption 3. For each $x \in \mathcal{F}$, the objective function is finite, i.e., $\mathbb{E}[G(x, \xi)] < \infty$.

Assumption 4. For each $x \in \mathcal{F}$, the variance of the random simulation observations is finite, i.e., $\text{Var}(G(x, \xi)) < \infty$.

Lemma 1. Under Assumptions 1 - 4, the following assertions hold:

(i) Suppose $x \in \mathcal{L}(\mathcal{N}^1)$. Then $\mathbb{P}(X_i = x \forall l > k | X_k = x) > 0$.

(ii) Suppose $x \notin \mathcal{L}(\mathcal{N}^1)$. Then $\mathbb{P}(X_i = x \forall l > k | X_k = x) = 0$. 


Proof. Let \{G_j\} be the sequence of i.i.d random variables corresponding to observations from a simulation at any design point.

(i) Consider \(x \in \mathcal{L}(\mathcal{N}^1)\) and \(y \in \mathcal{F} \setminus \mathcal{L}(\mathcal{N}^1)\). Let \(Z_j = G_j(x) - G_j(y)\). The difference in sample means, \(\hat{g}_m(x) - \hat{g}_m(y) = m^{-1} \sum_{j=1}^m Z_j = \bar{Z}\). Note that the stochastic process corresponding to the summation of the sequence of i.i.d random variables, \(\{Z_j\}\) constitutes a random walk. Since by definition, \(g(x) - g(y) = \mu < 0\), the random walk generated by the summation process, \(\sum Z_j\) drifts downwards. For a local minimum to be deemed the incumbent point for all \(l > k\), it indicates that the sampling criterion is not satisfied at the incumbent solution. Therefore, we need to show that, \(\mathbb{P}(\bar{Z} < c \frac{\hat{\sigma}}{\sqrt{m}} \text{ for all } m) > 0\). Let \(Z_0 = \delta > 0\).

\[
\mathbb{P}(\bar{Z} < c \frac{\hat{\sigma}}{\sqrt{m}} \text{ for all } m \mid Z_0 = \delta) > \mathbb{P}(\bar{Z} < 0 \text{ for all } m \mid Z_0 = \delta)
\]

\[
= \mathbb{P}(\sum_{j=1}^m Z_j < 0 \text{ for all } m \mid Z_0 = \delta)
\]

Let \(\phi(\lambda) = \exp\{\Psi(\lambda)\}\), where \(\Psi(\lambda)\) is the cumulant generating function (log of the moment generating function) of a Normal random variable given by \(\Psi(\lambda) = \lambda t + \frac{1}{2} \lambda^2 t^2\). Thus \((\phi(\lambda))^m = \exp\{m\Psi(\lambda)\}\). Then, the random variable,

\[
Y_m = \exp\{\lambda(Z_1 + \cdots + Z_m) - m\Psi(\lambda)\}
\]

is a martingale. Choose \(\lambda^* \neq 0\) such that, \(\Psi(\lambda^*) = 0\). Consider the following stopping time,

\[
T = \inf\{m : \sum Z_j \geq a \text{ or } \sum Z_j \leq b\}
\]

Using the optional sampling theorem (Williams, 1991),

\[
p_a \exp(\lambda^*a) + (1 - p_a) \exp(\lambda^*b) = 1.
\]

Then, \(p_a = \frac{1 - \exp(\lambda^*a)}{\exp(\lambda^*a) - \exp(\lambda^*b)}\) and \(p_b = \frac{1 - \exp(\lambda^*a)}{\exp(\lambda^*b) - \exp(\lambda^*a)}\). Setting \(b \to -\infty\), \(\lim_{b \to -\infty} p_a = \exp(-\lambda^*a)\). Therefore, \(\mathbb{P}(\sum_{j=1}^m Z_j < 0 \text{ for all } m \mid Z_0 = \delta) = \exp(-\lambda^*\delta)\). For any \(\delta > 0\), \(\exp(-\lambda^*\delta) > 0\).

(ii) Consider \(x \notin \mathcal{L}(\mathcal{N}^1)\). By definition, there exists a \(y \in \mathcal{N}^1(x)\) such that \(g(y) < g(x)\). Therefore, the difference in sample means is \(\hat{g}_m(x) - \hat{g}_m(y) = m^{-1} \sum_{j=1}^m Z_j = \bar{Z}\), with \(\mathbb{E}[\bar{Z}] = \mu > 0\). Thus, the random walk corresponding to the summation of \(Z_j\) has a positive drift. Therefore, we need to show that, \(\mathbb{P}(\bar{Z} < c \frac{\hat{\sigma}}{\sqrt{m}} \text{ for all } m) = 0\). We know that:

(i) \(\hat{\sigma}_m \to \sigma \text{ wp1 as } m \to \infty\).
(ii) If $\sum_j Z_j$ is a random walk with drift, then, from the law of the iterated logarithm,

$$\limsup_m \frac{S_m - m\mu}{\sqrt{m \log \log m}} = \sqrt{2}$$

where $S_m = \sum_j Z_j$.

From (i), we see that for $m$ large enough, $\frac{\hat{\sigma}_m}{\sigma} < (1 + \epsilon)$ with probability one. That is, given $\epsilon > 0$, $\exists M_1(\epsilon)$ such that $\forall m \geq M_1(\epsilon)$,

$$\hat{\sigma}_m < \sigma (1 + \epsilon).$$

(2.2)

Also from (ii) above, given $\epsilon > 0$, $\exists M_2(\epsilon)$ such that

$$\limsup_m (S_m - m\mu) \geq (\sqrt{2} - \epsilon)\sqrt{m}\sqrt{\log \log m}.$$  

Since $\mu > 0$, this means that $\exists M_2(\epsilon)$ such that $\forall m \geq M_2(\epsilon)$

$$\limsup_m S_m \geq c\sigma (1 + \epsilon)\sqrt{m}.$$  

(2.3)

From (2.2) and (2.3), we can conclude that there exists a $M(\epsilon)$ such that $\forall m \geq M(\epsilon)$,

$$\limsup_m \sum_j Z_j \geq c\hat{\sigma}_m \sqrt{m}.$$  

2.4 Numerical Results

In this section, we first show the adaptiveness of ADALINE on an inventory optimization problem, and then we illustrate the performance of ADALINE on two problems: (i) a bus scheduling problem, and (ii) a multi-dimensional quadratic minimization. We also compare the performance of ADALINE and R-SPLINE on the bus scheduling problem.

2.4.1 Inventory Optimization with ADALINE

Consider the classical periodic review $(s, S)$ inventory optimization problem with zero lead-time, no backlogging, and a fixed ordering cost (Hong and Nelson, 2006; Koenig and Law, 1985). The objective is to
determine the values of $s$, and $S$ such that the long-run expected inventory cost per review period is minimized. We assume that the demand follows a Poisson distribution with $\lambda = 10$ units/period. In addition, the following constraints are imposed on the inventory system, $20 \leq s \leq 80$, $40 \leq S \leq 100$, and $S - s \geq 10$. The optimal inventory levels for this system can be analytically determined to be $(20, 53)$.

We implement ADALINE on the inventory optimization problem where a simulation call returns the average inventory cost per period averaged over 30 time periods. To eliminate the effects of initial transient states, a warm-up time of 50 periods is considered in every simulation. Figure 2.2 shows a typical progression of ADALINE on the inventory optimization problem with a simulation budget of 5000 replications. The algorithm converges near the vicinity of true solution very quickly, in this case, within about 10 iterations. In the first two iterations, the maximum exploration occurs, with a small sample size at every visited solution, as seen in the cumulative simulation calls by the end of iteration 2. In subsequent iterations, the sample size steadily increases, and the algorithm begins transitioning into the exploitation mode, with solutions concentrating around a small neighborhood. Eventually, as the algorithm converges in the vicinity of the optimal solution, a majority of the simulation effort is expended.

2.4.2 Bus Scheduling Problem

Passengers arrive at a depot and wait until they are picked up by departing buses. The arrival of passengers follows a homogeneous Poisson process with arrival rate, $\lambda$. The objective is to schedule $d$ buses between a fixed time interval $[0, \tau]$ so that the total expected waiting time of all passengers arriving between 0 and $\tau$ is minimized. We assume the following: (i) two buses are scheduled to depart, one each at time 0 and $\tau$ respectively to pick up the rest of the waiting passengers, and (ii) the buses are assumed to have unlimited capacity, that is, all passengers currently waiting in the depot can leave in a departing bus. The decision variable is the departure time of the buses, $x = (x_1, \cdots, x_d)$, and the constraints enforced include a non-negativity constraint, and that the departure times cannot exceed $\tau$.

We compare the performance of ADALINE and R-SPLINE on the bus scheduling problem with $\lambda = 10$, $\tau = 100$, and when the number of buses scheduled equal 9 and 19, respectively. Our comparison is based on 50 independent replications of the algorithms starting with the same initial solution. At every solution, the algorithm would have access to noisy passenger-waiting times through a simulation. Since an iteration in ADALINE is not equivalent to an iteration in R-SPLINE, to facilitate fair comparison, we plot the objective corresponding to varying levels of simulation budget, $B$. For the purpose of evaluation and comparison, we utilize the analytical expression for the total expected waiting time, $g(x) = \sum_{i=1}^{d+1} \lambda \frac{(x_i-x_{i-1})^2}{2}$. For any decision, $x$, $x_{(i)}$ denotes the $i^{th}$ element corresponding to the ordered decision with $x_{(0)} = 0$ and $x_{(d+1)} = \tau$. 
Figure 2.2: The figure shows the progression of ADALINE on the \((s,S)\) inventory policy optimization. The solutions visited through iterations, and the cumulative simulation budget utilized are plotted. Earlier iterations prioritize exploration at a smaller sample size, followed by exploitation with a significant increase in sampling.

and \(d\) being the number of buses to be scheduled.

When \(\text{mod} (\tau, d+1)\) equals 0, the minimum expected waiting time for the \(d\) bus scheduling problem is \(g(x^*) = \lambda \tau^2/(2(d+1))\), and the corresponding optimal schedule is buses departing every \(\tau/(d+1)\) time-units (any of the \(d!\) permutations is an optimal solution). For example, for the 9 bus scheduling problem, the optimal solution is one of the 9! permutations of \(x^* = (10, 20, 30, 40, 50, 60, 70, 80, 90)\), and the total expected waiting time is 5000 hours.

Figures 2.3 and 2.4 show the performance of ADALINE and R-SPLINE on the 9 bus and 19 bus scheduling problems respectively. Rather than plotting the individual sample paths, we plot percentiles of
Figure 2.3: The figure shows percentiles of the expected wait time corresponding to the solution returned by ADALINE and R-SPLINE at the end of \( b \) simulation calls on the nine-bus scheduling problem.

The true objective corresponding to the random decision at a budget level based on the 50 replications. While implementing R-SPLINE, the sample growth rate that governs the sample size and the budget within each retrospective iteration is set to be a geometric series with \( r = 1.1 \). Please note that the growth rate governs two independent parameters in R-SPLINE; namely interpolation sample size, \( M^I \); neighborhood enumeration sample size, \( M^E \). These parameters are adaptively determined in ADALINE without the requirement for any user input. In Figure 2.3, for the 9 bus scheduling problem, we can clearly see that ADALINE hits a \( N^1 \) local minimum in the vicinity of the global minimum by 10,000 simulation calls. In the case of R-SPLINE, to achieve similar performance, it takes nearly twice the effort. Though ADALINE finds a local minimum quickly, R-SPLINE with its ability to oscillate between local solutions, eventually achieves a good quality solution.

For the 19 bus scheduling problem, shown in Figure 2.4, ADALINE performs significantly better than R-SPLINE indicating that fixed sample growth strategies are not efficient. Almost all replications of ADALINE reach a local minimum within 30,000 simulation calls, whereas in the case of R-SPLINE, none of the replications converge to a local minimum even after expending 100,000 simulation calls. Our conjecture is that the coordinate search phase after neighborhood enumeration helps ADALINE to outperform other...
Figure 2.4: The figure shows percentiles of the expected wait time corresponding to the solution returned by ADALINE and R-SPLINE at the end of \( b \) simulation calls on the nineteen-bus scheduling problem.

algorithms when the dimensionality increases and improve its finite time performance as seen in the 19-bus scheduling problem. We do not compare ADALINE with any other algorithm as Wang et al. (2013) establishes the superior performance of R-SPLINE over other existing algorithms.

### 2.4.3 Multi-dimensional Quadratic minimization

Consider the 30 dimensional quadratic function given by:

\[
h(x) = x_1^2 + x_2^2 + \cdots + x_{30}^2 + 1.
\]

By introducing randomness in the above function, we consider a stochastic version, \( G(x) = h(x) + \xi(x) \), where \( \xi(x) \sim N(0, 0.1h(x)) \). Our objective is to minimize \( E[G(x)] \) subject to the boundary constraint, \(-100 \leq x_i \leq 100, \forall i \in 1, \cdots, 30\). We apply ADALINE on the above problem with the initial solution, \( x^* = (80, \cdots, 80) \). Figure 2.5 illustrates the performance of ADALINE on the quadratic minimization problem, where the percentile values corresponding to 50 independent replications are plotted against the simulation calls.
2.5 Conclusion

We present ADALINE, a new algorithm for integer-ordered simulation optimization problems, where a line search and neighborhood enumeration are performed sequentially to identify a local minimum within a defined neighborhood. The algorithm relies on a linear interpolation procedure in order to perform the line search, and the neighborhood enumeration acts as a confirmatory check for a local minimum. Unlike other algorithms for integer simulation optimization, ADALINE does not require the user to provide a sampling scheme thus eliminating the need for any parameter tuning.

We are able to show that ADALINE converges to a local minimum with probability one under certain structural conditions on the objective function. Our numerical results indicate that ADALINE outperforms other SO algorithms on integer-ordered feasible sets. A working code of ADALINE on integer feasible sets is available at http://simopt.org.
Chapter 3

ADALINE for Continuous Simulation Optimization

In this chapter, we consider the unconstrained simulation optimization problem of the form:

\[ P_{CO} : \text{minimize} \quad g(x) = \mathbb{E}_\xi [G(x, \xi)] \]

subject to \( x \in X \)

where, the feasible \( X \subseteq \mathbb{R}^d \) is continuous, and the objective function can be observed only through a stochastic simulation. We assume that a Monte Carlo simulation provides a point estimate of the unknown objective function, \( \hat{g}_m(x) = m^{-1} \sum_{j=1}^m G(x, \xi_j) \), such that \( \lim_{m \to \infty} \hat{g}_m(x) = g_m(x) \), where \( m \) denotes the simulation effort. We also assume that an estimate of the gradient, \( \hat{\nabla} g_m(x) \), is constructed through “noisy” gradient observations, \( \nabla G(x, \xi_j) \).

In addition to simulation optimization, problems of type \( P_{CO} \) are frequently encountered in large scale machine learning applications, where the parameters of a prediction model are determined by minimizing a loss function using a small sample of the training data (Byrd et al., 2012). To solve such problems, either the Stochastic Approximation (SA) technique, or the Sample Average Approximation (SAA) technique discussed in Chapter 1 are frequently used. Both SA and SAA require the gradient to be estimated using a suitable procedure (either by directly observing the gradient, or through finite difference methods), and have analogs in derivative based deterministic optimization strategies. First, we discuss the fundamentals of deterministic line search methods, and extend line search techniques to develop ADALINE-Continuous (Adaptive Line
Search for Continuous Simulation Optimization for solving Problem $P_{CO}$.

### 3.1 Deterministic Line Search Methods

Deterministic line search algorithms are iterative techniques that seek to identify a local minimum on continuous feasible sets. Given an initial solution, $x_0$, the algorithm identifies a sequence of iterates, $\{x_k\}$ using the recursive relation:

$$x_{k+1} := x_k + s_k d_k,$$

where $d_k$ is the direction of descent, and $s_k$ is the step length along the descent direction. The descent direction is chosen such that it ensures a decrease in objective function for a suitable step length. Thus, a line search method is a two step procedure, where the first step is to identify the direction of descent, $d_k$, followed by determining the step length along that direction.

To build more intuition about line search methods, consider the second order Taylor’s expansion of the function, $g$ at $x_{k+1}$:

$$g(x_k + s_k d_k) = g(x_k) + s_k d_k^T \nabla g(x_k) + \frac{1}{2} s_k^2 d_k^T \nabla^2 g(x_k) d_k$$  \hspace{1cm} (3.1)

From (3.1), it should be noted that the condition, $d_k^T \nabla g(x_k) < 0$ should hold for the algorithm to guarantee a decrease in objective function at the new iterate, $x_{k+1}$. Writing $d_k^T \nabla g(x_k)$ as a dot product, $||d_k|| \cdot ||\nabla g(x_k)|| \cos \theta$, the condition can be further simplified as $\cos \theta < 0$. For the descent direction to guarantee a decrease in objective function, it should be of the form $d_k = -B_k^{-1} \nabla g(x_k)$, where $B_k^{-1}$ is a positive-definite matrix, choices of which lead to many variations in line search methods as seen later.

Once the direction of descent, $d_k$, is determined, the optimal step length is the minimizer of $\min_s g(x_k + sd_k)$. In practice, finding an exact step length at each iteration is computationally intensive, and hence, the step length is usually determined by an inexact search method. An inexact line search is an iterative process to determine the step length such that the resulting iterate satisfies one or more “sufficient decrease” conditions. Two of the commonly used conditions for determining step length are the Armijo-Goldstein conditions (Armijo et al., 1966; Nocedal and Wright, 2006), which guarantees a step length that minimizes the functional value to at least a fraction attained by a first order Taylor’s expansion while preventing the step length from becoming too small (Nocedal and Wright, 2006).

$$g(x_k + s_k d_k) \leq g(x_k) + \eta s_k d_k^T \nabla g(x_k)$$  \hspace{1cm} (3.2)

$$g(x_k + s_k d_k) \geq g(x_k) + (1 - \eta) s_k d_k^T \nabla g(x_k)$$  \hspace{1cm} (3.3)
The choice of the descent direction, particularly the choice of $B_k$ leads to different variations of line search methods. When $B_k$ is the Hessian matrix and a unit step length is used, it is the well-known Newton’s method, whereas if $B_k$ is the identity matrix, it is the gradient or the steepest descent method. However, Newton’s method has several disadvantages including the expensive computational effort in calculating the Hessian, and the steepest descent method suffers from inferior asymptotic rate of convergence. Quasi-Newton methods overcome these limitations by approximating the Hessian at each iteration using the gradient from previous iterate. Depending on the mechanism for updating $B_k$, many Quasi-Newton algorithms can be designed, and for a thorough treatment of these methods, and other line search algorithms, we refer readers to Nocedal and Wright (2006).

### 3.2 Variable Sampling in Simulation Optimization

From Chapter 1, recall that SA, the analogue of deterministic line search in SO, uses a Newton-type recursion with a fixed step length, and that the SAA can utilize the same algorithms with no modifications. However, these methods are sensitive to the estimation error of the gradient, and the objective function respectively (Kushner and Clark, 2012). Moreover, the convergence and finite time performance of these algorithms are highly dependent on the user-specified choice of sample size and the step size (Powell, 2007), often referred to as the learning rate in machine learning applications.

A poor choice of step length or a large sample size introduces inefficiency, whereas insufficient sampling leads to non-convergence. Specifically, the sampling scheme, which primarily governs the quality of the estimated objective function and the gradient (the direction of descent) is of particular importance. Therefore, our focus will be on developing a gradient based line search technique that adaptively determines the sample size at every iteration. As is the case with other gradient based methods, our algorithm will seek to determine a local minimum, $x^*$, of Problem $P_{CO}$. Recall that, by definition, $x^* \in \mathbb{R}^d$ is a local minimum if and only if any other feasible solution within an $\epsilon$-neighborhood of $x^*$ is inferior to $x^*$.

In SO, several variable sampling schemes have been proposed to work either within a SA or SAA framework. Recall that, in the SAA procedure, the true objective function in Problem $P_{CO}$ is approximated with the sample average, $\hat{g}_m(x) = m^{-1} \sum_{j=1}^m G(x, \xi_j)$. Then, a sequence of approximate problems are solved using any deterministic optimization procedure, and for a large enough sample size, SAA converges to an optimal solution. Since SAA requires the approximate problem to be solved repeatedly starting with the same initial solution, to conserve simulation effort, Homem-De-Mello (2003) proposed a variable sampling scheme where the sample size, $m_k$, in iteration $k$ increases at a certain rate such that $m_k \rightarrow \infty$. However, the sampling scheme is fixed, and non-decreasing irrespective of the problem.
Other refinements to SAA have been proposed that incorporate variable sampling, including the Retrospective Approximation (RA) procedure, where a sequence of sample path problems with increasing sample sizes are solved to adequate tolerance (Pasupathy, 2010). However, the sampling scheme in RA is still fixed, oblivious of the solution quality in any iteration. To overcome these issues in sample path optimization, Deng and Ferris (2009) propose a Bayesian dynamic sampling scheme that determines the sample size based on the quality (uncertainty) of the obtained solution.

Notice that, in SAA, a numerical optimization procedure is utilized at every iteration. Being iterative methods, these deterministic techniques result in a sequence of solutions visited in an iteration, where the functional value is evaluated with a fixed sample size, \( m_k \). This renders SAA to be computationally expensive procedure for solving Problem \( P_{CO} \) (Deng and Ferris, 2009). On the other hand, the unconstrained nature of Problem \( P_{CO} \) presents itself conducive to be solved by within an iterative framework with varying sample sizes at each iteration. We explore such variable sampling strategies further in this chapter.

Dynamic sampling methods that have been recently developed to work within a SA framework (Byrd et al., 2012; Krejić and Krklec, 2013; Pasupathy and Schmeiser, 2010) have addressed this issue by determining the sample size as a trade-off between the two errors, either by following a non-decreasing sampling strategy or a fully dynamic strategy. Byrd et al. (2012) proposes a non-decreasing sampling strategy where the sample size at each iteration is determined at the end of the previous iteration by balancing the norm of the estimated gradient and its variance. In Krejić and Krklec (2013), the sample size at an iteration is chosen using a fully dynamic sampling strategy that balances the estimation error in the objective with the decrease in objective function value. However, some of the disadvantages of these methods include the sample size being fixed at the previous iterate (a systemic sampling scheme), fixed maximum sampling effort as in Krejić and Krklec (2013) that may affect convergence, and strong conditions on sample growth rate.

A sequential sampling strategy where the sample size at any iteration is determined by balancing the errors at the current iterate and that has no fixed upper bound on the sample size will help overcome some of these issues. We build upon sequential sampling schemes described in Hashemi et al. (2014); Pasupathy and Schmeiser (2010) and use existing deterministic line search methods to ensure that the new simulation optimization algorithm converges to a local minimum. In the subsequent sections, we present the algorithm and a numerical study to illustrate the performance of the algorithm.
3.3 ADALINE - Continuous Simulation Optimization

The paradigm of ADALINE on continuous feasible sets is similar to that of deterministic line search methods, where a direction of descent is identified first, followed by a search along that direction. However, when “noisy” gradients are observed, to estimate the direction of descent with sufficient confidence, the first phase of our algorithm determines the sample size adaptively based on the incumbent solution quality. The following steps are performed in one iteration of our algorithm:

S.1 An adaptive sampling criterion is employed to determine the sample size, and the gradient is estimated at each iterate;

S.2 A line search along the negative gradient direction identified in S.1 is employed to identify a better solution.

As discussed earlier, the sampling scheme we propose allocates simulation effort directly proportional to the quality of the visited solution. This is achieved by letting the sample size be determined as a trade-off between the estimation error and the optimization error. By letting the sample size depend only on the solution quality in a sequential sampling framework, our algorithm also eliminates the need for other user-defined parameters. To provide some perspective, our adaptive sampling strategy relies on one user-specified parameter to determine the sample size at any iteration, whereas, a comparable sampling strategy developed in Krejić and Krklec (2013) requires four user-specified parameters just to determine the sample size. Once the sample size is determined, a line search is performed in S.2 to identify a better solution.

In our algorithm, we let $X_k$ denote the $k^{th}$ visited solution, and $M_k$ the sample size in the $k^{th}$ iteration. The minimum sample size at any iteration is denoted by $M_{\text{min}}$. The estimated gradient, $\hat{\nabla}g(X_k)$, and the sample variance, $\hat{\sigma}^2_{N_k}(\hat{\nabla}(g(X_k)))$ form the basis for the optimization error, and the estimation error respectively.

3.3.1 Details of the algorithm & pseudocode

Recall that the line search method uses a recursive equation of the form, $X_{k+1} = X_k + s_k d_k$. The search direction, $d_k$, is a function of the estimated gradient, and is of the form, $d_k = -B_k^{-1} \hat{\nabla}g(X_k)$. Based on the structure of the matrix, $B_k$, the line search method is classified as follows: (i) Stochastic Gradient Descent, where $B_k$ is the identity matrix, $I$; and (ii) Stochastic quasi-Newton method (BFGS method), where $B_k$ is an approximation of the Hessian that approaches the true Hessian as the algorithm progresses (Nocedal
In both of the methods, once a directional estimate is available, an inexact line search is performed to determine the step length, \( s_k \) along the direction \( d_k \) using the Armijo-Goldstein conditions (3.2) and (3.3). Note that the above algorithm is a stochastic counterpart to deterministic line search methods with the exception that the true gradient/functional values are replaced with their corresponding sample mean. In the proposed strategy, the sampling effort, \( M_k \), at any iteration is a trade-off between a linear transformation of the sample covariance matrix that denotes the estimation error, \( \hat{\sigma}_n(\nabla g(X_k)) \), and the optimization error given by the estimated norm of the gradient, \( ||\nabla g_{M_k}(X_k)|| \) (Hashemi et al., 2014)

\[
M_k = \inf_{m > [kr]} \left\{ m : \frac{\kappa(\alpha, M_k) \times \hat{\sigma}_m(\nabla g(X_k))}{\sqrt{m}} \leq ||\nabla g_m(X_k)|| \right\} \quad \text{and} \quad r > 1.
\]

The constant, \( \kappa(\alpha, m) \), can be chosen to be the \( \alpha^{th} \) percentile of a Student’s \( t \) distribution with \( m - 1 \) degrees of freedom. Following the results of Pasupathy et al. (2015), in order to ensure almost sure convergence, an escort sequence is used to establish a sufficient growth in the minimum sampling effort at each iteration. Since an inexact line search estimates the function at the new iterate, \( \hat{g}_{M_k}(X_{k+1}) \), by imposing a simple condition on \( M_{k+1} \), we obtain two possible algorithmic variation as follows: (i) a dynamic sampling strategy that just satisfies the condition (3.4) at each iteration, and (ii) a non-decreasing sampling strategy where \( M_k \leq M_{k+1} \). In the first strategy, it should be noted that reduction in sample size is possible between iterations making it a completely adaptive sampling scheme, whereas in the second strategy, such reduction is not permitted, and the algorithm only signals when the sample size should be increased.

### 3.4 Numerical Results

We implement our adaptive sampling line search algorithm on the stochastic counterpart of the following well known deterministic functions: (i) The Rosenbrock function (Deng and Ferris, 2009), and (ii) the Aluffi-Pentini function (Aluffi Pentini et al., 1985), each with two decision variables. A stochastic counterpart is generated by multiplying one of the decision variables in each of the above functions with a random variable with mean 1. In order to illustrate the advantages of our sampling strategy, we also compare our results with an existing adaptive sampling strategy proposed by Krejić and Krklec (2013).

Despite the sampling strategy in our proposed algorithm being adaptive, that is, automatically signal an increase or decrease in sample sizes between iterations, we can devise a variation which restricts the sample sizes to be non-decreasing between iterations. Moreover, the choice of the descent direction in the line search technique results in two other variations of our algorithm: (i) a stochastic gradient descent (SGD) method, and (ii) a stochastic BFGS method (Quasi-Newton). Thus, the four possible variations of
Algorithm 4 Adaptive Sampling for Continuous Simulation Optimization

Require: Total budget $B$; Starting solution $x_0 \in \mathbb{R}^d$; Initial sample size $M_{\min}$; Sample growth rate $r > 1$; Backtracking parameter $\beta$; Armijo parameter $\eta$; Confidence level $\alpha$

Ensure: Best Solution $X_{\text{best}}$; Estimated function value $\hat{g}_{M_k}(X_{\text{best}})$; Estimated gradient $\hat{\nabla}g(X_{\text{best}})$

1: Set $k \leftarrow 0$; $X_k \leftarrow x_0$; $\text{ncalls} \leftarrow 0$
2: repeat {Adaptive Sampling Line Search}
3:   $M_k \leftarrow \max\{\lceil k^r \rceil, M_{\min}\}$
4:   Observe $G(X_k, \xi_j)$, $\nabla G(X_k, \xi_j)$, where $j = 1, \cdots, M_k$.
5:   Estimate $\hat{g}_{M_k}(X_k) = M_k^{-1} \sum_{j=1}^{M_k} G(X_k, \xi_j)$, $\hat{\nabla}g(X_k) = M_k^{-1} \sum_{j=1}^{M_k} \nabla G(X_k, \xi_j)$
6:   Estimate $\hat{\sigma}_{M_k}(\hat{\nabla}g(X_k)) = \sqrt{\sum_{i=1}^{d} \hat{\sigma}^2(X_k^i)}$
7:   Set $\text{ncalls} \leftarrow \text{ncalls} + M_k$
8:   while $\kappa(\alpha, M_k) \times \frac{\hat{\sigma}_{M_k}(\hat{\nabla}g(X_k))}{\sqrt{M_k}} > \|\hat{\nabla}g(X_k)\|$ do
9:     $M_k \leftarrow M_k + 1$; $\text{ncalls} \leftarrow \text{ncalls} + 1$
10:    Update $\hat{g}_{M_k}(X_k)$, $\hat{\nabla}g(X_k)$, $\hat{\sigma}_{M_k}(\hat{\nabla}g(X_k))$
11: end while
12: Set $d_k \leftarrow -B_k^{-1}\hat{\nabla}g(X_k)$ \quad $B_k^{-1}$ is set to be $I$ in gradient descent or an approx. of the Hessian Inverse $(H_k^{-1})$ in the BFGS method.
13: Determine $s_k$ using inexact line search using Goldstein’s condition with parameter $\beta$ and $0 < \eta < 1/2$ such that
14: \[
\hat{g}(X_k + s_k d_k) \leq \hat{g}(X_k) + \eta s_k d_k^T \hat{\nabla}g(X_k) \& \\
\hat{g}(X_k + s_k d_k) \geq \hat{g}(X_k) + (1 - \eta) s_k d_k^T \hat{\nabla}g(X_k)
\]
15: Set $X_{k+1} \leftarrow X_k + s_k d_k$
16: $k \leftarrow k + 1$
17: Update $\text{ncalls}$
18: until $\text{ncalls} > B$
19: return $X_{\text{best}}$; $\hat{g}_{M_k}(X_{\text{best}})$; $\hat{\nabla}g(X_{\text{best}})$
the algorithm based on the sampling strategy, and the line search technique are as follows: (i) SGD with adaptive sampling, (ii) SGD with non-decreasing sampling, (iii) SBFGS with adaptive sampling, and (iv) SBFGS with non-decreasing sampling.

Recall that, in the stochastic BFGS update, an approximation of the Hessian matrix is generated using the estimated gradient, and the step size. In order to avoid frequent noisy updates to the Hessian approximation, particularly at the initial iterations of the algorithm, we follow the work of Byrd et al. (2014), and update the approximation of the Hessian matrix only at every $L^{th}$ iteration. As discussed earlier, an escort sequence is used to establish a sufficient growth in the sample size between iterations, that is, to provide a lower bound on the sample size every iteration. In our experiments, we set the escort sequence as a polynomial function, $k^r$, where $k$ denotes the iteration number, and $r$ is a constant.

We run multiple independent replications of the algorithm starting with the same initial solution. Since the algorithm is designed to expend infinite simulation effort on convergence to a local minimum, we terminate the algorithm on exceeding the simulation budget, $B$. A call to the simulation oracle, in addition to providing a realization of the objective function, $G(x, \xi)$, also provides a noisy observation of the gradient, $\nabla G(x, \xi)$. The following parameters govern the sample size, and the line search during the optimization:

<table>
<thead>
<tr>
<th>Algorithm Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total budget ($B$)   : $3 \times 10^5$</td>
</tr>
<tr>
<td>Initial sample size ($M_{\text{min}}$) : 2</td>
</tr>
<tr>
<td>Sample growth rate ($r$) : 1.1</td>
</tr>
<tr>
<td>Backtracking parameter ($\beta$) : $1/2$</td>
</tr>
<tr>
<td>Armijo parameter ($\eta$) : $1 \times 10^{-4}$</td>
</tr>
</tbody>
</table>

### 3.4.1 Aluffi-Pentini Function

The deterministic Aluffi-Pentini function in two dimensions is converted to its stochastic counterpart by multiplying a random component to the first term, such that, $x = (x_1\xi, x_2)$. We minimize the expectation of the resulting stochastic function giving rise to the following unconstrained optimization problem:

$$
\min_{x} g(x) = \mathbb{E}_{\xi}[0.25(x_1\xi)^4 - 0.5(x_1\xi)^2 + 0.1(x_1\xi) + 0.5x_2^2]
$$

The random variable, $\xi$, is assumed to follow a normal distribution with mean, $\mu$, equal to 1, and variance, $\sigma^2$, equal to 0.1. As an analytical form for the function is available, we determine the set of all local minima
as:

\[ x^* = \{ (-0.863, 0), (0.772, 0), (0.093, 0) \}. \]

On implementation, the algorithm may converge to one of the local minimum, that is, converge to a point where \( |\nabla g(x)| = 0 \). The initial solution for each independent replication is set to be far away from the set of all local minima, in this case, \( x_0 = (18, 25) \).

![Figure 3.1: The figure shows percentiles of the norm of the true gradient corresponding to the solutions returned by all variations of our algorithm on the Aluffi-Pentini problem](image)

In Figure 3.1, we plot the percentile values of the true gradient corresponding to the solution obtained at various budget levels with our adaptive sampling strategy. As seen in the figure, the percentile values of the true gradient at a budget level of 10,000 for the adaptive sampling strategy is better than that of the non-decreasing sampling strategy suggesting that the adaptive sampling strategy converges in the vicinity of
a local minimum more quickly than the non-decreasing sampling scheme. However, both of these sampling schemes perform similar with further increase in the total simulation budget.

Figure 3.2: Performance of the adaptive samplings strategy on the Aluffi-Pentini problem in comparison with the sampling strategy in Krejić and Krklec (2013)

In Figure 3.2, we compare the performance of our algorithm with the adaptive sampling strategy proposed in (Krejić and Krklec, 2013), with a Quasi-Newton update, and a maximum sample size, $M_{\text{max}} = 1500$. In this figure, we plot the product of the square root of the cumulative simulation effort and the norm of true gradient averaged over several independent replications across the iterations. Since, we expect the algorithm to converge to a stationary point, the closer the product is to zero, the better the quality of the obtained solution by the algorithm. As it can be clearly seen, our algorithm converges to an iterate that is at least four times better within about twenty five iterations, whereas the adaptive strategy in (Krejić and Krklec, 2013) does not provide a better solution despite expending the entire simulation budget.

Figure 3.3 illustrates the sample size across all independent replications as a function of the iteration for the Aluffi-Pentini problem. At the initial stages of the algorithm, the impact of the escort sequence is significant for at least 50% of the replications, and as the algorithm progresses, the sampling criterion in (3.4) takes over in determining the sample size. However, for the non-decreasing sampling strategy, the escort sequence does not seem to show any significant role in sample size determination.
3.4.2 Rosenbrock Function

We obtain the stochastic counterpart of the Rosenbrock function in two dimensions by multiplying the first decision variable, $x_1$ with a random variable, $\xi$. We assume that $\xi$ follows a normal distribution with mean, $\mu$, equal to 1, and variance, $\sigma^2$, equal to 0.1. By taking an expectation with respect to $\xi$, we get the following unconstrained optimization problem:

$$
\min_{x} g(x) = \mathbb{E}_\xi [100(x_2 - (x_1 \xi)^2)^2 + (x_1 \xi - 1)^2]
$$

The local optima for this function is $x^* = (0.209, 0.048)$. Since, optimization without curvature information results in non-convergence, following (Krejić and Krklec, 2013), we only apply the stochastic BFGS technique on the problem. Figure 3.4 shows the norm of the true gradient at various budget levels while executing SBFGS method with dynamic sampling on the Rosenbrock function starting with an initial iterate, $X_0 = (5, 8)$. 

Figure 3.3: Sample growth in the Aluffi-Pentini problem for the Stochastic BFGS method
Figure 3.4: The figure shows percentiles of the norm of the true gradient corresponding to the solutions returned by all variations of our algorithm on the Rosenbrock problem.

As in the case of the Aluffi-Pentini function, our sampling strategy yields better results when compared to the strategy provided in (Krejić and Krklec, 2013), as shown in Figure 3.5.

Figure 3.5: Performance of the adaptive samplings strategy on the Rosenbrock problem in comparison with the sampling strategy in Krejić and Krklec (2013).
Figure 3.6 illustrates the sample growth for the Rosenbrock problem. Unlike the Aluffi Pentini problem, the reliance on the escort sequence is very significant in the middle phases of the algorithm with perfectly adaptive sampling strategy. For the non-decreasing sampling strategy, the escort sequence does not seem to play any role in sample size determination.

![Graph showing sample growth for Stochastic BFGS methods with adaptive and non-decreasing sampling strategies.]

Figure 3.6: Sample growth in the Rosenbrock problem for the Stochastic BFGS method

3.5 Conclusion

In this Chapter, we presented an adaptive sampling algorithm for continuous simulation optimization problems that determines the sample size as a trade-off between the estimation error, and the optimization error. We have improved upon one of the crucial drawbacks of current stochastic line search algorithms by eliminating the need for user defined parameters, and their tuning. We also illustrate the superior performance of the algorithm by comparing it with the adaptive sampling strategy proposed in (Krejić and Krklec, 2013).
Chapter 4

Surgery Rescheduling using Simulation Optimization

In this chapter, we demonstrate the use and superiority of ADALINE over a competing algorithm on a (quasi-realistic) surgery scheduling problem. The total costs incurred due to assigning and performing surgeries in an operating room (OR) account for a large percentage of a hospital's operating expenses, in some cases up to 50% of the total operating cost (Elixhauser and Andrews, 2010). Since the number of surgeries performed varies significantly between specialties, optimal allocation of capacity and efficient scheduling practices are critical to improving OR financial performance. Hospitals typically allocate OR capacity to specialties based on a variety of factors including the historical volume, priority level of the specialty, as well as to incentivize surgeons for increasing the overall utilization of the operating rooms (OR) (Hosseini and Taafe, 2014).

Surgical specialties compete, not just for the operating rooms, but also for other resources including equipment and technicians, necessitating the need for a scheduling policy that streamlines the allocation of these resources. Therefore, a weekly scheduling policy, called “Block Scheduling” is usually utilized as a framework where surgeries with similar resource needs are grouped together and then assigned to specific time blocks in the future (Blake et al., 2002). Block scheduling improves surgeon satisfaction and leads to better utilization of operating rooms while minimizing the total overhead cost.

In addition to performing elective surgeries that are scheduled in advance, hospitals have to accommodate non-elective surgeries that require expedited surgical intervention. These pose significant challenges to the scheduler due to the uncertainty in their arrival times and surgical resource needs. Also, the actual versus planned duration of surgeries vary significantly and often lead to schedule disruptions. These dis-
ruptions add to the operating cost of a hospital either by forcing the scheduler to reschedule/cancel elective procedures or accommodate the non-elective surgeries by assigning overtime work.

Every day schedulers have to decide on protocols to accommodate elective and non-elective cases under a capacity constrained operating room environment. A sub-optimal rescheduling decision leads to financial losses due to high overtime costs, postponement costs and cancellation costs (Gul et al., 2015). Thus, a sophisticated rescheduling policy is required to aid the scheduler in making the following decisions: (i) the number of elective cases that should be postponed, (ii) the policy for scheduling non-elective cases that are realized on any given day, and (iii) the overtime hours needed to accommodate all cases planned for that day. A successful scheduling policy is also one that fits into the block scheduling framework without perturbing the surgical blocks too much.

Non-elective surgeries, one of the primary causes of uncertainty in the operating room environment, can further be classified into urgent and semi-urgent surgeries. Urgent surgeries (emergency) require immediate surgical intervention and semi-urgent surgeries require prioritized intervention within a specified time period. Prior work on urgent surgeries concludes that dedicating specific operating rooms minimizes disruption to the existing schedule (Heng and Wright, 2013). Therefore, we restrict the scope of our work to only accommodating elective surgeries and semi-urgent surgeries, particularly on the development and optimization of a rescheduling policy for operating room resource allocation.

The remainder of this work is organized as follows: in Section 4.1, we discuss relevant work in the literature on surgical scheduling; in Section 4.2, we introduce the problem statement and develop a mathematical model for finding an optimal solution; in Section 4.3, we discuss characteristics of optimal solution to the true problem, its advantages and disadvantages, and how the disadvantages can be overcome using a threshold-based scheduling policy, and describe the heuristic in detail; in Section 4.4, we present an optimization and simulation optimization based framework for optimizing the threshold levels; and finally in Section 4.5, we detail the performance of the simulation optimization (SO) framework and the integer optimization on a synthetic data set, followed by concluding remarks.

4.1 Literature review

Surgery scheduling is an extensively studied problem with prior work categorized into the following three groups based on the time horizon of decision making: (i) operating room capacity planning (strategic decisions), (ii) block scheduling, that is, allocation of surgical capacity to specialties (tactical decisions), and finally (iii) allocation of individual surgeries based on the block schedule to operating rooms (operational
decisions). A majority of the literature on OR capacity planning and surgical block allocation utilize an integer programming framework, usually with an objective to increase the long term utilization of operating rooms (Blake and Donald, 2002; Day et al., 2012; Hosseini and Taaffe, 2014). While these models account for the variability in utilization of operating rooms, they do not explicitly consider the impact of scheduling practices that induce such variability.

To overcome the disadvantages associated with conventional models, Zhang et al. (2009) propose an integrated framework for capacity allocation and surgery scheduling. The model allocates surgical time between elective and urgent surgeries with the objective of minimizing the length of stay after a surgical procedure. However, by failing to account for the uncertainty and replacing the random variables with their expected value, the solution from the model is rendered inferior. Min and Yih (2010) propose a model for the scheduling and assignment of elective cases when the duration of surgeries are random, and under constrained operating room resources. In this case, the assignment of surgeries, and allocation of other resources is guided by a two stage stochastic programming approach, solved using sample average approximation (SAA)—an approach where the random variables are replaced by their realizations, and solved using deterministic techniques. If not for the allocation of secondary resources, this work is equivalent to an elective surgery assignment problem with random surgery durations, where the objective is to minimize the over time cost.

Lamiri et al. (2009) propose a stochastic model for assigning elective cases across the planning horizon so that the overall cost of accommodating urgent surgeries is minimized. Unlike Zhang et al. (2009), the random variables in the model are replaced by Monte Carlo realizations resulting in an approximate problem that is solved using deterministic techniques. Surgeries that exceed the available capacity are accommodated using overtime, whereas, elective surgeries may often be rescheduled under such circumstances. By failing to account for the dynamic nature of the decision process, the utility of this model is primarily restricted to developing an initial schedule for electives.

Herring and Herrmann (2012) study the single day surgery scheduling problem, where a policy is developed for assigning surgeries to a particular day in the future. By either accommodating the surgeries immediately or deferring decisions to the future, schedulers can optimize the utilization of surgical blocks on a dynamic basis until the day of the surgery. However, even for smaller surgical blocks, this problem is intractable due to the “curse of dimensionality” in their proposed dynamic programming framework. Therefore, they propose an optimization based threshold policy that differentiates between competing specialties to occupy a surgical block. Note that the similar threshold based approach is applicable for the surgery rescheduling problem, for we study the allocation of surgical blocks between elective and urgent surgeries rather than competing specialties.
Gul et al. (2015) develop a comprehensive optimization framework for dynamic scheduling of elective surgeries across a planning horizon. Though they do not differentiate between elective, or urgent surgeries, their optimization framework is flexible enough to accommodate such modifications. On a dynamic basis, surgeries are scheduled and repetitively rescheduled so that the overall cost of performing surgeries is minimized. Similar to the work of Lamiri et al. (2009), this model utilizes a Monte Carlo simulation based optimization approach to represent the uncertainty in the demand, and duration of surgeries. Though the solution from this model is optimal and provides critical insights on allocating surgical capacity to develop block schedules, as we later discuss, these solutions are not implementable in real time.

The capacity allocation model developed by Zonderland et al. (2010) is the only work in the surgery scheduling literature that considers semi-urgent surgeries. Theirs is a queuing model that can be formulated in a Markov decision process (MDP) framework to optimize the capacity reserved for semi-urgent surgeries every week. However, they assume that all semi-urgent surgeries are assigned on a weekly basis, and are realized at the beginning of the week according to a Poisson process. Though these assumptions yield an analytical expression, they render the model unrealistic, for the OR scheduler often makes adjustments to the schedule based on individual surgeries, and on a daily basis.

Our problem is very similar to those studied in Gul et al. (2015) and Zonderland et al. (2010). By considering the preliminary schedule for electives as a model parameter, we differ from Gul et al. (2015) where electives are also scheduled. Note that the preliminary schedule in our model can be the optimal solution to the static surgery assignment models from Lamiri et al. (2009); Min and Yih (2010); Zhang et al. (2009). At the same time, we overcome the drawbacks associated with Zonderland et al. (2010) by letting semi-urgent surgeries to be scheduled everyday, and by making no assumptions on the distribution of the random variables that govern the uncertainty. Thus, we believe our work fits seamlessly into existing scheduling practices, for it provides an implementable policy on accommodating semi-urgent surgeries.

4.2 Problem Statement

We consider the problem of accommodating elective and semi-urgent surgeries in a hospital, where the electives are scheduled in advance and the semi-urgent surgeries have an unknown demand. In general, elective surgeries of various surgical specialties are assigned at least a week prior to the surgery, based on a block schedule. The demand for semi-urgent surgeries can only be realized a day or two prior to the surgery, making it a challenge for the scheduler to accommodate them, given the limitations on available operating room time. Thus, the objective is to develop a scheduling policy that accommodates elective and semi-urgent surgeries with minimal disruption to the existing schedule, and that reduces the cost of adding unplanned
A detailed description of the problem is as follows: At the start of a week, information on the exact number of elective surgeries over the entire coming week is known to the OR scheduler. The capacity allocated to every surgical specialty is also fixed based on the block schedule. Before executing the schedule on any particular day, requests for semi-urgent surgeries are realized at the start of the day. Upon realization of semi-urgent surgical requests, the OR scheduler has to accommodate them so that the overall cost of disrupting the existing schedule is minimized. If sufficient slack in the schedule is available, semi-urgent surgeries may be accommodated without any disruption to the original schedule. In the event of not having any additional capacity in the existing schedule, semi-urgent surgeries have to be accommodated by one of the following means: (i) postponing currently scheduled elective surgeries, (ii) cancellation of elective surgical cases, (iii) accommodating semi-urgent surgeries with overtime.

Each of the above three decisions incur an additional cost that increases the operating cost of the hospital. For example, a decision to postpone an elective surgery is followed by another decision on fixing the day it should be performed. In general, when the requests for semi-urgent surgeries are realized on any particular day, the following sequence of decisions are made by the OR scheduler:

(i) A schedule for accommodating semi-urgent surgeries using available capacity.

(ii) A new schedule after postponement and cancellation of elective surgeries that were previously scheduled for the day.

(iii) Inclusion of additional capacity via overtime, if any, when the original capacity allocated based on the block schedule is limited.

We assume the following about the semi-urgent surgeries: (i) the requests are made by a surgical specialty at the start of day; and (ii) the semi-urgent cases should be accommodated within 3 days (including the day of arrival), failing which, a loss in revenue is incurred by the hospital. Despite the above relaxing assumptions, the problem is difficult to solve because of the uncertainty in the number of semi-urgent surgeries requested, and the requirement to make such decisions on a repetitive basis. Thus, a decision framework to solve the surgery rescheduling problem should incorporate the following features, (i) a mechanism for generating realizations from the probability distribution of the demand and duration of surgeries, and (ii) an optimization model that minimizes the total cost of rescheduling for a realization of these random variables. In the remainder of this section, we discuss the random variables followed by the optimization model that makes rescheduling decisions for a realization of the random variables across the entire time horizon.
4.2.1 Random Variables

The two sources of randomness in the surgery rescheduling problem are: (i) the number of semi-urgent surgeries requested for each specialty at the beginning of the day, and (ii) the duration of all surgeries that are scheduled for a particular day. In this section, we first introduce the notations and then define these two random variables:

**Parameters**

- \( n \) : number of specialties
- \( h \) : length of the planning horizon

**Sets and Indices**

- \( i \) : index for specialty type, \( i \in \{1, \ldots, n\} \)
- \( l, k, q, t \) : index for time period, \( t \in \{1, \ldots, h\} \)
- \( \mathcal{E}_{it} \) : set of all elective surgeries of specialty \( i \) scheduled for day \( t \)

Now, concerning the randomness in surgical request, let \( U_{it} \sim F_{U_{it}}(\cdot) \) be the random variable that denotes the number of semi-urgent surgeries of specialty \( i \) requested on day \( t \), and a realization of the random variable be \( u_{it} \). Let \( U_i = (U_{1t}, \ldots, U_{nt}) \) denote the collection of these random variables across all specialties on day \( t \), and a realization of \( U_i \) is \( u_i \). Figure 4.1 shows \( m \) independent realizations of the number of semi-urgent cases across the planning horizon of length \( h \), where each realization is indexed by the superscript, \( s \).

![Figure 4.1: Realization of Random Variables that represent the requests for Semi-urgent surgeries](image-url)

Let \( D_i \sim F_{D_i}(\cdot) \) be the random variable that denotes the duration of any surgery in specialty \( i \). Let \( e_{it} \) denote the number of elective surgeries of specialty \( i \) on day \( t \), that is, \( e_{it} = |\mathcal{E}_{it}| \), and \( e_t \) denote the vector of elective surgeries scheduled for day \( t \). A realization of the duration of elective surgeries consists of \( e_{it} \) independent and identically distributed copies of the random variable, \( D_i \). Similarly, for semi-urgent surgeries, \( u_{it} \) i.i.d
random variables following the distribution, $F_{D_i}(\cdot)$, are generated on any particular day.

The set of all surgical durations realized on day $t$ be denoted by $D_t$. $D_t$ has two elements, each in-turn is a set corresponding to the duration of semi-urgent and elective surgeries, respectively. Further, each one of these sets is a collection of random vectors of surgery durations across multiple specialties. For example, if $u_t = (u_{1t}, u_{2t})$, the duration of these surgeries is a set with two elements, where each element is a vector of length $u_{1t}$ and $u_{2t}$ respectively. A realization of the number of semi-urgent surgeries and their duration at any time period $t$ across all surgical specialties is $S_t = \{u_t, D_t\}$. Similarly, we denote a realization across the entire time horizon by the set, $S = \{\{u_t\}_{\text{1,}} \leq t \leq h}, \{D_t\}_{\text{1,}} \leq t \leq h\}$.

At the start of the day, when the surgical requests are realized, the scheduler has to make decisions so that the rescheduling cost is minimized. Note that, for stochastic optimization, explicitly incorporating the probability distribution of the random variables, and analytically determining an optimal solution requires painstaking effort. Likewise, solving such problems using classical optimization techniques may even be impossible, particularly when the uncertain variables have an infinite support, as in continuous probability distributions.

Often, to overcome these issues, a finite number of realizations of the random variables are used to construct a discrete approximation, and the resulting deterministic equivalent of the problem is solved using existing optimization techniques. This technique is known as the Sample Average Approximation (SAA) method or the scenario based optimization method. In the following section, we formulate the surgery rescheduling problem as a deterministic problem where the realizations of the random variables become input parameters to the mathematical model. For convenience, we denote the set of all such realizations of these random variables used for SAA by $S$, with every individual realization indexed by $s$, $\forall s \in \{1, \cdots, |S|\}$.

### 4.2.2 Surgery Rescheduling using SAA

For the rescheduling problem, decisions are made sequentially everyday with little or no information about the future. Similar to the other surgery scheduling problems studied in the literature (Gul et al., 2015; Lamiri et al., 2009), we consider a multi-stage optimization framework where the objective is to minimize total expected cost of rescheduling across the entire planning horizon. Particularly, we extend the optimization framework studied in Gul et al. (2015) for formulating the surgery rescheduling problem, with additional decision variables for semi-urgent surgeries. Before we detail the formulation, we list all notations used in the optimization model:
Sets and Indices

- \( S \): set of all realizations of the random variables across the planning horizon
- \( s \): index for realization, \( s \in \{1, \cdots, |S|\} \)
- \( a, b \): indices for elective and semi-urgent surgeries

Parameters

- \( v_{it} \): capacity allocated to specialty \( i \) on day \( t \) (in number of surgeries)
- \( d_{i}^{e} \): average duration (in minutes) of an elective surgery for specialty \( i \)
- \( d_{i}^{u} \): average duration (in minutes) of a semi-urgent surgery for specialty \( i \)

Cost

- \( c_{i}^{r} \): cost of regular time per hour for specialty \( i \)
- \( c_{i}^{o} \): cost of overtime per hour for specialty \( i \)
- \( c_{i}^{a} \): cost of additional capacity per hour for specialty \( i \)
- \( c_{i}^{p} \): cost of postponement per day for specialty \( i \)
- \( c_{i}^{ce} \): cost of canceling an elective surgery for specialty \( i \)
- \( c_{i}^{cu} \): loss in revenue incurred due to not scheduling an elective case for specialty \( i \)

Decision Variables

- \( x_{aist} \): 1; if elective surgery \( a \) of specialty \( i \) is scheduled for day \( t \) on day \( k \) under realization \( s \)
  0; otherwise
- \( y_{bisk} \): 1; if semi-urgent surgery \( b \) of specialty \( i \) is requested on day \( t \) being performed on day \( k \)
  \( k \in \{t, t+1, t+2\} \) and \( k \leq h \) under realization \( s \)
  0; otherwise
- \( \sigma_{aist} \): 1; if elective surgery \( a \) of specialty \( i \) is cancelled on day \( t \) under realization \( s \)
  0; otherwise
- \( \rho_{bisk} \): 1; if semi-urgent surgery \( b \) of specialty \( i \) is requested on day \( t \) cannot be accommodated
  0; otherwise

Collectively, the set of all decisions corresponding to a realization of the random variables is written as,
\[
\{w_{k}(s)\}_{1 \leq k \leq h} = \{x_{k}(s)\}_{1 \leq k \leq h}, \{y_{k}(s)\}_{1 \leq k \leq h}, \{\sigma_{k}(s)\}_{1 \leq k \leq h}, \{\rho_{k}(s)\}_{1 \leq k \leq h} \}.
\]
Similarly, the set of all de-
decisions on day $t$ is $w_t(s) = \{x_t(s), y_t(s), \sigma_t(s), \rho_t(s)\}$. If $J^t(\{w_k(s)\}_{k \leq t}, s)$ denotes the cost of rescheduling under realization $s$, the resulting multi-stage optimization problem can be formulated as follows:

$$
P_{SR} : \min_{\{w_1\}} \mathbb{E}[J^1(\{w_1\}, \{U_1, D\}) + \min_{\{w_2\}} \mathbb{E}[J^2(\{w_2\}_{1 \leq k \leq 2}, \{U_2, D\}) + \cdots + \min_{\{w_h\}} \mathbb{E}[J^h(\{w_h\}_{1 \leq k \leq h}, \{U_h, D\})]]
$$

subject to

$$w_t \in F_t \quad \forall t \in \{1, \cdots, h\}$$

where $F_t$ is the feasible set for decision variables corresponding to day $t$. Because the expectation (sample average) is taken over the realizations of the random variables in SAA, the random variables at any time period are replaced with their realizations, $S_t$. Thus, the expected cost of rescheduling across the entire horizon starting from day $t$, denoted by $J^t$ under the set of all realizations, $S$, can be written using the recursive equation $J^t(\{w_k(s)\}_{k \leq t}, S) = \mathbb{E}[J^t(\{w_k(s)\}_{k \leq t+1}, S) + J^{t+1}(\{w_k(s)\}_{k \leq t+1}, S)]$. Following the above framework, we can write the optimization formulation corresponding to each day of the multi-stage surgery rescheduling problem under realization $s$ as follows:

$$\min_{\{x^{s}_{1}, y^{s}_{1}, \sigma^{s}_{1}, \rho^{s}_{1}\}} J^1(\{w_1(s)\}, s) = \sum_{i=1}^{n} c_{i}^{s} d_{i}^{s} v_{i1} + \sum_{a \in \mathcal{E}_{1}, k=1}^{3} c_{a}^{s} \sigma_{a1}^{s} + \sum_{b \in \{1, \cdots, u_{11}^{s}\}}^{h-1} \sum_{q=1}^{h} \sum_{k=q+1}^{h} (k-q) c_{b}^{s} x_{a1k}^{s} + \sum_{b \in \{1, \cdots, u_{11}^{s}\}}^{h} \sum_{k=1}^{3} (k-1) c_{b}^{s} y_{b1k}^{s} + c_{1}^{s} \sigma_{11}^{s}$$

subject to

$$\sum_{a \in \mathcal{E}_{1}} x_{a1q}^{s} \leq g_{aiq} \quad \forall i \in \{1, \cdots, n\}, \forall q \in \{1, \cdots, h\} \quad (4.1)$$

$$\sum_{k=q}^{h} x_{a1k}^{s} + \sigma_{a1}^{s} = g_{aiq} \quad \forall i \in \{1, \cdots, n\}, \forall q \in \{1, \cdots, h\}, \forall a \in \{E_q, \cdots, E_T\} \quad (4.2)$$

$$\sum_{a \in \mathcal{E}_{1}} x_{a11}^{s} + \sum_{b \in \{1, \cdots, u_{11}^{s}\}} y_{b11}^{s} \leq v_{i1} \quad \forall i \in \{1, \cdots, n\} \quad (4.3)$$

$$\sum_{k=1}^{3} y_{b1k}^{s} + \rho_{b11}^{s} = 1 \quad \forall i \in \{1, \cdots, n\}, \forall a \in \mathcal{E}_{1} \quad (4.4)$$

$$\sum_{a \in \mathcal{E}_{1}} x_{a1q}^{s} + \sum_{b \in \{1, \cdots, u_{11}^{s}\}} y_{b1q}^{s} \leq v_{iq} \quad \forall i \in \{1, \cdots, n\}, \forall q \in \{2, 3\} \quad (4.5)$$
\begin{align*}
\sum_{a \in \mathcal{E}_1} D_{ai}^s x_{ai1}^s + \sum_{b \in \{1, \ldots, u_1^s\}} D_{bi}^s y_{bi1}^s & - o_{i1}^s - d_i^s v_{i1} \leq 0 & \forall i \in \{1, \ldots, n\} \quad (4.6) \\
x_{ai1q}^s, y_{bi1k}^s, \sigma_{ai1}, \rho_{bi1}^s \in \{0, 1\} & \forall i \in \{1, \ldots, n\}, \forall a \in \mathcal{E}_i, \forall b \in \{1, \ldots, u_1^s\} \quad (4.7) \\
o_{i1}^s & \geq 0 & \forall i \in \{1, \ldots, n\} \quad (4.8)
\end{align*}

The above formulation corresponds to the first day of rescheduling when the random variables realized are indexed by \( s \). Constraint 4.1 restricts surgeries with an initial assignment in the future from being assigned on stage 1. Constraint 4.2 ensures that surgeries are either assigned to one of the days across the entire time horizon where they can be performed or they are canceled. Constraint 4.3 limits the allocation of surgeries such that the capacity constraint on day 1 is satisfied. Constraints 4.4 and 4.5 guide the allocation of semi-urgent surgeries within a span of two days from the day of request such that the capacity constraint is satisfied, and finally, Constraint 4.6 calculates the overtime incurred under realization \( s \). For days 2, \ldots, \( h \), the multi-stage optimization problem can be formulated as follows:

\begin{align*}
\text{minimize} \quad & J^i(\{w_k(s)\}_{k \leq t+1}) = \sum_{i=1}^{n} \left( c_i^t d_i^t v_{it} + \sum_{a \in \{\mathcal{E}_i, \ldots, \mathcal{E}_{t-1}\}} C_{ia}^{ce} \sigma_{ai1}^s + \sum_{b \in \{1, \ldots, u_1^s\}} C_{ib}^{cu} \rho_{bi1}^s + \sum_{q=t}^{h-1} \sum_{a \in \mathcal{E}_i} \sum_{k=q+1}^{h} (k-q)c_i^p x_{aikt}^s \right) + J^{t+1}(\{w_k(s)\}_{k \leq t+1}, S) \\
\text{subject to} \quad & \sum_{q=1}^{t-1} x_{aiqt}^s = \sum_{k=t}^{h} x_{aikt}^s + \sigma_{ai1}^s & \forall i \in \{1, \ldots, n\}, \forall a \in \{\mathcal{E}_i, \ldots, \mathcal{E}_{i(t-1)}\} \quad (4.9) \\
\sum_{a \in \{\mathcal{E}_i, \ldots, \mathcal{E}_{i(t-1)}\}} x_{ai(t-1)}^{s, t} + \sum_{b \in \{1, \ldots, u_1^s\}} \sum_{q=t-2}^{t-1} y_{bitti}^s + \sum_{q=t}^{h-1} \sum_{b \in U_s^t} y_{biqt}^s \leq v_{it} & \forall i \in \{1, \ldots, n\} \quad (4.10) \\
\sum_{k=t}^{h} y_{bitti}^s + \rho_{bi1}^s & = 1 & \forall i \in \{1, \ldots, n\}, \forall b \in \{1, \ldots, u_1^s\} \quad (4.11) \\
\sum_{a \in \mathcal{E}_i} x_{ai(t+1)}^s + \sum_{l \in \{t-1, t\}} \sum_{b \in U_s^t} y_{bi(t+1)}^s & \leq v_{i(t+1)} & \forall i \in \{1, \ldots, n\} \quad (4.12) \\
\sum_{a \in \mathcal{E}_i} x_{ai(t+2)}^s + \sum_{b \in \{1, \ldots, u_1^s\}} y_{bi(t+2)}^s & \leq v_{i(t+2)} & \forall i \in \{1, \ldots, n\} \quad (4.13)
\end{align*}
\[ \sum_{a \in \{E_i \} \cup \{E_{it}\}} D_{ai} x_{aitt} + \sum_{t=1}^{T} \sum_{b \in \{E_{it}\}} D_{ibt} y_{ibt} - o_{it}^* - d_{it}^* v_{it} \leq 0 \quad \forall i \in \{1, \ldots, n\} \]  
\[ x_{aitq}^*, y_{bitk}^*, \sigma_{ait}^*, \rho_{bit}^* \in \{0, 1\} \quad \forall i \in \{1, \ldots, n\}, \forall a \in \{ \} \cup \{E_{it}\}, \forall b \in \{1, \ldots, u_{it}^*\} \]  
\[ \forall q \in \{1, \ldots, h\}, \forall k \in \{t, t+1, t+2\} \]  
\[ o_{it}^* \geq 0 \quad \forall i \in \{1, \ldots, n\} \]  

We let the total expected cost at stage \( h + 1 \), \( J^{h+1}({\{w_k(s)\}_{k \leq h}, S}) = 0 \). Constraint 4.9, equivalent to Constraint 4.2, ensures that assignment decisions from the past are either upheld or the schedule is further modified, where surgeries may be postponed to the future or even canceled. Constraint 4.10 restricts the total number of surgeries performed on any day, including the surgeries postponed from the past does not exceed the available capacity. Constraint 4.11, equivalent to Constraint 4.4 schedules semi-urgent surgeries within a three day limit, failing which they are not performed. Constraints 4.12-4.13 satisfy the capacity restriction on future days while semi-urgent surgeries are postponed. Constraint 4.14 calculates the overtime associated with realization \( s \). As in the case of (Gul et al., 2015), the above multi-stage optimization is NP-hard because it corresponds to the bin packing problem with fixed number of bins.

### 4.3 Solution Methodology

The multi-stage optimization model in Section 4.2.2 minimizes the total expected cost of accommodating elective and semi-urgent surgeries under an uncertain operating environment. Irrespective of the quality of approximation, an optimal solution to the multi-stage optimization problem has the following disadvantages:

(i) It is not implementable in real time because the decisions made on a rolling basis vary with the realized number of semi-urgent surgeries, that is, the scheduling policy is not well defined and varies with each realization.

(ii) The SAA method only minimizes the expected total cost over a subset of possible realizations.

(iii) The multi-stage optimization model for minimizing the expected total cost is NP-hard and there is no guarantee that an optimal solution can be found within the limited duration available to make a decision.

In addition to the solution being impractical to implement, the model has some other drawbacks: (i) given a very short time horizon during which surgeries are rescheduled, reversing previous decisions based on future realizations may be impossible on a daily basis, and (ii) introducing a decision variable for every semi-urgent
surgery results in a random number of decision variables being realized at every day of decision making, increasing the dimensionality of the problem. To overcome the shortcomings of the model, we reformulate the problem by eliminating decision reversibility, and by indexing over the number of surgeries rather than individual surgeries. We introduce notations for the revised problem as follows:

**Revised Decision Variables**

- $x_{it}^s$: elective surgeries of specialty $i$ performed on day $t$ under realization $s$
- $y_{itk}^s$: semi-urgent surgeries of specialty $i$ requested on day $t$ being performed on day $k$, $k \in \{t, t+1, t+2\}$ and $k \leq h+1$
- $p_{itk}^s$: elective surgeries of specialty $i$ postponed from day $t$ to day $k$, $k \in \{t+1, \cdots, h\}$ under realization $s$.
- $\sigma_{it}^s$: electives canceled on day $t$ for specialty $i$ under realization $s$.
- $\rho_{it}^s$: semi-urgent surgeries not scheduled on day $t$ for specialty $i$ under realization $s$.
- $z_{itk}^s$: additional capacity added to day $k$ to accommodate semi-urgent surgeries of specialty $i$ realized under realization $s$ on day $t$.

Note that, even if the multi-stage optimization model in Section 4.2.2 were to be reformulated by eliminating decision reversibility, and by reducing the solution space by removing indices on individual surgeries, the disadvantages with implementing the optimal solution still persist. Therefore, we develop a scheduling model/policy that is uniform to implement, without the need for solving an optimization model in real time, and that also minimizes the total expected cost.

Recall from Section 4.1 that Herring and Herrmann (2012) solve the single day surgery allocation using dynamic programming. Although dynamic programming yields the optimal solution, the dimensionality grows exponentially with the number of surgeries making it impossible to solve a real sized problem\(^1\). Therefore, Herring and Herrmann (2012) resort to solving it using a threshold based scheduling heuristic. Similarly, Zonderland et al. (2010) indicates that for accommodating special surgical requests into an already existing schedule, threshold based policies prove to be optimal. Following their work, we develop a threshold based heuristic that will overcome the drawbacks associated with the original multi-stage optimization model, and will be consistent for the OR scheduler to implement on a daily basis. Further, the threshold policy reduces the dimensionality by limiting the number of decision variables, which should in turn allow us to consider multiple realizations while optimizing the problem.

\(^1\)In a real sized surgery scheduling problem, the operating room capacity is at least 6 hours of operating time per day
4.3.1 Scheduling Heuristic

The heuristic that we propose includes two components: (i) a threshold parameter, and (ii) a scheduling policy that determines the value of the decision variables. By setting a threshold, a portion of the total available capacity is reserved only for performing semi-urgent surgeries, and all requests that fall below the threshold are accommodated within the same day.

If $\alpha_{it}$ denotes the threshold for specialty $i$ on day $t$, the number of semi-urgent surgeries that can be accommodated on the same day, $y^s_{it}$, is restricted to $\alpha_{it}$. However, the scheduler has to not only accommodate semi-urgent surgeries, but also modify the schedule of electives due to such accommodations. As mentioned in Section 4.2, rescheduling entails the following decisions: (i) determining when to schedule the postponed semi-urgent surgeries, that is, determine $y^s_{it(t+1)}$ and $y^s_{it(t+2)}$; (ii) establish a schedule for the elective cases to be performed, $x^e_{it}$, and those that need to be postponed, $p_{itk}$, $t < k \leq h$; and finally, (iii) determine planned overtime in the future, $z^s_{it}$, to perform semi-urgent surgeries. We propose a prioritization based scheduling policy that will work in tandem with the threshold level, and guide the scheduler through the above decisions. Figure 4.2 shows the decision making sequence for the surgical rescheduling problem over a $h$ day horizon.

The scheduling policy we propose determines the above decisions on a repetitive basis in a myopic way for each specialty, that is, only semi-urgent surgeries realized on a particular day are considered, and no information from the future is incorporated into decision making. Such myopic policies are often used as a benchmark while evaluating the performance of numerous scheduling approaches (Gul et al., 2015; Herring and Herrmann, 2012). In addition, each specialty, with its own surgical block, has an independent schedule. Though the duration of surgeries is not considered explicitly in decision making, the effects of random surgical durations are eventually added to the objective function. Note that, given a threshold, any
Scheduling heuristic can be accommodated to fit into the existing framework.

Since semi-urgent surgeries require prompt medical attention, and the loss in revenue associated with not performing a semi-urgent surgery is higher than for electives, a schedule for all semi-urgent surgeries that are realized on a particular day is first determined. As mentioned above, the threshold level determines \( y_{it}^s \), the number of cases that can be accommodated the same day. Surgeries that are to be postponed, \( \theta_u = u_{it}^s - y_{it}^s \), are scheduled the following day if any slack is available in the schedule, that is, \( y_{it(t+1)}^s \) is determined based on \( \alpha_{it(t+1)} \), and any unscheduled surgery is performed on day \( t+2 \), satisfying the condition,

\[
u_{it}^s = y_{it}^s + y_{it(t+1)}^s + y_{it(t+2)}^s + \rho_{it}^s.
\]

If sufficient capacity is not available in the block schedule to accommodate all surgeries, additional capacity, in the form of planned overtime, \( z \), is added. Note that, in this policy, even though we consider specialties to be independent of each other, the additional slots may well be available through release of unused capacity by other specialties (Block release). Nevertheless, we set an upper bound on the planned overtime capacity to account for the effects of unplanned overtime due to random surgical durations. All other surgeries that exceed the threshold, are never scheduled, leading to a loss in revenue. Once semi-urgent surgeries have been accommodated, fixing those decisions, the existing schedule for electives is modified. Out of the \( e_{it} \) electives that are pre-scheduled, \( x_{it}^s \) are performed, where \( x_{it}^s = e_{it} - \alpha_{it} - \sum_{k<t} \rho_{ikt}^s \), that is, all available slots after accommodating semi-urgent surgeries are released for electives. Other electives, \( \theta_E = e_{it} - x_{it}^s \), have to be postponed or canceled, and subsequently, a new schedule determined. For this purpose, we derive a threshold for elective surgeries, \( \gamma_{it} \), by letting all unreserved capacity to be utilized for scheduling electives, that is, \( \gamma_{it} = v_{it} - \alpha_{it} \). Following a process similar to that of postponing semi-urgent cases, the schedule for postponed electives, \( p_{ikt}^s, t < k < h + 1 \) is determined by utilizing \( \gamma_{it} \) as a threshold, and all electives that cannot be postponed are canceled. The heuristic is repeatedly applied across the entire time horizon for any realization, with rescheduling decisions made every day. These decisions are, in turn, used to evaluate the objective function which includes the cost of postponement, additional capacity, regular time, and unplanned overtime.

### 4.3.2 Total Cost of Rescheduling

If \( \mathcal{H} \) denotes the heuristic, under realization \( s \), the resulting schedule is, \( (x, y, z, p, \sigma, \rho) = \mathcal{H}_s(\alpha) \). With \( \alpha \) fixed, the total cost of rescheduling under any realization \( s \) is:

\[
J(\alpha, s) = \sum_{i=1}^{n} \sum_{t=1}^{h} \left( c_{it}^e \sigma_{it}^s + c_{it}^w \rho_{it}^s + \sum_{k>t} c_{it}^p (k-t) p_{ikt}^s \right) + \sum_{k=t}^{h} c_{it}^p \rho_{ikt}^s.
\]
Algorithm 5 Threshold based Scheduling Heuristic

Require: Initial capacity allocation, \( v \); Initial schedule of elective surgeries, \( E \); Threshold level, \( \alpha \); Semi-urgent cases realized under realization \( s, u^* \).

Ensure: Schedule of semi-urgent surgeries, \( y \); Schedule of elective surgeries, \( x \); Postponed electives, \( p \);
Semi-urgent surgeries performed in planned overtime, \( z \)

\[
\text{for } t \in \{1, \cdots, h\} \text{ do}
\]
\[
\text{for } i \in \{1, \cdots, n\} \text{ do}
\]
\[
\text{Set } \beta_{it} \leftarrow \alpha_{it}; \gamma_{it} \leftarrow v_{it} - \alpha_{it}
\]
\[
\text{Set } \theta_u \leftarrow 0, \theta_E \leftarrow 0
\]

Schedule Semi-urgent Surgeries & Electives
\[
y_{itt} \leftarrow \min \left( U_{it}, \alpha_{it} - 1_{\{t \geq 2\}} y_{i(t-1)t} + 1_{\{t \geq 3\}} \left( z_{i(t-2)t} - y_{i(t-2)t} \right) \right)
\]
\[
x_{it} \leftarrow \min \left( \epsilon_{it}, v_{it} - \alpha_{it} + 1_{\{t \geq 3\}} z_{i(t-2)t} - 1_{\{t \geq 2\}} \sum_{k < t} p^k_{it} \right)
\]
\[
\theta_u \leftarrow U_{it} - y_{itt}, \theta_E \leftarrow \epsilon_{it} - x_{it}
\]

Schedule Postponed Semi-urgent Surgeries
\[
\text{if } \theta_u > 0 \text{ then}
\]
\[
y_{it(t+1)} \leftarrow \min \left( \theta_u, \alpha_{i(t+1)} - 1_{\{t \geq 2\}} y_{i(t-1)(t+1)} + 1_{\{t \geq 3\}} z_{i(t-1)(t+1)} \right)
\]
\[
y_{it(t+2)} \leftarrow \min \left( \theta_u - \sum_{q=t}^{t+1} y_{itq}, \alpha_{i(t+2)} + \delta_{i(t+2)} \right)
\]
\[
z_{it(t+2)} \leftarrow \left( y_{it(t+2)} - \alpha_{i(t+2)} \right)^+
\]
\[
\rho_{it} \leftarrow \theta_u - \sum_{q=t}^{t+2} y_{itq}
\]
\[
\text{else if } t = h - 1 \text{ then}
\]
\[
y_{it(t+1)} \leftarrow \min \left( \theta_u, \alpha_{i(t+1)} - 1_{\{t \geq 2\}} y_{i(t-1)(t+1)} + 1_{\{t \geq 2\}} z_{i(t-1)(t+1)} \right)
\]
\[
\rho_{it} \leftarrow \theta_u - \sum_{q=t}^{t+1} y_{itq}
\]
\[
\text{else}
\]
\[
\rho_{it} \leftarrow \theta_u
\]
\[
\text{end if}
\]

Schedule Postponed Elective Surgeries
\[
\text{if } \theta_E > 0 \text{ then}
\]
\[
q \leftarrow t + 1
\]
\[
\text{while } \theta_E > 0 \text{ and } q \leq h \text{ do}
\]
\[
p_{itq} \leftarrow \min \left( \theta_E, \left( \gamma_{iq} - n_{iq} - 1_{\{t \geq 2\}} \sum_{l=1}^{t-1} p_{ilq} \right)^+ \right)
\]
\[
\theta_E \leftarrow \theta_E - p_{itq}
\]
\[
q \leftarrow q + 1
\]
\[
\text{end while}
\]
\[
\sigma_{it} \leftarrow \theta_E
\]
\[
\text{end if}
\]
\[
\text{end for}
\]
\[
\max_{(h,t+2)} \sum_{k=t+1} \left( c^{p}_{ik}(k-t) y_{itk} + c^{p}_{it} d^{u}_{it} z^{s}_{it} + c^{r}_{it} d^{r}_{it} v_{it} + c^{r}_{it} \sigma^{s}_{it} \right),
\]
where $o_{it}^s$ is the overtime incurred under realization $s$.

### 4.4 Threshold Determination

In Section 4.3, we elaborated the scheduling heuristic, where, given a threshold, $\alpha$, and the request for semi-urgent cases, $u^*$, a feasible schedule is determined. The cost of implementing this scheduling policy, $J(\alpha, s)$, is most sensitive to the choice of threshold, $\alpha$. Therefore, it is important that the threshold levels are aligned with other parameters in the model, including the cost, and the distribution of the random variables.

However, finding the best threshold is a non-trivial problem due to the following reasons, (i) the number of feasible alternatives grows exponentially with the capacity of the operating room, $v$, and the number of specialties, $n$, (ii) the threshold policy induces a non-linear relationship between the threshold levels, $\alpha$, and the resulting cost.

The operators that link the threshold, $\alpha$ with the scheduling decisions, $(x, y, z, p, \rho, \sigma)$ are clearly non-linear. With respect to the feasible space, consider the surgical rescheduling problem with $n = 1$, and $h = 5$, and $v_t = 10, \forall t \in 1, \cdots, h$. Though the number of decision variables is five, a threshold for each day of the week, finding the best solution requires us to evaluate 100,000 feasible solutions, and optimization techniques that do not exploit the structure of the problem may fail in identifying a good solution. Following is a formulation of the non-linear optimization problem to determine the optimal threshold level.

$$P^T : \begin{align*}
\text{minimize} & \quad J(\alpha) = \mathbb{E}[J(\alpha, \{U_t\}_{1 \leq t \leq h}, \{D_t\}_{1 \leq t \leq h})] \\
\text{subject to} & \quad \alpha_{it} \leq v_{it} \quad \forall i \in \{1, \cdots, n\}, \forall t \in 1, \cdots, h
\end{align*}$$

Note that, unlike the multi-stage optimization framework in Section 4.2, the threshold determination problem, $P^T$ is a static optimization problem under uncertainty. As in the case of the multi-stage optimization framework, explicitly incorporating the distribution is still impossible, and we resort to using the SAA method using realizations of the random variables. Given a threshold level, $\alpha$, Monte Carlo simulation generates the realizations, $S$. For each realization, $s \in S$, the scheduling heuristic, $H$ is applied and the resulting rescheduling cost, $J(\alpha, s)$, is observed. The reformulated problem with the expected rescheduling cost replaced by the sample mean is:

$$\tilde{P}^T_S : \begin{align*}
\text{minimize} & \quad \tilde{J}(\alpha, S) = \frac{1}{|S|} \sum_{s \in S} J(\alpha, s) \\
\text{subject to} & \quad \alpha_{it} \leq v_{it} \quad \forall i \in \{1, \cdots, n\}, \forall t \in 1, \cdots, h
\end{align*}$$
Problem $\hat{P}_S^T$ is the deterministic equivalent of the stochastic threshold determination problem and can be solved using one of the many existing optimization techniques. Let $\alpha^*_S$ denote the optimal solution to the approximation, $\hat{P}_S^T$, and the corresponding expected cost be $J(\alpha^*_S)$. If the set of optimal solutions to the true problem, $P^T$ is denoted by $\pi^*$, then $\alpha^*_S$ converges to an element in $\pi^*$, when the number of realizations, $|S|$, is large, specifically, as $|S| \to \infty$.

Our objective is to identify a solution that is close enough to the set $\pi^*$ without sacrificing the finite time performance. Towards this goal, we consider two approaches: (i) a SAA technique, that is, solving Problem $\hat{P}_S^T$ with fixed number of realizations, and (ii) a simulation based optimization approach. By introducing a simulation based approach, we emphasize on quantifying the overwhelming impact of approximation strategies (uncertainty) on stochastic optimization problems. Further, we restrict the scope of this work to only comparing the quality of solution obtained using the SAA method and a simulation optimization method, and exclude studying other advanced decomposition techniques.

### 4.4.1 Optimize threshold using Integer Programming

In this section, we present an integer programming formulation for optimizing the threshold levels when heuristic $\mathcal{H}$ is utilized for the rescheduling decisions. With uncertainty in surgical requests restricted to a finite number of realizations, the objective of this formulation is to minimize the sample mean of the total cost across these realizations. Given the requirement to explicitly formulate the policy in the mathematical model, individual scheduling rules are linearized using convexification procedures. The resulting linear integer programming formulation for threshold optimization is as follows:

$$
\text{minimize } J(\alpha, S) = c_i d_i^v v_{it} + \frac{1}{|S|} \sum_{i=1}^{|S|} \left( \sum_{t=1}^h \left( \sum_{i=1}^n \left( C_i^c \sigma_i^h_t + C_i^c \rho_i^t \sum_{k>t}^h c_i^p (k-t) p_{itk} \right) + \sum_{k=t+1}^{\max(h,t+2)} c_i^p (k-t) y_{itk} + c_i^v d_i^v z_{it} \right) \right) \quad (4.18)
$$

subject to

1. $\alpha_{it} \leq v_{it} \quad \forall i, \forall t \in 1, \cdots, h$ (4.19)
2. $\gamma_{it} = v_{it} - \alpha_{it} \quad \forall i, \forall t \in 1, \cdots, h$ (4.20)
3. $y_{itt}^s \leq u_{it}^s \quad \forall s, \forall i, \forall t \in 1, \cdots, h$ (4.21)
4. $y_{itt}^s \leq A_{it}^s \quad \forall s, \forall i, \forall t \in 1, \cdots, h$ (4.22)
\[
\begin{align*}
    u^s_{it} & \leq y^s_{itt} + (1 - G^1_s) \times M & \forall s, \forall i, \forall t & \in 1, \ldots, h \quad (4.23) \\
    A^s_{it} & \leq y^s_{itt} + (1 - H^1_s) \times M & \forall s, \forall i, \forall t & \in 1, \ldots, h \quad (4.24) \\
    G^1_s + H^1_s & = 1 & \forall s, \forall i, \forall t & \in 1, \ldots, h \quad (4.25) \\
    A^s_{it} & \geq \alpha_{it} - \mathbb{1}_{(t \geq 3)} \sum_{k=t-2}^{t} y^s_{ikt} + \mathbb{1}_{(t \geq 3)} z^s_{i(t-2)t} & \forall s, \forall i, \forall t & \in 1, \ldots, h \quad (4.26) \\
    A^s_{it} & \geq v_{it} - \alpha_{it} - \mathbb{1}_{(t \geq 2)} \sum_{k=1}^{t-1} p^s_{ikt} - \mathbb{1}_{(t \geq 3)} \sum_{k=t-2}^{t-1} y^s_{ikt} \\
    & \quad + \mathbb{1}_{(t \geq 3)} z^s_{i(t-2)t} & \forall s, \forall i, \forall t & \in 1, \ldots, h \quad (4.27) \\
    \alpha_{it} - \mathbb{1}_{(t \geq 3)} \sum_{k=t-2}^{t} y^s_{ikt} + \mathbb{1}_{(t \geq 3)} z^s_{i(t-2)t} & \geq A^s_{it} - (1 - G^2_s) \times M & \forall s, \forall i, \forall t & \in 1, \ldots, h \quad (4.28) \\
    v_{it} - \alpha_{it} - \mathbb{1}_{(t \geq 2)} \sum_{k=1}^{t-1} p^s_{ikt} - \mathbb{1}_{(t \geq 3)} \sum_{k=t-2}^{t-1} y^s_{ikt} \\
    & \quad + \mathbb{1}_{(t \geq 3)} z^s_{i(t-2)t} & \geq A^s_{it} - (1 - H^2_s) \times M & \forall s, \forall i, \forall t & \in 1, \ldots, h \quad (4.29) \\
    G^2_s + H^2_s & = 1 & \forall s, \forall i, \forall t & \in 1, \ldots, h \quad (4.30) \\
    x^s_{it} & \leq e_{it} & \forall s, \forall i, \forall t & \in 1, \ldots, h \quad (4.31) \\
    x^s_{it} & \leq v_{it} + \mathbb{1}_{(t \geq 3)} z^s_{i(t-2)t} - \mathbb{1}_{(t \geq 2)} \sum_{k=1}^{t-1} p^s_{ikt} - \mathbb{1}_{(t \geq 3)} \sum_{k=t-2}^{t-1} y^s_{ikt} & \forall s, \forall i, \forall t & \in 1, \ldots, h \quad (4.32) \\
    e_{it} & \leq x^s_{it} + (1 - G^3_s) \times M & \forall s, \forall i, \forall t & \in 1, \ldots, h \quad (4.33) \\
    v_{it} + \mathbb{1}_{(t \geq 3)} z^s_{i(t-2)t} - \mathbb{1}_{(t \geq 2)} \sum_{k=1}^{t-1} p^s_{ikt} - \mathbb{1}_{(t \geq 3)} \sum_{k=t-2}^{t-1} y^s_{ikt} & \leq \\
    & \quad x^s_{it} (1 - H^3_s) \times M & \forall s, \forall i, \forall t & \in 1, \ldots, h \quad (4.34) \\
    G^3_s + H^3_s & = 1 & \forall s, \forall i, \forall t & \in 1, \ldots, h \quad (4.35) \\
    y^s_{it(t+1)} & \leq u^s_{it} - y^s_{itt} & \forall s, \forall i, \forall t & \in 1, \ldots, h - 1 \quad (4.36) \\
    y^s_{it(t+1)} & \leq \alpha_{i(t+1)} - \mathbb{1}_{(t \geq 2)} y^s_{it(t-1)(t+1)} & \forall s, \forall i, \forall t & \in 1, \ldots, h - 1 \quad (4.37) \\
    u^s_{it} - y^s_{itt} & \leq y^s_{it(t+1)} + (1 - G^4_s) \times M & \forall s, \forall i, \forall t & \in 1, \ldots, h - 1 \quad (4.38) \\
    \alpha_{i(t+1)} - \mathbb{1}_{(t \geq 2)} y^s_{it(t-1)(t+1)} & \leq y^s_{it(t+1)} + (1 - H^4_s) \times M & \forall i, \forall t & \in 1, \ldots, h - 1 \quad (4.39) \\
    G^4_s + H^4_s & = 1 & \forall s, \forall i, \forall t & \in 1, \ldots, h - 1 \quad (4.40) \\
    y^s_{it(t+2)} & \leq u^s_{it} - y^s_{itt} - y^s_{it(t+1)} & \forall s, \forall i, \forall t & \in 1, \ldots, h - 2 \quad (4.41) \\
    y^s_{it(t+2)} & \leq \alpha_{i(t+2)} + \delta_{i(t+2)} & \forall s, \forall i, \forall t & \in 1, \ldots, h - 2 \quad (4.42) \\
    u^s_{it} - y^s_{itt} - y^s_{it(t+1)} & \leq y^s_{it(t+2)} + (1 - G^5_s) \times M & \forall s, \forall i, \forall t & \in 1, \ldots, h - 2 \quad (4.43) \\
    \alpha_{i(t+2)} + \delta_{i(t+2)} & \leq y^s_{it(t+2)} + (1 - H^5_s) \times M & \forall s, \forall i, \forall t & \in 1, \ldots, h - 2 \quad (4.44)
\end{align*}
\]
\begin{align*}
G_{it}^{5s} + H_{it}^{5s} &= 1
\end{align*}
\forall s, \forall i, \forall t \in 1, \ldots, h - 2 \quad (4.45)
\begin{align*}
\alpha_{it} + \alpha_{i(t+2)} &\leq y_{it(t+2)}^{s} - \frac{1}{2}(1 - G_{it}^{6s}) \times M
\end{align*}
\forall s, \forall i, \forall t \in 1, \ldots, h - 2 \quad (4.46)
\begin{align*}
z_{it(t+2)}^{s} &\geq z_{it(t+2)}^{s} - \frac{1}{2}(1 - H_{it}^{6s}) \times M \geq 0
\end{align*}
\forall s, \forall i, \forall t \in 1, \ldots, h - 2 \quad (4.47)
\begin{align*}
G_{it}^{6s} + H_{it}^{6s} &= 1
\end{align*}
\forall s, \forall i, \forall t \in 1, \ldots, h - 2 \quad (4.48)
\begin{align*}
p_{itk}^{s} &\geq u_{it}^{s} - \frac{1}{2}(1 - H_{it}^{7s}) \times M
\end{align*}
\forall s, \forall i, \forall t \in 1, \ldots, h - 1 \quad (4.49)
\begin{align*}
\rho_{it}^{s} &\leq \rho_{it}^{s} + (1 - G_{itk}^{7s}) \times M
\end{align*}
\forall s, \forall i, \forall t \in 1, \ldots, h - 1 \quad (4.50)
\begin{align*}
B_{itk}^{s} &\leq B_{itk}^{s} + (1 - G_{itk}^{7s}) \times M
\end{align*}
\forall s, \forall i, \forall t \in 1, \ldots, h - 1 \quad (4.51)
\begin{align*}
p_{itk}^{s} &\leq e_{it} - \frac{1}{2}(1 - H_{it}^{7s}) \times M
\end{align*}
\forall s, \forall i, \forall t \in 1, \ldots, h - 1 \quad (4.52)
\begin{align*}
G_{itk}^{7s} + H_{itk}^{7s} \times M &= 1
\end{align*}
\forall s, \forall i, \forall t \in 1, \ldots, h - 1 \quad (4.53)
\begin{align*}
B_{itk}^{s} &\geq v_{ik}^{s} - \frac{1}{2}(1 - H_{it}^{8s}) \times M
\end{align*}
\forall s, \forall i, \forall t \in 1, \ldots, h - 1 \quad (4.54)
\begin{align*}
v_{ik}^{s} &\geq v_{ik}^{s} - \frac{1}{2}(1 - H_{it}^{8s}) \times M
\end{align*}
\forall s, \forall i, \forall t \in 1, \ldots, h - 1 \quad (4.55)
\begin{align*}
B_{itk}^{s} &\geq B_{itk}^{s} - (1 - G_{itk}^{8s}) \times M
\end{align*}
\forall s, \forall i, \forall t \in 1, \ldots, h - 1 \quad (4.56)
\begin{align*}
\sigma_{it}^{s} &\geq \sigma_{it}^{s} - \frac{1}{2}(1 - H_{it}^{8s}) \times M
\end{align*}
\forall s, \forall i, \forall t \in 1, \ldots, h - 1 \quad (4.57)
\begin{align*}
G_{itk}^{8s} + H_{itk}^{8s} &= 1
\end{align*}
\forall s, \forall i, \forall t \in 1, \ldots, h - 1 \quad (4.58)
\begin{align*}
G_{it}^{8s} + H_{it}^{8s} &= 1
\end{align*}
\forall s, \forall i, \forall t \in 1, \ldots, h - 1 \quad (4.59)
\begin{align*}
\sigma_{it}^{s} &\geq \sigma_{it}^{s} - \frac{1}{2}(1 - H_{it}^{8s}) \times M
\end{align*}
\forall s, \forall i, \forall t \in 1, \ldots, h - 1 \quad (4.60)
The objective function, 4.18, is the sum of the regular cost for a fixed block schedule, and the cost of cancellation, postponement, and capacity addition across the time horizon for the individual specialties. Note that the cost of overtime is ignored, for doing so requires that individual surgeries are indexed often resulting in an intractable model. To illustrate the linearization of constraints, and thus the rules in the scheduling heuristic, \( H \), consider the constraint \( Z = \min(A, B) \). For the \( \min \) operator, convexification involves the following:

\[
\begin{align*}
Z & \leq A \\
Z & \leq B \\
A & \leq Z + (1 - d_1) \times M \\
B & \leq Z + (1 - d_2) \times M \\
d_1 + d_2 & = 1 \\
d_1, d_2 & \in \{0, 1\}
\end{align*}
\]

Notice that the convexification of the \( \min \) operator results in addition of two binary variables, \( d_1 \), and \( d_2 \), and four additional constraints. The parameter, \( M \) is the big-M variable that is sufficiently large, set to be 10,000 throughout the formulation. Using a similar procedure, the \( \max \) operator can be linearized. Upon linearization, the scheduling rule that governs the accommodation of semi-urgent surgeries on the same day of request is represented by constraints 4.19 - 4.30. The binary variables \( G \), and \( H \) are the dummy variables from the linearization procedure throughout this formulation. Constraints 4.31 - 4.35 ensure that the total number of electives that can be accommodated out of the \( e_{it} \) cases pre-scheduled on any day \( t \) does not exceed the capacity available after accommodating semi-urgent surgeries, and previously postponed electives.

Constraints 4.36 - 4.49 correspond to the rules for postponing semi-urgent surgeries, and their accom-
modation within two days from the day of request, with any additional planned overtime capacity. Note that for days $h-1$ and $h$, any semi-urgent surgery that cannot be accommodated within the last day of the week are canceled resulting in a loss of revenue, $c^\text{cm}_i$, number of which is denoted by constraint 4.50. Constraints 4.52 - 4.59 correspond to the scheduling rules for postponed electives. Similar to the semi-urgent surgeries, electives that cannot be accommodated within the end of the week are canceled incurring a cancellation cost of $c^\text{ce}_i$. Though the class of constraints in the formulation is restricted to four, that is, an individual class of rules for scheduling semi-urgent surgeries and electives followed by their postponement, the addition of binary variables adds to the computational complexity of the problem. For example, linearizing the constraints in the rescheduling problem with $n = 2$, $|S| = 1$, and $h = 5$, results in 80 binary variables being added to the problem. The threshold determination problem is also an NP-hard because the threshold determination under one realization is still equivalent to the bin packing problem with constraints on the number of bins.

### 4.4.2 Optimize threshold using Simulation Optimization - ADALINE

Classical optimization algorithms to solve the threshold optimization model provided above are often iterative, and visit a sequence of solutions which improve the objective function until a termination criterion is satisfied. However, due to the stochastic nature of the problem, quality of the resulting solution, $\alpha^*_S$, is heavily reliant on the number of random realizations. Although incorporating a large number of realizations yields a better approximation, the computational efficiency of the algorithm greatly diminishes with the number of realizations. On the other hand, there is no guarantee on the quality of $\alpha^*_S$ when only a handful of realizations are used. In addition, failure to incorporate the quality of an iterate in choosing the number of realizations induces inefficiency in these algorithms. Therefore, a balance between optimization, and sampling is often required to find a good quality solution without compromising on the algorithm’s performance.

Moreover, the optimization approach requires the analytical form of the objective function and constraints to be explicitly specified, often rendering it inflexible to accommodate even minor modifications in scheduling policy. Likewise, even seemingly simple mathematical models can be hard to solve when non-convexity is induced in the constraint space due to the scheduling policy. For instance, in the threshold determination problem, reformulation of the non-convex constraints that govern the scheduling heuristic, $\mathcal{H}$, results in dummy variables, and reformulation constraints being explicitly added to the model. This significantly increases the complexity of the model, despite the fact that the total number of decision variables are only $n \times h$.

---

2 On specifying a realization, $s$, and the threshold level, $\alpha$, the scheduling policy yields the realized cost of rescheduling, $J^s$, which is equivalent to the random outcome from a simulation. Therefore, we use the terms realization and sample size interchangeably throughout this work.
To overcome the issues of sampling induced inefficiency, approximation quality, and inflexibility associated with traditional optimization models, we propose a simulation based optimization approach for threshold optimization. In the context of surgery rescheduling, simulation optimization serves two purposes: (i) address the impact of the number of realizations on solution quality and also accommodate adaptive sampling strategies, and (ii) flexible enough to work with a broad range of threshold policies for scheduling. We apply ADALINE, a simulation optimization algorithm for integer-ordered feasible sets, developed earlier in Chapter 2 for the threshold optimization problem.

ADALINE identifies a local minimum by incorporating a line search and a neighborhood enumeration procedure within an adaptive sampling framework. Recall that, a feasible solution $x^*$ is a local minimum if and only if any other feasible solution in the neighborhood is inferior to $x^*$. ADALINE trades off the error due to estimation with the quality of the visited solution, expending more simulation effort at solutions close to a local minimum. This scheme, in addition to being efficient also minimizes the potential for inferior solutions to be deemed a local minimum.

ADALINE distinguishes between the simulation model that generates the total cost using a scheduling policy from the optimization procedure for determining the threshold. By doing so, it eliminates the need for explicitly specifying the scheduling policy using a mathematical formulation. Unlike the SAA based threshold optimization, where overtime costs are excluded, the scheduling policy in ADALINE can optimize the threshold inclusive of the overtime costs, which, in some instances, may be as high as 12% (Dexter et al., 2001) of the total cost. Even though ADALINE is designed to yield a local minimum, as we observe later in Section 4.5 from our numerical results, better solutions are obtained using ADALINE than via the classical approach. This indicates that the consequences of not including overtime costs, and the approximation error due to limited sampling have a greater impact on the quality of the resulting solution. Following is the framework for implementing ADALINE on the surgery rescheduling problem:

\begin{algorithm}[h]
\caption{Threshold Optimization using ADALINE}
\begin{algorithmic}[1]
\Require Initial threshold level, $\alpha_0$; Minimum sample size, $M_{\text{min}} \geq 2$; Total simulation budget, $B$
\Ensure Solution, $\alpha^* \in X$ such that $\hat{J}(\alpha^*) < \hat{J}(\alpha)$, $\forall \alpha$ such that $||\alpha^* - \alpha|| = 1$
\begin{algorithmic}
\State Generate $M_{\text{min}}$ realizations of the random surgical demand, $u^*$ and the duration of surgeries, $D^*$, $s = 1, \cdots, M_{\text{min}}$.
\State $\alpha^* \leftarrow \text{ADALINE}(\alpha_0, M_{\text{min}})$
\end{algorithmic}
\end{algorithmic}
\end{algorithm}
4.5 Numerical Results

For our numerical analysis, we consider a hospital that follows a block schedule with operating room capacity shared between four surgical specialties. We let the planning horizon for rescheduling decisions, $h$, to be five days. We assume that the block schedule is aggregated at the level of number of surgeries that can be performed on any particular day. Due to lack of data, we generate a synthetic data set where 80% of the total available capacity is scheduled with electives, closely reflecting the elective surgery utilization that hospitals plan for (Tyler et al., 2003). Further, we assume that the semi-urgent surgeries are requested at the start of the day, the number of which follows a negative binomial distribution. Table 4.1 shows the mean and standard deviation of the number of semi-urgent surgeries requested every day for these individual surgical groups. Following empirical studies on the duration of surgeries (Min and Yih, 2010; Strum et al., 2000), we let the duration of every surgery follow a lognormal distribution. In table 4.3, the parameters of the distribution of surgery duration is presented for each specialty. The maximum number of surgeries that can be performed on any particular day is set to be 10 for each of the surgical groups. Assuming that 80% of the capacity is scheduled with elective cases, a total of eight surgeries are assigned in advance to one of the days in the planning horizon. The cost of planned operating room capacity per hour, that is, regular time capacity is $c^r = (500, 600, 700, 400)$. The overtime cost per hour is set to be 1.5 times the cost of regular

<table>
<thead>
<tr>
<th>Surgical Group</th>
<th>Day 1</th>
<th>Day 2</th>
<th>Day 3</th>
<th>Day 4</th>
<th>Day 5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Mean</td>
<td>Mean</td>
<td>Mean</td>
<td>Mean</td>
</tr>
<tr>
<td>Specialty 1</td>
<td>3.8</td>
<td>4.0</td>
<td>4.0</td>
<td>3.8</td>
<td>4.0</td>
</tr>
<tr>
<td>Specialty 2</td>
<td>4.3</td>
<td>3.3</td>
<td>3.3</td>
<td>4.3</td>
<td>5.0</td>
</tr>
<tr>
<td>Specialty 3</td>
<td>4.6</td>
<td>3.2</td>
<td>4.6</td>
<td>4.6</td>
<td>1.5</td>
</tr>
<tr>
<td>Specialty 4</td>
<td>4.1</td>
<td>5.0</td>
<td>4.5</td>
<td>4.3</td>
<td>3.3</td>
</tr>
</tbody>
</table>

Table 4.1: Table shows the mean and standard deviation of the negative binomially distributed random variable indicating the number of semi-urgent cases requested on any particular day.

<table>
<thead>
<tr>
<th>Surgical Group</th>
<th>Mean</th>
<th>Standard deviation</th>
<th>Skewness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specialty 1</td>
<td>104.0</td>
<td>17.5</td>
<td>3.0</td>
</tr>
<tr>
<td>Specialty 2</td>
<td>89.5</td>
<td>7.0</td>
<td>1.1</td>
</tr>
<tr>
<td>Specialty 3</td>
<td>114.5</td>
<td>4.5</td>
<td>1.5</td>
</tr>
<tr>
<td>Specialty 4</td>
<td>90.0</td>
<td>8.5</td>
<td>1.3</td>
</tr>
</tbody>
</table>

Table 4.2: Table shows the parameters of the the distribution of surgery duration
capacity. The postponement cost per day is $500 for every surgery postponed, and the cancellation cost is set to be \( c^c = (2000, 3000, 4000, 3000) \). The cost of not performing semi-urgent surgeries is $1000 higher than the cost of cancelling electives across all surgical groups. Recall that, given a realization \( s \), and a threshold level, \( \alpha \), scheduling heuristic \( H \) yields the total cost of rescheduling across the entire time horizon.

### 4.5.1 Threshold Optimization using ADALINE

We first illustrate the performance of ADALINE on the optimal threshold determination problem. The starting solution for the algorithm, \( \alpha_0 \), is set to have 9 spots reserved for semi-urgent surgeries every day. The input parameters of ADALINE, the minimum sample size, \( M_{\text{min}} \), and the total simulation budget, \( B \), are 2 and 200,000 respectively. We perform 10 independent replications of the algorithm starting with the same solution and determine the optimal threshold levels. Then, the expected cost of rescheduling for any threshold level is estimated using a simulation with large sample size, \( \tilde{M} \approx 1500 \).

Figure 4.3 shows the percentile values of the expected cost corresponding to optimal solutions found by multiple replications of the algorithm. As we can see, the algorithm quickly converges in the vicinity of a local solution within a budget of 5000 calls. In Figure 4.4, the performance of ADALINE as a function of elapsed time is illustrated.

![Threshold Optimization - ADALINE](image)

**Figure 4.3:** Figure shows the expected total cost of rescheduling as a function of the total simulation calls for the solutions identified by ADALINE at specified budget levels.
4.5.2 Threshold Optimization using Integer programming

We solve the SAA problem with sample sizes (number of realizations), $M = 1, 2, 3, 5, 10, 20$ and compare it with the solution from ADALINE. Corresponding to every value of $M$, the best solution obtained within a specified time limit, and its optimality gap is determined. Similar to the implementation of ADALINE, to account for uncertainty in semi-urgent surgeries, and thus its impact on the obtained solution, we perform 10 independent replications of the algorithm for every value of $M$. We implement the threshold optimization problem using PuLP (Mitchell et al., 2011), a mathematical programming environment in Python. The resulting integer optimization problem is solved using IBM ILOG CPLEX 12.0 on a PC with 2.40 GHz Intel Core i7 and 8 GB memory.

The size of the deterministic equivalent increases rapidly with the sample size, $M$ and early termination of the algorithm results in inferior solutions. For larger instances of the threshold optimization problem, the optimality gap of the resulting solution may be as high as 35\%. Table 4.3 lists the average optimality gap of the resulting solution corresponding to varying values of $M$, and at specified time limits.

Even though larger values of $M$ yield solutions that close the optimality gap, guaranteeing such a solution in finite time may be impossible. We believe that an appropriate sampling based decomposition strategy, as developed in (Gul et al., 2015), will result in better solutions, while not being computationally intensive. However, such techniques usually entail more overhead due to time incurred with modeling.
Table 4.3: Average optimality gap of the solutions to the mixed integer programming formulation obtained at various time limits

<table>
<thead>
<tr>
<th>Sample Size</th>
<th>Time (in minutes)</th>
<th>0.5</th>
<th>1.0</th>
<th>2.0</th>
<th>3.0</th>
<th>5.0</th>
<th>10.0</th>
<th>20.0</th>
<th>30.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>17.2</td>
<td>14.1</td>
<td>11.9</td>
<td>11.3</td>
<td>10.2</td>
<td>8.1</td>
<td>6.9</td>
<td>6.5</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>43.8</td>
<td>38.0</td>
<td>34.2</td>
<td>32.8</td>
<td>32.0</td>
<td>31.1</td>
<td>29.6</td>
<td>29.1</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>56.6</td>
<td>48.4</td>
<td>45.1</td>
<td>44.6</td>
<td>43.3</td>
<td>40.9</td>
<td>35.9</td>
<td>34.2</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>83.4</td>
<td>74.3</td>
<td>53.7</td>
<td>50.3</td>
<td>44.7</td>
<td>44.0</td>
<td>38.8</td>
<td>38.0</td>
</tr>
</tbody>
</table>

Figure 4.5 compares solution quality from ADALINE and by solving the integer optimization problem as a function of the elapsed time. For every solution obtained by solving the deterministic equivalent, the expected cost of rescheduling is determined by simulating the rescheduling policy with a large sample size, $\tilde{M}$. It can be seen that the average expected cost of the solution obtained using ADALINE across 10 independent replications of the algorithm is at least 2% better than the average expected solution from CPLEX. When the fixed regular time cost of operating room resources is ignored, ADALINE decreases the rescheduling cost by close to 4% than the best solution from the SAA technique with fixed number of realizations. Note that ADALINE converges to the vicinity of an optimal solution within half a second, and without the need for any significant modeling effort. A simulation optimization framework with ADALINE also provides flexibility in the choice of scheduling policy. In Figure 4.6, expected cost of 100 different solutions returned from ADALINE and solutions obtained by solving the SAA problem with $M=2$ are plotted in a histogram. As clearly seen in the figure, almost 70% of the solutions from ADALINE have an expected cost of $384,000$, whereas 90% of the solutions from solving the integer optimization problem are worse than $390,000$. Though ADALINE is designed to find a local minimum, its ability to adaptively determine the sample size results in superior performance over the SAA method.

4.6 Conclusion

In this work, we propose a surgery rescheduling policy to accommodate semi-urgent surgeries on a daily basis under demand uncertainty. To overcome implementation issues associated with solutions from a multi-stage optimization framework, we propose a threshold based scheduling policy that exclusively reserves a
Figure 4.5: Figure shows the expected total cost of rescheduling as a function of elapsed time for the solutions identified by the integer optimization framework with various sample sizes.

Figure 4.6: Figure shows a comparison of the quality of solutions obtained with thresholds determined through ADALINE and the Integer Optimization framework.

Proportion of the capacity for performing semi-urgent surgeries. In order to determine the optimal threshold levels, we utilize an integer programming approach, and a simulation optimization method.
For the integer optimization method, the scheduling policy is formulated as a linear integer program through convexification of constraints, and solved using CPLEX. This method yields inferior solutions for they approximate the problem using a limited number of realizations failing to account for the uncertainty in determining an optimal solution. Furthermore, the size of the problem is large, even with a handful of realizations, resulting in the integer programming approach often failing to find an optimal solution within finite time. On the other hand, ADALINE outperforms the traditional approach in yielding a good solution without being too computationally intensive. Although ADALINE is designed to yield only a local solution, by distributing simulation effort in line with the quality of the solution using an adaptive sampling framework, it quickly converges to the vicinity of a local minimum.

The simulation optimization approach provides flexibility in accommodating a variety of threshold based scheduling policies eliminating the need for formulating and solving an integer programming problem. The threshold levels are determined through a static optimization framework for the dynamic nature of decision making is implicitly handled through the scheduling policy. However, the dynamic and stochastic nature of the surgery rescheduling problem suggests several areas for future research, including, but not limited to: (i) a simulation optimization framework for dynamic decision making problems in the presence of uncertainty, particularly if an adaptive sampling framework can be implemented at various stages of the multi-stage problem, and (ii) a framework for obtaining global optimal solutions to the surgery rescheduling problem.
Chapter 5

Concluding Remarks

In this thesis, we develop ADALINE, a line search based algorithm to find a local minimum for continuous, and integer-ordered simulation optimization (SO). ADALINE addresses one of the important questions in SO regarding the choice of sample size through an adaptive sampling scheme. It does so by allocating the simulation effort based on measures of proximity to a local solution. ADALINE on continuous feasible sets mimics deterministic line search strategies, where the sample size at every iteration is determined based on sequential sampling schemes developed in Pasupathy et al. (2015). By trading off the estimation error (standard error on the estimated gradient) with the optimization error (norm of the estimated gradient), ADALINE eliminates the need for user defined sampling schemes that lead to inefficiency in traditional SO algorithms. Numerical results indicate that it consistently outperforms other adaptive strategies for continuous simulation optimization.

ADALINE on integer ordered feasible sets, alternates between a line search, and a neighborhood enumeration procedure to identify a local minimum. The adaptive line search exploits structure in the objective function whereas the neighborhood enumeration is a confirmatory test for a local minimum. In addition to the adaptive sample sizes, finite time performance of ADALINE in integer-ordered feasible regions is improved by a co-ordinate search following the neighborhood enumeration. Under certain structural conditions on the objective function, and with finiteness of the feasible set, we are able to show that ADALINE converges with probability one to the set of local minima. Extensive numerical implementation shows that ADALINE has better finite time performance than R-SPLINE that similarly uses a line search, and enumeration with fixed sample sizes at every iteration.

In the final chapter, we formulate and solve a surgical rescheduling problem to show the perfor-
mance of ADALINE on an operating room rescheduling problem. We devise a threshold based heuristic for rescheduling, and determine the optimal threshold levels using ADALINE. Our numerical results suggest that ADALINE finds threshold levels that result in lower total expected cost of rescheduling in comparison with Sample Average Approximation (SAA).

Some avenues for future investigation include further improving the finite time performance, and using ADALINE as the local search procedure of choice in random restart methods. ADALINE is currently designed to be a non-terminating algorithm, that is, it expends all available simulation budget on convergence to a local minimum. However, the algorithm can be terminated without expending all simulation effort when the probability of a solution being deemed a local minimum is at least \((1 - \alpha)\), akin to ranking and selection procedures. In addition, ADALINE, due to its finite time performance, is a good candidate for random restart methods to identify a global minimum. Extending ADALINE to simulation optimization problems with an equality constraint presents further challenges due to the requirement to redefine the neighborhood as any feasible solution is a local minimum with equality constrained problems.
Bibliography


Blake, John T and Donald, Joan. Mount sinai hospital uses integer programming to allocate operating room time. Interfaces, 32(2):63–73, 2002.


Appendix A

ADALINE for Continuous Simulation Optimization

```matlab
function [FunctionValue GradientValue
    GradientPercentile NumberofIterations
    NORMGradIteration ErrorIteration Effort SS Y
    Seeds]=StochasticOptimizationDARTS(
    NumberGenerations,TotalBudget,StartingPoint,Seed
    ,str,linesearch,BudgetPoint,Reduction,
    MaxOptionSet,EpsilonMaxOption)

% ENTER NUMBER OF GENERATIONS (ANY EVEN NUMBER)>1)

n=length(StartingPoint);
Increment=0;

% GENERATING SEEDS Pierre L'Ecuyer's Random Number
% Generator
SeedI(1)=Seed;
for i=1:5
    [SeedI(i+1) ~]=u16807d(SeedI(i));
end
for i=1:2:NumberGenerations−1
    [SeedI,~]=mrg32k3a(SeedI);
end
Seeds(i)=SeedI(3);
Seeds(i+1)=SeedI(6);
parfor i=1:NumberGenerations
    % Parallized Loop for each call of the DARTS
    CallNumber,TotalBudget1,StartingPoint,Seed,str,
    linesearch,OPTION_GENERATOR,BudgetPoint,
    ReductionAvailable,MinSamplingOptionSet,EpslnSam
    ,SamplingFactor,ThetaFactor
    [FunctionValue(i,:),GradientValue(i,:),Result{i}]=
    SODARTS(i,TotalBudget,StartingPoint,Seeds(i),str
    ,linesearch,1,BudgetPoint,Reduction,MaxOptionSet
    ,EpsilonMaxOption,0,0);
end

% Collection of results from individual runs and
% compiling them for plot generation
figure;
for i=1:NumberGenerations
```

78
NumberofIterations(i)=size(Result{i},1);
for k=1:n
    Y(1:NumberofIterations(i),Increment*n+k)=Result{i}(:,
            k);
end
NORMGradIteration(1:NumberofIterations(i),i)=Result{i}(:,
            n+2);
LastValue(i)=Result{i}(NumberofIterations(i),n+2);
Effort(1:NumberofIterations(i),i)=Result{i}(:,n+3);
SS(1:NumberofIterations(i),i)=Result{i}(:,n+4);
Increment=Increment+1;
end
for i=1:NumberGenerations
    SS(NumberofIterations(i)+1:size(NORMGradIteration,1),
            i)=Result{i}(NumberofIterations(i),n+4);
    NORMGradIteration(NumberofIterations(i)+1:size(
            NORMGradIteration,1),i)=LastValue(i);
    Effort(NumberofIterations(i)+1:size(NORMGradIteration
            ,1),i)=TotalBudget;
end
MaxNumber=min(50,floor(max(NumberofIterations)));
for j=1:MaxNumber
    ErrorIteration(j)=sum(NORMGradIteration(j,:)*sqrt(
            Effort(j,:))')/sum(sum(NORMGradIteration(j,:)
            ~=0));
    AverageSampleSize(j)=mean(SS(j,:));
end
SampleSizeMax=max(AverageSampleSize);
ErrorIteration=ErrorIteration';
PercentileValues=[25,50,75,90,95];
for i=1:length(PercentileValues)
    for j=1:length(BudgetPoint)
        GradientPercentile(i,j)=prctile(GradientValue(:,j),
                PercentileValues(i));
    end
    subplot(1,2,1);
    plot(BudgetPoint,GradientPercentile(i,:),'-x',
            'linewidth',1.5);
    hold all;
end
% GENERATION OF PLOT
grid on;
xlabel('Budget');ylabel('\|\nabla f\| at last iteration');title('Budget Vs \|\nabla f\| at last iteration');
legend('25th Percentile','50th Percentile','75th
Percentile','90th Percentile','95th Percentile')

subplot(1,2,2);
[ax,p1,p2]=plotyy(1:MaxNumber,ErrorIteration,1:
            MaxNumber,AverageSampleSize);
set(p1,'LineWidth',2.5);
set(p2,'LineWidth',1.5);
xlabel('Iteration');ylabel(ax(1),'\sqrt{W_{k}} \times \|\nabla f\| at Iteration');
ylabel(ax(2),'M_{k}');
set(ax(1),'YLim',[0 50]);
set(ax(2),'YLim',[0:7500:SampleSizeMax]);
grid on;
title('Iteration Vs \sqrt{W_{k}} \times \|\nabla f\| at Iteration');
annotation('textbox',
            [0.739,0.748,0.14568081991215204,0.111827956989],
            'String',
            {'X_{0}:[5,8]','TOTAL BUDGET:200000','NO. INDEPENDENT RUNS: 50'});
hold off;
savefig('OutputFigure.fig');
save('OutputFile.mat','FunctionValue', 'GradientValue '
            ', 'GradientPercentile', 'NumberofIterations', '
            'NORMGradIteration', 'ErrorIteration', 'Effort',
            'Y', 'Seeds');
end

function [OutputFunctionValue,OutputGradientValue, Result,H] = SODARTS(CallNumber,TotalBudget1,
StartingPoint, Seed, str, linesearch, 
OPTION_GENERATOR, BudgetPoint, ReductionAvailable, 
MinSamplingOptionSet, EpslnSam, SamplingFactor, 
ThetaFactor)

global TotalBudget M S fhandle X SampleSize Iteration 
Flag EF EGRADF UtilizedBudget H RandomSeed n;

TotalBudget=TotalBudget1;
MaxIterationNumber=1000;

if (MinSamplingOptionSet==0)
EpslnSam=−1;
end

% NUMBER OF DIMENSIONS IN THE PROBLEM
n=length(StartingPoint);

% CHOOSE THE FUNCTION FOR THE ORACLE & DIMENSION OF 
PROBLEM
if (strcmp(str,'AluffiPentini'))
M=1;
S=sqrt(0.1);
elseif(strcmp(str,'Rosenbrock'))
  % linesearch='Goldstein_BFGS';
  M=1;
  S=sqrt(0.1);
end
if strcmp(linesearch,'Goldstein_BFGS')
H=eye(n)^−1;
end
fhandle=str2func(str);
fhandleline=str2func(linesearch);

% PARAMETER INITIALIZATION
UtilizedBudget=0;
Iteration=0;
X(1,:)=StartingPoint;
Flag=1;
count=0;

% VECTOR INITIALIZATION
F=zeros(TotalBudget,1);
% Function realization at each sample (budget 
point)
GRADF=zeros(TotalBudget,n);
% Gradient realization at each sample (budget 
point)
EF=zeros(MaxIterationNumber);
% Estimated functional value at each iterate 
EGRADF=zeros(MaxIterationNumber,2);
% Estimated gradient value at each iterate 
SampleSize=zeros(MaxIterationNumber);
% Sample size at each iterate 
LineSearchSampleSize=zeros(MaxIterationNumber);
% Effort in Line Search 
NORMGRADF=zeros(MaxIterationNumber);
% Norm of the gradient at each iteration
CI=zeros(MaxIterationNumber);
% Half−width of the norm of the estimated 
gradient at each iteration
CumulativeBudget=zeros(MaxIterationNumber);
% Total effort expended until Iteration k 
TotalSampleSize=zeros(MaxIterationNumber);
TrueObjectiveSolution=zeros(MaxIterationNumber);
% True objective function value at kth iterate
TrueGradientSolution=zeros(MaxIterationNumber,n);
% True gradient at kth iterate
StepSize=zeros(MaxIterationNumber);
% Step size in the line search performed at kth 
iteration
EnteredSampling=zeros(MaxIterationNumber);
% Indicator variable that more sampling is 
performed at the current iteration

RandomSeed(1)=Seed;
Q=0;
CumulativeBudget(1)=0;

FileName=strcat('DARTS_',str,'_',linesearch, ' 
__ReductionOption=',int2str(ReductionAvailable),'
__Call=',int2str(CallNumber),'.txt');
fileID=fopen(FileName,'w');
fprintf(fileID,'% 3s % 11s % 11s % 11s % 11s % 11s %

80
while (UtilizedBudget<TotalBudget && Flag==1)
    % COMPUTE THE TRUE FUNCTIONAL VALUE AT EACH ITERATE
    [TrueObjectiveSolution(Iteration+1),
     TrueGradientSolution(Iteration+1,:)]=
        ComputeTrueValue(X(Iteration+1,:),M,S,str);

    NORMTRUEGRADIENT(Iteration+1)=norm(
        TrueGradientSolution(Iteration+1,:));

% INCREMENT THE ITERATION AND SET THE INITIAL
% SAMPLE SIZE TO 0

    Iteration=Iteration+1;
    LineSearchSampleSize(Iteration)=0;
    EnteredSampling(Iteration)=0;
    SampleSize(Iteration)=0;
    T=0;

    % if (Iteration<=2)
    %      fhandleline=str2func('Goldstein');
    % else
    %      fhandleline=str2func(linesearch);
    % end

% INITIATING THE PROCESS FOR EACH ITERATION (MEET
% MINIMUM SAMPLE SIZE REQUIREMENT OR USE VALUES
% FROM PREVIOUS ITERATE)

if (Iteration==1)
    for i=1:2
        [F(i),GRADF(i,:),RandomSeed(i+1)]=fhandle(X(Iteration:

        SampleSize(Iteration)=SampleSize(Iteration)+1;
        UtilizedBudget=UtilizedBudget+1;
    end

    EF(Iteration)=mean(F(1:2));
    EGRADF(Iteration,:)=sum(GRADF(1:2,:))/2;
    NORMGRADF(Iteration)=norm(EGRADF(Iteration,:));
    CI(Iteration)=1/sqrt(2)*ThetaFactor+tinv(0.975,1)*
        sqrt(norm(var(GRADF(1:2,:)),1)))/2^(0.5+SamplingFactor);
else
    SampleSize(Iteration)=SampleSize(Iteration−1);
    EF(Iteration)=mean(F(UtilizedBudget−SampleSize( Iteration)+1:UtilizedBudget));
    EGRADF(Iteration,:)=sum(GRADF(UtilizedBudget−
        SampleSize(Iteration)+1:UtilizedBudget,:))/
        SampleSize(Iteration);
    NORMGRADF(Iteration)=norm(EGRADF(Iteration,:));
    CI(Iteration)=1/sqrt(SampleSize(Iteration))*
        ThetaFactor+tinv(0.975,SampleSize(Iteration)−1)*
        sqrt(norm(var(GRADF(UtilizedBudget−SampleSize( Iteration)+1:UtilizedBudget,:)),1))/(SampleSize( Iteration))^0.5+SamplingFactor);
end

% INCREASE SAMPLING UNTIL $C \frac{\hat{\sigma}_{m_k}(X_k)}{||\nabla \hat{f}(X_k)||}\leq \frac{1}{C}
    % while (CI(Iteration)>=(NORMGRADF(Iteration)) && Flag ==1)
        EnteredSampling(Iteration)=1;
    if(UtilizedBudget==TotalBudget)
        Flag=0;
        break;
    else
        SampleSize(Iteration)=SampleSize(Iteration)+1;
        UtilizedBudget=UtilizedBudget+1;
        [F(UtilizedBudget),GRADF(UtilizedBudget,:),RandomSeed
            (SampleSize(Iteration)+1)]=fhandle(X(Iteration:

    EF(Iteration)=mean(F(UtilizedBudget−SampleSize( Iteration)+1));
else
    SampleSize(Iteration)=SampleSize(Iteration)+1;
    UtilizedBudget=UtilizedBudget+1;
    [F(UtilizedBudget),GRADF(UtilizedBudget,:),RandomSeed
        (SampleSize(Iteration)+1)]=fhandle(X(Iteration:

    EF(Iteration)=mean(F(UtilizedBudget−SampleSize( Iteration)+1));
end
Iteration)+1:UtilizedBudget));
EGRADF(Iteration,:)=sum(GRADF(UtilizedBudget−
SampleSize(Iteration)+1:UtilizedBudget,:))/
SampleSize(Iteration);
NORMGRADF(Iteration)=norm(EGRADF(Iteration,:));
CI(Iteration)=1/sqrt(SampleSize(Iteration))∗
ThetaFactor+tinv(0.975,SampleSize(Iteration)−1)
∗sqrt(norm(var(GRADF(UtilizedBudget−SampleSize( Iteration)+1:UtilizedBudget,:)),1))/SampleSize( Iteration)^(0.5+SamplingFactor);
end
end

% REDUCE SAMPLE SIZE AT CURRENT ITERATION IF
% POSSIBLE
if (CI(Iteration)≤(NORMGRADF(Iteration)) &&
ReductionAvailable==1 && SampleSize(Iteration)>2 &&
EnteredSampling(Iteration)==0)
while (CI(Iteration)≤(NORMGRADF(Iteration)) &&
SampleSize(Iteration)−T>=2)
T=T+1;
EF(Iteration)=mean(F(UtilizedBudget−SampleSize( Iteration)+1:UtilizedBudget−T));
EGRADF(Iteration,:)=sum(GRADF(UtilizedBudget−
SampleSize(Iteration)+1:UtilizedBudget−T,:))/
SampleSize(Iteration)−T);
NORMGRADF(Iteration)=norm(EGRADF(Iteration,:));
% CI(Iteration)=1/sqrt(SampleSize( Iteration)−T)∗ThetaFactor+tinv(0.975,SampleSize( Iteration)−T)∗
sqrt(norm(var(GRADF(UtilizedBudget−SampleSize( Iteration)+1:UtilizedBudget−T,:)),1))/
(SampleSize(Iteration)+1−T)^^(0.5+SamplingFactor) ;
if MinSamplingOptionSet==0
SampleSize(Iteration)=SampleSize(Iteration)−T+1;
end
end

% SET THE SAMPLE SIZE TO BE THE MAX (K^(1+EPSILON),
M_K)
if MinSamplingOptionSet==1
if EnteredSampling(Iteration)==1
NewSampleSize=max(floor(Iteration^(1+EpslnSam)),
SampleSize(Iteration));
else
NewSampleSize=max(floor(Iteration^(1+EpslnSam)),
SampleSize(Iteration)+1−T);
end
if (NewSampleSize > SampleSize(Iteration))
count1=0;
for i=1:NewSampleSize−SampleSize(Iteration)
count1=count1+1;
[F(UtilizedBudget+i),GRADF(UtilizedBudget+i,:),
RandomSeed(SampleSize(Iteration)+count1+1)]=
fhandle(X(Iteration,:),RandomSeed(SampleSize( Iteration)+count1),M,S);
end
UtilizedBudget=UtilizedBudget+NewSampleSize−
SampleSize(Iteration);
EF(Iteration)=mean(F(UtilizedBudget−NewSampleSize+1:
UtilizedBudget));
EGRADF(Iteration,:)=sum(GRADF(UtilizedBudget−
NewSampleSize+1:UtilizedBudget,:))/(
NewSampleSize);
NORMGRADF(Iteration)=norm(EGRADF(Iteration,:));
CI(Iteration)=1/sqrt(SampleSize(Iteration)+1)*
ThetaFactor+tinv(0.975,NewSampleSize−1)*sqrt(

$\text{var(GRADF(UtilizedBudget−NewSampleSize+1:
UtilizedBudget,:)),1}$)/(NewSampleSize)^0.5+
SamplingFactor);
SampleSize(Iteration)=NewSampleSize;
end
% LINE SEARCH TO DETERMINE THE NEXT ITERATE
if (Flag==1 && TotalBudget>=UtilizedBudget+SampleSize
(Iteration))
[LF,LGRADF,LineSearchSampleSize(Iteration),StepSize(Iteration)]=fhandleline(1);
for i=1:LineSearchSampleSize(Iteration)
F(UtilizedBudget+i,1)=LF(i,1);
GRADF(UtilizedBudget+i,:)=LGRADF(i,:);
end
UtilizedBudget=UtilizedBudget+LineSearchSampleSize
(Iteration);
else
LineSearchSampleSize(Iteration)=TotalBudget−
UtilizedBudget;
StepSize(Iteration)=1;
UtilizedBudget=UtilizedBudget+LineSearchSampleSize
(Iteration);
Flag=0;
end
CumulativeBudget(Iteration)=UtilizedBudget;
TotalSampleSize(Iteration)=SampleSize(Iteration)+
LineSearchSampleSize(Iteration);

% PRINT THE RESULTS AT EACH ITERATION
fprintf(fileID,'% 3d % 11.5f % 11.5f % 11.5f % 11.5f %
11.5f % 11.5f % 11.5f % 8d % 8d % 8.4f % 8d\n

n','...
OutputValue=OPTIMALVALUE';
OutputFunctionValue=TRUEFUNCTION;
OutputGradientValue=TRUEGRADIENT;
OutputSeed=RandomSeed(SampleSize(Iteration)+1);
fclose(fileID);

% PLOT THE TRAJECTORY OF THE ALGORITHM OVER A SINGLE
% RUN WHEN IT IS NOT CALLED BY A GENERATOR
if OPTION_GENERATOR==0
plottrajectory(str,M,S,X,Iteration,
LineSearchSampleSize(Iteration));
end
Result=[X(1:Iteration,:) TrueObjectiveSolution(1:
Iteration)' NORMTRUEGRADIENT(1:Iteration)' 
CumulativeBudget(1:Iteration)' SampleSize(1:
Iteration)'];
end

function[LF,LGRADF,SSL,alphatempout]=Goldstein_BFGS(
YY)
global TotalBudget M S fhandle X SampleSize Iteration 
Flag EF EGRADF RandomSeed UtilizedBudget H n;
% INITIALIZE THE ARRAYS THAT COLLECT INFORMATION 
% FROM EVERY BUDGET POINT
LF=[];
LGRADF=[];

% H=[2*(M^2+S^2)+1200*X(Iteration,1)^2*(M^4+3*S^4+6*M^2*S^2)
% -400*X(Iteration,2)*(M^2+S^2),
% -400*X(Iteration,1)*(M^2+S^2),-400*X(Iteration
% ,1)*M^2+5*S^2);200]^-1;
% INITIALIZE THE PARAMETERS
SSL=0;
Flag=1;
I=1;
a(1)=0;
b(1)=10^-4;
Condition=0;
bracketingmultiplier=2;

% PARAMETERS FOR THE LINE SEARCH METHOD (GOLDSTEIN 
% CONDITION PARAMETERS)
alpha=YY;
c=0.0001;

% INITIALIZE THE LINE SEARCH ITERATION AND THE 
% POINTS
alphatemp(I)=alpha;

% SAMPLE FROM THE INITIAL ALPHA(I) VALUE
LinearModelResult(I)=EF(Iteration)−c*alphatemp(I)*
(EGRADF(Iteration,:)*EGRADF(Iteration,:))';
LinearModelResultNegative(I)=EF(Iteration)−(1−c)∗
alphatemp(I)*(EGRADF(Iteration,:)*EGRADF(
Iteration,:))';
XL(I,:)=X(Iteration,:)=alphatemp(I)*(H+EGRADF{
Iteration,:})';
for k=1:SampleSize(Iteration)
SSL=SSL+1;
[FEndingPoint(k,1),GRADFEndingPoint(k,:),~]=fhandle(
XL(I,:),RandomSeed(k),M,S);
if ((UtilizedBudget+SSL)>=TotalBudget)
Flag=0;
break;
end
end
FXL(I)=mean(FEndingPoint);
GRADFLX(I,:)=sum(GRADFEndingPoint)/SampleSize(
Iteration);
LF=[LF;FEndingPoint];
LGRADF=[LGRADF;GRADFEndingPoint];
% Wolfe(I)=
% CHECK IF THE FIRST RULE IS SATISFIED BY THIS POINT
while(Condition==0 && Flag==1 && alphatemp(I)>10^-7)
% && alphatemp(I)>10^-8
if(FXL(I)<=LinearModelResult(I))
if (FXL(I)>=LinearModelResultNegative(I))
Condition=1;
break;
else
I=I+1;
a(I)=alphatemp(I-1);
b(I)=b(I-1);
if (b(I)==b(I))
alphanew=alpha(I)+b(I)/2;
else
alphatemp(I)=bracketingmultiplier*alphatemp(I-1);
end
LinearModelResult(I)=EF(Iteration)−c*alphatemp(I)*EGRADF(Iteration,:)*EGRADF(Iteration,:); % the linear model result
LinearModelResultNegative(I)=EF(Iteration)−(1−c)*alphatemp(I)*EGRADF(Iteration,:)*EGRADF(Iteration,:); % the linear model result negative
end
for k=1:SampleSize(Iteration)
SSL=SSL+1;
FEndingPoint(k,1),GRADFEndingPoint(k,:)]=fhandle(XL(I,:),RandomSeed(k),M,S);
if ((UtilizedBudget+SSL)>=TotalBudget)
Flag=0;
break;
end
end
FXL(I)=mean(FEndingPoint);
GRADFLX(I,:)=sum(GRADFEndingPoint)/SampleSize(Iteration);
end
end
alphatempout=alphatemp(I);
X(Iteration+1,:)=XL(I,:);
SK=−(alphatempout*H*EGRADF(Iteration,:))';
YK=GRADFLX(I,:)';
RHOK=1/(SK'*SK); % the linear model
if (Iteration==1)
H=(YK'*SK)/(YK'*YK)*H;
end
if YK'*SK>0
H=(eye(n)−RHOK*SK*YK')*H+(eye(n)−RHOK*YK*SK')*RHOK *YK')';
end
end
function[FX,GRADFL,SSL,alphatempout]=Goldstein(YY)
global TotalBudget M S fhandle X SampleSize Iteration
Flag EF EGRADF RandomSeed UtilizedBudget;
% INITIALIZE THE ARRAYS THAT COLLECT INFORMATION FROM EVERY BUDGET POINT
LF=[];
LGRADF=[];
% INITIALIZE THE PARAMETERS
SSL=0;
Flag=1;
I=1;
a(I)=0;
b(I)=10^-4;
Condition=0;
bracketingmultiplier=2;

% PARAMETERS FOR THE LINE SEARCH METHOD (GOLDSTEIN
CONDITION PARAMETERS)
alpha=YY;
c=0.0001;

% INITIALIZE THE LINE SEARCH ITERATION AND THE
POINTS
alphatemp(I)=alpha;

% SAMPLE FROM THE INITIAL ALPHA(I) VALUE
LinearModelResult(I)=EF(Iteration)−c*alphatemp(I)*
(EGRADF(Iteration,:)*EGRADF(Iteration,:))';
LinearModelResultNegative(I)=EF(Iteration)−(1−c)*
alphatemp(I)*EGRADF(Iteration,:)*EGRADF(Iteration,:));
XL(I,:)=X(Iteration,:)
−alphatemp(I)*EGRADF(Iteration,:);
for k=1:SampleSize(Iteration)
SSL=SSL+1;
[FEndingPoint(k,1),GRADFEndingPoint(k,:),~]=fhandle(XL(I,:),RandomSeed(k),M,S);
if ((UtilizedBudget+SSL)>=TotalBudget)
Flag=0;
break;
else
I=I+1;
a(I)=alphatemp(I−1);
b(I)=b(I−1);
if (b(I)<b(I))
alphatemp(I)=(a(I)+b(I))/2;
else
alphatemp(I)=bracketingmultiplier*alphatemp(I−1);
end
LinearModelResult(I)=EF(Iteration)−c*alphatemp(I)*
(EGRADF(Iteration,:)*EGRADF(Iteration,:))';
LinearModelResultNegative(I)=EF(Iteration)−(1−c)*
alphatemp(I)*EGRADF(Iteration,:)*EGRADF(Iteration,:));
XL(I,:)=X(Iteration,:)
−alphatemp(I)*EGRADF(Iteration,:);
for k=1:SampleSize(Iteration)
SSL=SSL+1;
[FEndingPoint(k,1),GRADFEndingPoint(k,:),~]=fhandle(XL(I,:),RandomSeed(k),M,S);
if ((UtilizedBudget+SSL)>=TotalBudget)
Flag=0;
break;
end
end
FXL(I)=mean(FEndingPoint);
GRADFLX(I,:)=sum(GRADFEndingPoint)/SampleSize(Iteration);
LF=[LF;FEndingPoint];
LGRADF=[LGRADF;GRADFEndingPoint];
end

% CHECK IF THE FIRST RULE IS SATISFIED BY THIS POINT
while(Condition==0 && Flag==1 && alphatemp(I)>10^-7)
if (FXL(I)<=LinearModelResult(I))
if (FXL(I)>=LinearModelResultNegative(I))
\[ X(I,:) = X(\text{Iteration}, :) - \alpha(I) EGRADF(\text{Iteration}, :) \]

\[
\text{for } k = 1: \text{SampleSize(\text{Iteration})}
\text{SSL} = \text{SSL} + 1;
\text{[FEndingPoint(k,:), GRADFEndingPoint(k,:)] = fhandle(X(I,:), RandomSeed(k), M, S);} \]

\[
\text{if } ((\text{UtilizedBudget} + \text{SSL}) \geq \text{TotalBudget})
\text{Flag} = 0;
\text{break;}\]

\text{end}\]

\[
\text{FXL(I)} = \text{mean(FEndingPoint)};
\text{GRADFLX(I,:)} = \text{sum(GRADFEndingPoint)} / \text{SampleSize(\text{Iteration})};
\]

\[
\text{LF} = [\text{LF}; \text{FEndingPoint}];
\text{LGRADF} = [\text{LGRADF}; \text{GRADFEndingPoint}];
\]

\text{end}\]

\[
\text{alphatempout} = \alpha(I);\]

\[
X(\text{Iteration}+1,:) = XL(I,:);\]

\text{end}\]

\[
\text{function} [\text{ObjectiveValue, GradientValue}] = \text{ComputeTrueValue}(Y, M, S, \text{str})
\text{if } (\text{strcmp(str, 'AluffiPentini'))}
\text{ObjectiveValue} = 0.25 * Y(1)^4 * (M^4 + 6*M^2*(S)^2 + 3*(S)^4) - 0.5 * Y(1)^2 * (M^2 + (S)^2) + 0.1 * Y(1)*M + 0.5 * Y(2)^2;
\text{GradientValue}(1) = 4 * Y(1)^3 * (M^4 + 6*M^2*(S)^2 + 3*(S)^4) - Y(1)*(M^2 + (S)^2) + 0.1 * M;
\text{GradientValue}(2) = Y(2);
\text{NormGradientValue} = \text{norm(GradientValue)};
\text{else (strcmp(str, 'Rosenbrock'))}
\text{ObjectiveValue} = Y(1)^2 * (M^2 + (S)^2) + 1 - 2 * Y(1) * M + 100 * (Y(2)^2 - 2 * Y(1) + 4 * Y(1)^3 * (M^4 + 6*M^2*(S)^2 + 3*(S)^4) - 2 * Y(1)^2 * (M^2 + (S)^2));
\text{GradientValue}(1) = 2 * Y(1) * (M^2 + (S)^2) - 2 * M + 100 * (4 * Y(1)^3 * (M^4 + 6*M^2*(S)^2 + 3*(S)^4) - 4 * Y(1) * Y(2) * (M^2 + (S)^2));
\text{GradientValue}(2) = 200 * (Y(2)^2 - Y(1)^2) * (M^2 + (S)^2));
\text{NormGradientValue} = \text{norm(GradientValue)};
\end{verbatim}
% INPUT: X=[X_1,....,X_N] and the Seed from the Master problem.
% OUTPUT: The functional value and the true gradient at a realization and the next random seed.
% FUNCTION CALLS: Generate random variates.
% PARAMETERS: a,b,c,d
% Set Parameters
% A) Functional Parameters
a=0.25;b=0.5;c=0.1;d=0.5;
% B) Distributional Parameters for the Noise function
% M is the mean and Sg is the standard deviation of the normal random variable (when using other distributions, please change them and alter the parameters accordingly)
% Generate a random realization for the noise function
[Seed1 URV]=u16807d(Seed1); % URV= Uniform Random Variate
R=norminv(URV,M1,Sg1);
% R=1;
F1=a*(XX(1)*R)^4−b*(XX(1)*R)^2+c*XX(1)*R+d*XX(2)^2;
GRADFX1=4*a*(XX(1))^3*R^4−2*b*XX(1)*R^2+c*R;
GRADFX2=2*d*XX(2);
GRADF1=[GRADFX1,GRADFX2];
end

function [F1, GRADF1, Seed1] = Rosenbrock(XX,Seed1,M1,Sg1)
% INPUT: X=[X_1,....,X_N] and the Seed from the Master problem.
% OUTPUT: The functional value and the true gradient at a realization and the next random seed.
% FUNCTION CALLS: Generate random variates.
% PARAMETERS: a,b,c,d
% Set Parameters
% A) Functional Parameters
a=0.25;b=0.5;c=0.1;d=0.5;
% B) Distributional Parameters for the Noise function
% M is the mean and Sg is the standard deviation of the normal random variable (when using other distributions, please change them and alter the parameters accordingly)
% Generate a random realization for the noise function
[Seed1 URV]=u16807d(Seed1); % URV= Uniform Random Variate
R=norminv(URV,M1,Sg1);
% R=1;
F1=(XX(1)*R−1)^2+100*(XX(2)−(XX(1)*R)^2)^2;
GRADFX1=2*(XX(1)*R−1)*R−400*(XX(2)−(XX(1)*R)^2)*(XX(1)*R^2);
GRADFX2=200*(XX(2)−(XX(1)*R)^2);
GRADF1=[GRADFX1,GRADFX2];
end

function [iseed,u16807d]=u16807d(iseed)
%..........................................................................
% bruce schmeiser january 1989.
% a linear congruential pseudorandom number generator .
% using constant 16807 and modulus (2^31)−1.
% iseed = iseed ×16807 (mod 2^31 −1)
% in correct implementations, starting with a seed value of 1 will .
% result in a seed value of 1043618065 on call number 10001. 
% for implementations that don't require double precision, see .
% s.k. park and k.w. miller, "random numbers generators: good .
% ones are hard to find," cacm, 31, 10 (October 1988), 1192−1201. .
% in correct implementations, starting with a seed value of 1 will .
% result in a seed value of 1043618065 on call number 10001. 
% input: iseed. integer.
%..........................................................................
% chosen from [1,2147483646] on the first call. 
% thereafter, the value returned from the last call.
% output: iseed. integer.
% to be used in the next call.
% output: u16807d. real.
% a pseudorandom number in (0,1).

u16807d=0;
while (u16807d<=0 || u16807d>=1)
    iseed = mod (iseed * 16807,2147483647);
    u16807d = iseed / 2147483648;
end
end

function [seed,u] = mrg32k3a(seed)

s1 = 1403580; ....
t1 = 810728;
s2 = 527612;
t2 = 1370589;
m1 = 4294967087;
m2 = 4294944443;
m3 = 4294967088;
P1 = mod( ( s1 * seed(1) ) − t1 * seed(2) , m1);
pl = mod( ( s2 * seed(4) ) − t2 * seed(5) , m2);
z = mod( ( p1 − p2 ) , m3 );
if (.z > 0 )
    u = z / m3;
elseif ( z == 0 )
    u = m1 / m3;
end
seed(1) = seed(2);
seed(2) = seed(3); seed(3) = p1;
seed(4) = seed(5);
seed(5) = seed(6); seed(6) = p2;

end

Raghu Pasupathy June 2012.

This is an implementation of Pierre L’Ecuyer’s Random Number Generator, MRG32K3A. ("Good Parameters and Implementations for Combined Multiple Recursive Random Number Generators", Operations Research 47, pp. 159−164, 1999.)
Appendix B

ADALINE for Integer Simulation Optimization

```python
# DECLARE THE STATIC GLOBAL VARIABLES AND INITIALIZE THE PARAMETERS
class G:
    oracle_parameter = [0.0, 0.0]
dimension = 0
total_neighbors = 0
total_budget = 0
isseed = 0
isseed_perturb = 0
alpha = 0
min_sample = 0
delta = 0
iteration_max = 400
imax = 15
s = 1
c = 2

@staticmethod
def calculate_dimensions():
    G.dimension = G.oracle_parameter[0]
    G.total_neighbors = 2 * G.oracle_parameter[0]

class DecisionVariable:
    x = [[0 for i in range(G.dimension)] for j in range(G.iteration_max)]

class Dynamic:
    output_seed = []
    urv = []

class OracleClass:
    @staticmethod
```

90
```python
def multi_quadratic(x_input_oracle, total_sample, current_sample):
    feasibility = 1
    est_value = 0
    est_var = float('Inf')
    true_value = 1
    ind_values = [0] * total_sample
    rv = [0] * total_sample
    if all(-100 <= i <= 100 for i in x_input_oracle):
        for i in range(G.dimension):
            true_value += x_input_oracle[i] ** 2
        mu = G.oracle_parameter[1]
        sigma = G.oracle_parameter[2] * true_value
        for i in range(total_sample):
            seed = Dynamic.output_seed[current_sample + i]
            seed_generator = u16807d_randomnumber(seed)
            u = seed_generator[1]
            rv[i] = norm.ppf(u, loc=mu, scale=sigma)
            ind_values[i] = true_value + rv[i]
            Dynamic.output_seed[current_sample+i+1] = seed
        est_value = np.mean(ind_values)
        if not total_sample == 1:
            est_var = np.var(ind_values, ddof=1)
        else:
            feasibility = 0
    return feasibility, est_value, est_var, true_value, ind_values

@staticmethod
def bus_scheduling(x_input_oracle, total_sample, current_sample):
    feasibility = 1
    est_value = 0
    est_var = float('Inf')
    true_value = 1
    ind_values = [0] * total_sample
    if all(0 <= i < G.oracle_parameter[2] for i in x_input_oracle):
        for i in range(total_sample):
            time_sum = 0
            time_bus = 0
            seed = Dynamic.output_seed[current_sample + i]
            seed_generator = u16807d_randomnumber(seed)
            u = seed_generator[1]
            if 1 - u > 0:
                time_arrive = (-math.log(1.0 - u))/G.oracle_parameter[1]
            if time_arrive > time_bus:
                time_bus = G.oracle_parameter[2]
                for j in range(G.oracle_parameter[0]):
                    if time_arrive <= x_input_oracle[j] < time_bus:
                        time_sum += time_bus - time_arrive
                        ind_values[i] = time_sum
                        Dynamic.output_seed[current_sample+i+1] = seed
                        est_value = np.mean(ind_values)
                        if not total_sample == 1:
                            est_var = np.var(ind_values, ddof=1)
                        else:
                            feasibility = 0
                    return feasibility, est_value, est_var, true_value, ind_values
        @staticmethod
def inventory_optimization(x_input_oracle, total_sample, current_sample):
    feasibility = 1
```

91
est_value = 0
est_var = float('Inf')
true_value = 1
ind_values = [0] * total_sample
no_days = G.oracle_parameter[2]
mean_demand = G.oracle_parameter[1]
warm_up = 100
holding_cost = 1
ordering_cost = 32
per_unit_cost = 3
shortage_cost = 5
start_inventory = x_input_oracle[1]
initial_inventory = [[0 for i in range(no_days + warm_up + 1)] \\
    for j in range(total_sample)]
order_quantity = copy.deepcopy(initial_inventory)
order_period = copy.deepcopy(initial_inventory)
holding = copy.deepcopy(initial_inventory)
inventory_position = copy.deepcopy(initial_inventory)
if x_input_oracle[1] - x_input_oracle[0] >= 10 and \\
20 <= x_input_oracle[0] <= 80 and 40 <= x_input_oracle[1] <= 100:
    for i in range(total_sample):
        cost = 0
        np.random.seed(Dynamic.output_seed[current_sample + i])
        demand = np.random.poisson(mean_demand,no_days + warm_up)
        seed_generator = u16807d_randomnumber(Dynamic.output_seed[ \\
            current_sample + i])
        initial_inventory[i][0] = start_inventory
        for j in range(no_days + warm_up):
            if initial_inventory[i][j] < x_input_oracle[0]:
                order_quantity[i][j] = x_input_oracle[1] \\
                - initial_inventory[i][j]
                order_period[i][j] = 1
                inventory_position[i][j] = initial_inventory[i][j] \\
                + order_quantity[i][j]
            shortage[i][j] = max(0,demand[j] - inventory_position[i][j])
            holding[i][j] = max(0,inventory_position[i][j] - demand[j])
            initial_inventory[i][j+1] = holding[i][j]
            if j >= warm_up:
                cost += shortage[i][j] * shortage_cost + holding[i][j] \\
                + holding_cost + per_unit_cost + order_quantity[i][j] \\
                + order_period[i][j] * ordering_cost
                ind_values[i] = cost
                Dynamic.output_seed[current_sample + i + 1] = seed_generator[1]
            est_value = np.mean(ind_values)
            if not total_sample == 1:
                est_var = np.var(ind_values,ddof=1)
            else:
                feasibility = 0
            return feasibility, est_value, est_var, true_value, ind_values

def oracle_call(x_input_oracle,total_sample,current_sample,problem):
    z = getattr(OracleClass,problem)(x_input_oracle,total_sample,current_sample)
    return z[0],z[1],z[2],z[3].z[4]

# RANDOM SEED GENERATOR
def u16807d():
    seed = G.iseed
    for i in range(G.total_budget+100):
        urv = 0
        while urv <= 0 or urv >= 1:
```
165 temp = seed * 16807
166 seed = temp % 2147483647
167 urv = seed / 2147483648
168 Dynamic.output_seed[i] = seed
169 Dynamic.urv.append(urv)

171 # RANDOM NUMBER GENERATOR
172 def u16807d_randomnumber(seed):
173 urv = 0
174 while urv <= 0 or urv >= 1:
175 temp = seed * 16807
176 seed = temp % 2147483647
177 urv = seed / 2147483648
178 return urv, seed

182 # INTEGER ORDERED SIMULATION OPTIMIZATION USING ADAPTIVE SPLINE
183 def stochastic_integrated(x_input_opt, problem, algorithm_parameters, run):
184 # INITIALIZE ALL THE SOLVER PARAMETERS
185 budget = G.total_budget
186 escort = algorithm_parameters[0]
187 gradient = algorithm_parameters[1]
188 adaptive_PLI = algorithm_parameters[2]
189 adaptive_NE = algorithm_parameters[3]
190 sol.tolerance = 0
191 min_sample = G.min_sample
192
193 # RUN THE U16807D GENERATOR IF THE PROBLEM TO BE SOLVED IS THE
194 # MULTI DIMENSIONAL QUADRATIC PROBLEM
195 if problem == 'multi_quadratic':
196 # u16807d()
197
198 # CREATE AND NAME THE LOG FILE THAT IS GENERATED
199 filename = "GradientDescent_SeedXi" + str(run+1) + ".txt"
200 f = open(filename, 'w')
201
202 # INITIALIZE ALL THE VARIABLES CORRESPONDING TO ONE RUN OF THE
203 # SIMULATION OPTIMIZATION ALGORITHM
204 true_objective = []
205 iteration_Q = []
206 sample_size = []
207 Work = []
208 Magnitude = []
209 overall_sample = []
210 fx_iteration = []
211 search_direction = []
212 out_search_direction = []
213 pli_sample_size = []
214 Q = 0
215 ncalls = 0
216 iteration = 0
217 max_sample = min_sample
218 x_best = float('inf')
219 DecisionVariable.x[iteration] = x_input_opt
220 x_best = copy.deepcopy(DecisionVariable.x[iteration])
221 f.write("Iteration				" + "
222 + "Decision Variable at start				" + "
223 + "Best at end of iteration				"
224 + "Estimated Total Cost			" + "
225 + "Search Direction			" + "
226 + "Work	".rjust(15) + "" + "Overall Sample Size".rjust(200) + 
227 feasibility = oracle_call(x_best, 1.0, problem)
```
if feasibility[0] == 0:
    print "Enter a feasible solution to start algorithm"
    return

if adaptive_PLI == 0:
    search_sample_size = 0

# PERFORM ADAPTIVE PLI AND NEIGHBORHOOD ENUMERATION UNTIL THE BUDGET GETS EXPENDED
while ncalls < G.total_budget:
    # START THE ADAPTIVE PLI FRAMEWORK
    if adaptive_PLI == 1 and ncalls + (G.dimension + 1) * min_sample < G.total_budget:
        linear_interpolation = Adaptive_PLI(x_best, 
                                           min_sample, ncalls, problem)
        fx_pli_best = linear_interpolation[1]
        x_pli_best = linear_interpolation[0]
        search_sample_size = linear_interpolation[2]
        ncalls = linear_interpolation[3]
        pli_sample_size.append(search_sample_size)
    
    # MAKE THE SAMPLE SIZES EQUAL FOR COMPARISON PURPOSES
    estimate_best = oracle_call(x_best, search_sample_size, 0, problem)
    fx_best = estimate_best[1]
    ncalls += search_sample_size

    # UPDATE X_BEST IF PLI RETURNS A BETTER SOLUTION
    # THAN THE CURRENT BEST
    if fx_pli_best < fx_best + sol_tolerance:
        x_best = x_pli_best
        fx_best = fx_pli_best

    # PERFORM LINE SEARCH AT THE END OF THE ADAPTIVE PLI PROCEDURE
    x_temp = copy.deepcopy(x_best)
    x1 = copy.deepcopy(x_temp)
    i = 0
    while i < G.imax:
        s = G.s * (G.c ** i)
        magnitude = np.linalg.norm(np.array(search_gradient))
        for k in range(len(x1)):
            x1[k] = x_temp[k] - np.floor(search_gradient[k] / magnitude * s + 
                                         np.sign(search_gradient[k]) * 0.5)
        feasibility_check = oracle_call(x1, 1, 0, problem)
        if feasibility_check[0] > 0 and 
           ncalls + search_sample_size < G.total_budget:
            sampling = oracle_call(x1, search_sample_size, 0, problem)
            ncalls += search_sample_size
            if sampling[1] < fx_best + sol_tolerance:
                fx_best = sampling[1]
                x_best = copy.deepcopy(x1)
            else:
                # print "PLI Line Search yields . fx_best ." at ", . best. "
                break
        else:
            # print "PLI Line Search failed due to solution being inferior".
            "
            break
    
    else:
        # print "PLI Line Search failed due to infeasibility".
        break
    i += 1

elif ncalls == (G.dimension + 1) * min_sample > G.total_budget:
    # print "Budget Exceeded"
if adaptive_NE == 1:
    neighborhood = neighborhoodsearch(ncalls, iteration, x_best,
    min_sample, adaptive_NE, problem)
else:
    neighborhood = neighborhoodsearch(ncalls, iteration, x_best,
    search_sample_size, adaptive_NE, problem)

ncalls = neighborhood[0]
overall_sample.append(neighborhood[1])
sample_size.append(neighborhood[2])
search_direction.append(neighborhood[3])
fx_best = neighborhood[6]
linesearch_NE = neighborhood[10]
if neighborhood[6] < fx_best + sol_tolerance:
    fx_best = neighborhood[5]
x_best = neighborhood[4]

# PERFORM NEIGHBORHOOD ENUMERATION

# CONDUCT LINE SEARCH ALONG THE SEARCH DIRECTION AFTER NEIGHBORHOOD ENUMERATION IF SAMPLING CRITERION IS SATISFIED
if ncalls < G.total_budget and search_direction[iteration] >= 0 and len(search_direction[iteration]) >= 0:
    if len(search_direction[iteration]) >= 1:
        max_sample_temp = max(overall_sample)
        [search_direction[iteration][0]], sample_size
        x_temp = copy.deepcopy(x_best)
x1 = copy.deepcopy(x_temp)
search_vector = [0] * G.dimension
for k in range(len(search_direction[iteration])):
    if (search_direction[iteration][k] % 2) == 0:
        search_vector[search_direction[iteration][k] // 2] =
        - neighborhood[9][search_direction[iteration][k]]
    else:
        search_vector[(search_direction[iteration][k] // 2) -
        neighborhood[9][search_direction[iteration][k]]]

i = 0
if adaptive_NE == 1 or linesearch_NE == 1:
    while i < G.imax:
        s = G.s * (G.c ** i)
        if gradient == 0:
            x1 = map(sum.zip(x_temp, np.array(search_vector)) * s)
        else:
            magnitude = np.linalg.norm(np.array(search_vector))
            for k in range(len(x1)):
                x1[k] =
                x_temp[k] + np.floor(search_vector[k] / magnitude * s
                + np.sign(search_vector[k])) * 0.5)
        feasibility_check = oracle_call(x1, 1, 0, problem)
        if feasibility_check[0] > 0 and 
        ncalls + overall_sample[iteration][search_direction[iteration][0]]
        <= G.total_budget:
            sampling = 
            oracle_call(x1, overall_sample[iteration]
            [search_direction[iteration][0]], 0, problem)
            ncalls += overall_sample[iteration][search_direction
            [iteration][0]]
            if sampling[1] < fx_best + sol_tolerance:
                fx_best = sampling[1]
x_best = copy.deepcopy(x1)
                # print "NE Line Search yields ", fx_best , " at ", x_best, "\n"
            else:
                # print "NE Line Search fails due to inferior solution ", "\n"
break
    # print "WE Line Search fails due to infeasible solution". "\n"
break
i += 1

# PRINT THE SEARCH DIRECTION AT THE END OF NEIGHBORHOOD ENUMERATION
if search_direction[iteration] != -1000
    and len(search_direction[iteration]) > 0:
        out_search_direction.append([0]*len(search_direction[iteration]))
else:
    out_search_direction.append([10001])
if search_direction[iteration] > 0:
    for ii in range(len(out_search_direction[iteration])):
        if search_direction[iteration][ii] % 2 == 0:
            out_search_direction[iteration][ii] = search_direction[iteration][ii] / 2 + 1
        else:
            out_search_direction[iteration][ii] = -(search_direction[iteration][ii] + 1) / 2
    else:
        out_search_direction[iteration][0] = -1000
f.write(str(iteration).rjust(2) + " " + str(DecisionVariable.x[iteration][:]).rjust(80) + " " + str(int(fx_best)).rjust(80) + " " + str(out_search_direction[iteration]) + " " + str(ncalls) + " " + str(overall_sample[iteration]).rjust(200) + "\n\n")
fx_iteration.append(fx_best)
iteration += 1

# DISCARD USED SEEDS AT THE END OF EVERY ITERATION
if max_sample_temp <= max_sample:
    max_sample_generate = u16807d_randomnumber(
    Dynamic.output_seed[max_sample])
    Dynamic.output_seed[max_sample] = max_sample_generate[1]
else:
    max_sample = max(max_sample, max_sample_temp)
    Dynamic.output_seed[0] = Dynamic.output_seed[max_sample]
else:
    # INCREMENT MINIMUM SAMPLE SIZE IF THE ESCORT SEQUENCE IS ACTIVATED
    if escort == 1: min_sample = max(int(iteration**0.65), G.min_sample)
    DecisionVariable.x[iteration][:] = x_best[:]
    Work.append(ncalls)
    Magnitude.append(np.linalg.norm
        (np.array(DecisionVariable.x[iteration][:])) -
        np.array(DecisionVariable.x[iteration-1][:])))
    # TO FIGURE OUT THE TRUE OBJECTIVE FUNCTION AT THE END OF EACH ITERATION, REPLACE WITH ESTIMATED VALUES IF TRUE FUNCTION IS UNKNOWN
    if Work[iteration-1] >= G.budget_vector[0] and Q <= len(G.budget_vector):
        if iteration == 1:
            inds = indices(G.budget_vector, lambda x: x <= Work[iteration-1])
        else:
            inds = indices(G.budget_vector, lambda x: Work[iteration-2]
            < x <= Work[iteration-1])
    Q += len(inds)
    iteration_Q.append(iteration)
if iteration < 1:
    for l in range(len(inds)):
        objective_value = true_objective_quadratic(x_input_opt)
        true_objective.append(objective_value)
else:
    objective_value = true_objective_quadratic(
        DecisionVariable.x[iteration_Q[-1]-1])
    for I in range(len(ind)):
        true_objective.append(objective_value)
    if Q >= len(G.budget_vector):
        objective_value = true_objective_quadratic(DecisionVariable.x[iteration-1])
        true_objective[-1] = objective_value

f.close()
return true_objective, pli_sample_size, overall_sample, ncalls, search_direction, x_best

def indices(a, func):
    return [i for (i, val) in enumerate(a) if func(val)]

def neighborhoodsearch(ncalls, iteration, x, min_sample, adaptive_NE, problem):
    linesearch_NE = 0
    delta = G.delta
    incumbent_solution = x
    alpha = 0.85
    budget_limit = 0
    diff_zone = 0
    neighbor_solution = [[incumbent_solution[j] for j in range(G.dimension)]
    for i in range(G.total_neighbors)]
    individual_incumbent = []
samplesize_incumbent = min_sample
    samplesize = [0] * G.total_neighbors
    est_value_incumbent = [0] * G.total_neighbors
    estValue = [0] * G.total_neighbors
    est_var_incumbent = 1
    est_var = [1] * G.total_neighbors
    mean_diff = [0] * G.total_neighbors
    se_diff = [1] * G.total_neighbors
    coeff_diff = [-1] * G.total_neighbors
    norm_diff = [1] * G.total_neighbors
    sum_xy = [0] * G.total_neighbors
    sum_xx = [0] * G.total_neighbors
    cov_xy = [0] * G.total_neighbors
    sample_reduced = [0] * G.total_neighbors
    sample_reduced_indices = []
    sample_only = [0] * G.total_neighbors
    sample_only_indices = []
    reduced_only = [0] * G.total_neighbors
    reduced_only_indices = []
    secondary_priority = [0] * G.total_neighbors
    secondary_priority_indices = []
    indicesample = []
    infeasible_index = []
    x_best = [-1] * G.dimension
    fx_best = -1000

    # CREATE ALL THE CANDIDATE SOLUTIONS AND SAMPLE A MINIMUM FROM EACH CANDIDATE
    for i in range(G.total_neighbors):
        if (i % 2) == 0:
            neighbor_solution[i][i // 2] += 1
        else:
            neighbor_solution[i][(i - 1) // 2] += -1
        feasibility_check = oracle_call(neighbor_solution[i]::, 1, 0, problem)
        infeasible_index.append(feasibility_check[0])
    feasible_set = [i for i in range(G.total_neighbors) if infeasible_index[i] == 1]
for i in feasible_set:
    if ncalls + min_sample < G.total_budget:
        estimate_neighbor = oracle_call(neighbor_solution[i][:], min_sample, 0, problem)
        est_value[i] = estimate_neighbor[1]
        est_var[i] = estimate_neighbor[2]
        ncalls += min_sample
        sample_size[i] = min_sample
    else:
        indices_to_sample = [i]
        ncalls = G.total_budget + 1
        budget_limit = 1
        break
    if i == feasible_set[0]:
        estimate_incumbent = oracle_call.incumbent_solution, min_sample, 0, problem)
        est_value_incumbent = [estimate_incumbent[1]] * G.total_neighbors
        est_value_incumbent = estimate_incumbent[2]
        individual_incumbent.extend(estimate_incumbent[4])
        ncalls += min_sample
        mean_diff[i] = est_value_incumbent[i] - est_value[i]
        sum_xx[i] = np.dot(np.array(estimate_neighbor[4], dtype=object),
                          np.array(estimate_neighbor[4], dtype=object))
        sum_xy[i] = np.dot(np.array(individual_incumbent, dtype=float),
                          np.array(estimate_neighbor[4], dtype=object))
        cov_xy[i] = (sum_xy[i] - min_sample * est_value_incumbent[i] *
                     est_value[i]) / (min_sample - 1)
    # CATEGORIZE THE FEASIBLE NEIGHBORS INTO 1 OF 4 CATEGORIES BASED ON
    # THEIR ESTIMATED OBJECTIVE AND STD. ERROR
    if est_var[i] + est_var_incumbent - 2 * cov_xy[i] > 0:
        se_diff[i] = math.sqrt((est_var[i] + est_var_incumbent -
                                 2 * cov_xy[i]) / min_sample)
        coeff_diff[i] = mean_diff[i] / se_diff[i]
    else:
        se_diff[i] = 0
        coeff_diff[i] = mean_diff[i] / (10 ** (-5))
    norm_diff[i] = 1/stats.t.ppf(alpha, min_sample-1) * abs(mean_diff[i])
    if se_diff[i] > norm_diff[i] and mean_diff[i] > diff_zone:
        reduced_only[i] = 1
        reduced_only_indices.append(i)
    elif se_diff[i] < norm_diff[i] and mean_diff[i] > diff_zone:
        sample_reduced[i] = 1
        sample_reduced_indices.append(i)
    elif se_diff[i] < norm_diff[i] and mean_diff[i] <= diff_zone:
        sample_only[i] = 1
        sample_only_indices.append(i)
    else:
        secondary_priority[i] = 1
        secondary_priority_indices.append(i)
# CHECK IF ADAPTIVE NEIGHBORHOOD ENUMERATION IS REQUIRED?
if len(sample_reduced_indices) > 0 and adaptive_NE == 0:
    linesearch_NE = 1
    list_direction = [i for i in sample_reduced_indices if mean_diff[i] > diff_zone]
e else len(reduced_only_indices) > 0 and adaptive_NE == 0:
    list_direction = [i for i in reduced_only_indices if mean_diff[i] > diff_zone]
# PERFORM ADAPTIVE NE PROCEDURE WHEN THERE IS NO DESCENT DIRECTION TO HOP ON TO
if len(reduced_only_indices) == 0 or adaptive_NE == 1:
    linesearch_NE = 1
# PERFORM ADAPTIVE NE PROCEDURE
while ncalls <= G.total_budget and budget_limit == 0
and len(sample_reduced_indices) == 0:

98
samplefromincumbent = 0
extr samp - [0] = G. total_neighbors
if \(\text{len}(\text{reduced\_only\_indices}) + \text{len}(\text{secondary\_priority\_indices}) > 0:\)
    indices to sample = reduced\_only\_indices[] + secondary\_priority\_indices[]
    r condition = 1
else:
    indices to sample = sample\_only\_indices[]
    r condition = 3
for indices in templist - indices to sample:
    if max(extr samp) == 0:
        indices to sample = templist[]
    if \(\text{len}(\text{sample\_only\_indices}) = \text{len}(\text{feasible\_set}):\)
        indices to sample = sample\_only\_indices[]
        r condition = 3
for j in templist:
    extr samp[j] = samplesize\_incumbent - samplesize[j]
    if extr samp[j] == 0:
        indices to sample.remove(j)
    if max(extr samp) == 0:
        indices to sample = templist[]
    for k in indices to sample:
        extr samp[k] = delta
    samplesize\_incumbent = 1
    if samplefromincumbent == 1:
        new\_sample\_incumbent = oracle\_call(incumbent\_solution, delta, sample\_size\_incumbent, problem)
        individual\_incumbent.extend(new\_sample\_incumbent[4])
        samplesize\_incumbent += delta
    n calls += delta
    current\_estimate = np.mean(individual\_incumbent)
# SAMPLE ALL THE NEIGHBORS THAT DO NOT SATISFY THE SAMPLING CRITERION
for j in indices to sample:
    if \(\text{n calls} + \text{extr samp[j]} < \text{G.total\_budget}\):
        new\_sample\_neighbor = oracle\_call(neighbor\_solution[j][:], extr samp[j], samplesize[j], problem)
    else:
        n calls += G.total\_budget
        break
    sum\_xx[j] += np.dot(np.array(new\_sample\_neighbor[4], dtype=object), np.array(new\_sample\_neighbor[4], dtype=object))
    sum\_xy[j] += np.dot(np.array(individual\_incumbent[-extr samp[j]:,], dtype=object), np.array(individual\_incumbent[-extr samp[j]:,], dtype=object))
    est\_value[j] = (est\_value[j] * samplesize[j] + np.mean(new\_sample\_neighbor[4]) * extr samp[j]) / (samplesize[j] + extr samp[j])
    n calls += extr samp[j]
    samplesize[j] += extr samp[j]
    est\_value\_incumbent[j] = np.mean(individual\_incumbent[0:samplesize[j]])
    mean\_diff[j] = est\_value\_incumbent[j] - est\_value[j]
    if est\_var[j] + est\_var\_incumbent - 2 * cov\_xy[j] > 0:
        se\_diff[j] = math.sqrt((est\_var[j] + est\_var\_incumbent - 2 * cov\_xy[j]) / samplesize\_incumbent)
    else:
        se\_diff[j] = 0
    coeff\_diff[j] = mean\_diff[j] / se\_diff[j]
    norm\_diff[j] = 1 / stats.t.ppf(alpha, samplesize\_incumbent - 1) * abs(mean\_diff[j])
# UPDATE THE NEIGHBORS BASED ON THE NEW INFORMATION AFTER SAMPLING

if se_diff[j] <= norm_diff[j] and mean_diff[j] > diff_zone:
    sample_reduced[j] = 1
    sample_reduced_indices.append(j)

if se_diff[j] == 0:
    coeff_diff[j] = mean_diff[j] / (10**(-7))

elif se_diff[j] <= norm_diff[j] and mean_diff[j] <= diff_zone and rcondition != 3:
    sample_only[j] = 1
    sample_only_indices.append(j)

elif se_diff[j] > norm_diff[j] and rcondition != 3:
    coeff_diff[j] = mean_diff[j] / (10**(-7))

elif se_diff[j] <= norm_diff[j] and mean_diff[j] <= diff_zone and rcondition != 3:
    sample_only[j] = 1
    sample_only_indices.append(j)

else:
    if se_diff[j] < norm_diff[j] and rcondition != 3:
        sample_only[j] = 1
        sample_only_indices.append(j)

    if mean_diff[j] < diff_zone and j in reduced_only_indices:
        secondary_priority[j] = 1
        secondary_priority_indices.append(j)
        reduced_only[j] = 0
        reduced_only_indices.remove(j)

elif mean_diff[j] > diff_zone and j in secondary_priority_indices:
    reduced_only[j] = 0
    secondary_priority_indices.remove(j)

else:
    if se_diff[j] < norm_diff[j] and rcondition != 3:
        sample_only[j] = 0
        sample_only_indices.remove(j)

        if rcondition == 3:
            if se_diff[j] > norm_diff[j] and mean_diff[j] <= diff_zone:
                secondary_priority[j] = 1
                secondary_priority_indices.append(j)
                sample_only[j] = 0
                sample_only_indices.remove(j)

            elif se_diff[j] > norm_diff[j] and mean_diff[j] > diff_zone:
                reduced_only[j] = 1
                reduced_only_indices.append(j)
                sample_only[j] = 0
                sample_only_indices.remove(j)

        elif se_diff[j] < norm_diff[j] and mean_diff[j] > diff_zone:
            reduced_only[j] = 1
            reduced_only_indices.append(j)
            sample_only[j] = 0
            sample_only_indices.remove(j)

        elif se_diff[j] < norm_diff[j] and mean_diff[j] <= diff_zone:
            sample_only[j] = 0
            sample_only_indices.remove(j)

            list_direction = [i for i in sample_reduced_indices if coeff_diff[i] > 0]

            max_sample = np.argmax(samplesize)

            # UPDATE X_BEST AND FX_BEST

            if len(list_direction) > 0:
                search_direction = list_direction
                x_best = neighbor_solution[search_direction[np.argmax([mean_diff[i] for i in list_direction])]]

                fx_best = est_value[search_direction[np.argmax([mean_diff[i] for i in list_direction])]]

            print fx_best

            elif mcalls > G.total_budget:
                budget_limit = 1
                search_direction = -1000
                fx_best = min(est_value_incumbent[indices_sample[0]], min(est_value))
else:
    search_direction = -1
    x_best = incumbent_solution
# print search_direction, x_best, "\n"
return ncalls, sample_size, sample_size_incumbent.

search_direction, x_best, fx_best, est_value_incumbent[max_sample],
neighbor_solution, budget_limit, mean_diff, linesearch_NE

def Adaptive_PLI(x_in, sample_size, ncalls, problem):
    delta = 1
    alpha = G.alpha
    x_best = x_in
    infeasible_simplex = 0
    d = G.dimension
    x = [[0 for i in range(d)] for j in range(d+1)]
    grad = [0] * d
    var_grad = [0] * d
    sum_xx = [0] * (d+1)
    w = [0] * (d+1)
    fnx = [10000000000000000] * (d+1)
    varfnx = [0] * (d+1)
    npoints = 0
    fbar = 0
    seed = u16807d_randomnumber(G.iseed_perturb)
    np.random.seed(G.iseed_perturb)
    perturb_urv = np.random.uniform(0, 1, d)
    G.iseed_perturb = seed[1]
    pert = [x_in[i] + 0.3*(perturb_urv[i]-0.5) for i in range(d)]
    x[0] = np.floor(pert)
    z = pert - x[0]
    z = np.insert(z, 0, 1)
    z = np.insert(z, len(z), 0)
    p = np.argsort(z)
    for i in range(1, d+1):
        x[i][i-1] = x[i-1][i-1] + 1
        xi = [z[i-1] - z[p[len(p)-i]] - z[p[len(p)-i-1]]
        x[i] = z[p[i]] - z[p[0]]
        if [x_in in x[i] for i in range(d+1)):
            npoints += 1
        # SAMPLE FROM EVERY POINT IN THE (d+1) SIMPLEX
        for i in range(d+1):
            feasibility_check = oracle_call(x[i], 1, 0, problem)
            if feasibility_check[0] == 1:
                npoints += 1
                estimate_x = oracle_call(x[i], sample_size, 0, problem)
                fnx[i] = estimate_x[1]
                sum_xx[i] = np.dot(np.array(estimate_x[4], dtype=object),
                                   np.array(estimate_x[4], dtype=object))
                varfnx[i] = (sum_xx[i] - (sample_size * fnx[i] ** 2)) / (sample_size - 1)
                fbar += w[i] * fnx[i]
                if i > 0:
                    sum_xy[i-1] = np.dot(np.array(estimate_x[4], dtype=object),
                                         np.array(old_collection, dtype=object))
                    cov_grad[i-1] = (sum_xy[i-1] - sample_size * fnx[i] * fnx[i-1]) / (sample_size - 1)
                    grad[p[len(p)-i-1]-1] = fnx[i] - ghatold

101
```python
var_grad[p[len(p)-i-1]-1] = varfnx[i] + varghatold - 2 * cov_grad[i-1]

ghatold = fnx[i] 
varghatold = varfnx[i]
old_collection = estimate_x[4]

else:
    npoints = 0
    grad = [0] * d
    fx_pli_best = 100000000
    ghatold = 0
    infeasible_simplex = 1
    break

se_grad = math.sqrt(np.linalg.norm(var_grad))/math.sqrt(sample_size)
norm_grad = 1/stats.t.ppf(alpha, sample_size-1) * np.linalg.norm(grad)
calls += npoints * sample_size

# ADAPTIVELY SAMPLE UNTIL THE SAMPLING CRITERION IS SATISFIED
while se_grad > norm_grad and calls <= G.total_budget + 1:
    for i in range(d+1):
        new_x = oracle_call(x[i], delta, sample_size, problem)
        fnx[i] = (fnx[i] * sample_size + np.sum(new_x[4]))
        / (sample_size + delta)
        sum_xx[i] += np.dot(np.array(new_x[4], dtype=object), np.array(new_x[4], dtype=object))
        varfnx[i] = (sum_xx[i] - ((sample_size + delta) * fnx[i] ** 2))
        / (sample_size + delta - 1)
        sbar += v[i] * fnx[i]
        if i > 0:
            sum_xy[i-1] += np.dot(np.array(new_x[4], dtype=object), np.array(old_collection, dtype=object))
            cov_grad[i-1] = (sum_xy[i-1] - (sample_size + delta) *
            fnx[i] * fnx[i-1])/(sample_size + delta -1)
            grad[p[len(p)-i-1]-1] = fnx[i] - ghatold
            var_grad[p[len(p)-i-1]-1] = varfnx[i] + varghatold - 2 * cov_grad[i-1]
            ghatold = fnx[i]
            varghatold = varfnx[i]
            old_collection = new_x[4]
        sample_size += delta

se_grad = math.sqrt(np.linalg.norm(var_grad))/math.sqrt(sample_size)
norm_grad = 1/stats.t.ppf(alpha, sample_size-1) * np.linalg.norm(grad)
calls += npoints * delta

if infeasible_simplex == 0:
    fx_pli_best = np.min(fnx)
    x_best = x[np.argmin([fnx[i] for i in range(d+1)])]
return x_best, fx_pli_best, grad, sample_size, calls

# CALCULATE THE TRUE OBJECTIVE SOLUTION FOR THE BUS SCHEDULING PROBLEM

def true_objective_bus(x_in):
    sort_x = np.sort(x_in)
    true_solution = G.oracle_parameter[1] * (sort_x[0] ** 2)/2
    for i in range(1,G.oracle_parameter[0]):
        true_solution += G.oracle_parameter[1] * ((sort_x[i] - sort_x[i-1]) ** 2)/2
    print true_solution
    return true_solution

# CALCULATE THE TRUE OBJECTIVE SOLUTION FOR THE QUADRATIC MINIMIZATION PROBLEM

def true_objective_quadratic(x_in):
    true_solution = np.dot(x_in,x_in) + 1
    return true_solution
```
Appendix C

Surgery Scheduling Problem

Scenario Generation for Surgery requests

This code generates the scenarios for surgical requests and their duration across the entire time horizon. Surgical requests include tentatively scheduled elective surgeries, realized semi-urgent surgeries and emergency surgeries.

# input parameters:
1. seed - seed(0),seed(1),seed(2) positive integers < m1
   , not all zero.
2. time_horizon - length of the time horizon
3. no_specialties - Total number of specialties considered
4. elective_surgeries
   semi_urgent_surgeries - parameters of the distribution of the number of elective, semi-urgent
   urgent_surgeries and emergency surgeries
5. duration - parameters of the distribution of elective, semi-urgent and emergency case duration
from _future_ import division
import numpy as np
import math
import sys
from scipy.stats import norm
from scipy import stats
import pandas as pd
import scipy.io
import copy
import matplotlib.pyplot as plt
import seaborn as sns

class parameters:
    seed = [11232, 1656, 46673, 6687897, 5643466, 35969]  # Seed for Pierre L'Ecreyuer's Random Number Generator (MRG32K3A)
total_budget = 1000
time_horizon = 5
no_specialties = 4
# dist_semi = 'Poisson'
# dist_duration = 'uniform'
dist_no_semi = 'negative_binomial'
dist_duration = 'log_normal'
z0 = [[10, 10, 10, 10], [10, 10, 10, 10], [10, 10, 10, 10], [10, 10, 10, 10]]
elective_surgeries = [[8, 8, 8, 8], [8, 8, 8, 8], [8, 8, 8, 8], [8, 8, 8, 8]]
semi_urgent_poison = [[3, 3, 3, 3], [3, 3, 3, 3], [3, 3, 3, 3], [3, 3, 3, 3], [3, 3, 3, 3]]
semi_urgent_nb = [[0.8, 3], [0.7, 3], [0.65, 3], [0.75, 3], [0.75, 3], [0.9, 3], [0.95, 3], [0.63, 3], [0.63, 3], [0.63, 3], [0.63, 3], [0.63, 3]]
semi_urgent_duration_log_normal = [[0.05, 1], [0.25, 0.55], [0.6, 0.3], [0.2, 0.65]]
semi_urgent_duration_uniform = [[1, 1.5], [1.3, 1.7], [1.4, 2.0], [1.5, 2.3]]
duration_log_normal = [[0.83, 0.73, 0.63, 0.26, 0.21], [0.83, 0.73, 0.63, 0.26, 0.21], [0.83, 0.73, 0.63, 0.26, 0.21]]
duration_uniform = [[1, 1.5], [1.3, 1.7], [1.4, 2.0], [1.5, 2.3]]

class Scenarios(object):
    index = 0
    urgent = []
    semi_urgent = []
    time_urgent = []
    time_semi_urgent = []
    time_elective = []
def __init__(self, urgent, semi_urgent, time_urgent, time_semi_urgent, time_elective):
    self.urgent.append(urgent)
    self.semi_urgent.append(semi_urgent)
    self.time_urgent.append(time_urgent)
    self.time_semi_urgent.append(time_semi_urgent)
    self.time_elective.append(time_elective)
    self.identity = Scenarios.index
    Scenarios.index += 1

class Dynamic:
    output_seed = []

def generate_surgeries():
    for z in range(parameters.total_budget):
```python
time_urgent = []
time_semi_urgent = []
time_elective = []

for j in range(len(parameters.no_specialties)):
    semi_urgent = [0 for i in range(len(parameters.time_horizon))]
    urgent = [0 for i in range(len(parameters.time_horizon))]
    seed = u16807d(Dynamic.output_seed[i])
    for i in range(len(parameters.time_horizon)):
        for j in range(len(parameters.no_specialties)):
            np.random.seed(seed[0])
            if parameters.dist_no_semi == 'negative_binomial':
                semi_urgent[i][j] = int(parameters.semi_urgent_nb[i][j][1] / np.random.negative_binomial(parameters.semi_urgent_nb[i][j][1], parameters.semi_urgent_nb[i][j][0]))
                dsu = [0] * semi_urgent[i][j]
                seed = u16807d(seed[0])
                np.random.seed(seed[0])
                for l in range(semi_urgent[i][j]):
                    if parameters.dist_duration == 'log_normal':
                        dsu[l] = round(np.random.lognormal(parameters.duration_log_normal[j][0], parameters.duration_log_normal[j][1]),2)
                        elif parameters.dist_duration == 'uniform':
                            dsu[l] = round(np.random.uniform(parameters.duration_uniform[j][0], parameters.duration_uniform[j][1]),2)
                            seed = u16807d(seed[0])
                            np.random.seed(seed[0])
                            time_semi_urgent.append(dsu)
                de = [0] * parameters.elective_surgeries[i][j]
                for p in range(len(de)):
                    if parameters.dist_duration == 'log_normal':
                        dp = round(np.random.lognormal(parameters.duration_log_normal[j][0], parameters.duration_log_normal[j][1]),0)
                        elif parameters.dist_duration == 'uniform':
                            dp = round(np.random.uniform(parameters.duration_uniform[j][0], parameters.duration_uniform[j][1]),0)
                            seed = u16807d(seed[0])
                            np.random.seed(seed[0])
                            time_elective.append(de)
                            seed = u16807d(seed[0])
                surgery = Scenarios(urgent, semi_urgent, time_urgent, time_semi_urgent, time_elective)

# Supporting seed generators (MRG32k3A, U16807d) and random variate generators for generating the scenarios
#
# generate a poisson random variate with the given parameter

def generate_poisson(cum_probability, mean):
    x = 0
    p = math.exp(-mean)
    s = p
    while cum_probability > s:
        x += 1
```

\[ p = \text{mean} / x \]
\[ s = p \]
\[ \text{return } x \]

**Threshold based Scheduling Heuristic**

```python
class G:
    iseed_perturb = 0
    total_budget = 0
    min_sample = 0
    delta = 0
    budget_vector = 0
    iteration_max = 500
    imax = 400
    s = 1
    c = 2
    dimension = (parameters.threshold_no-2) * parameters.time_horizon * parameters.no_specialties
    total_neighbors = 2 * dimension

class Decision:
    z = []
    p = []
    x = []
    y = []
    utilization_elective = []
    utilization_semi_urgent = []
    time_elective = []
    time_semi_urgent = []

    @staticmethod
def generate_decision():
        p = [[0 for j in range(parameters.no_specialties)] for i in range(parameters.time_horizon)]
        p.insert(0, [[0 for j in range(parameters.no_specialties)]] for k in range(parameters.time_horizon))
        # p.extend([[0 for i in range(parameters.no_specialties)]] for j in range(parameters.time_horizon))
        z = [[0 for i in range(parameters.no_specialties)] for j in range(parameters.time_horizon)]
        z.insert(0, [0])
        y = [[0 for j in range(parameters.no_specialties)] for i in range(parameters.time_horizon)]
        y.insert(0, [0])
        x = [[0 for i in range(parameters.no_specialties)] for j in range(parameters.time_horizon)]
        x.insert(0, [0])
        utilization_elective = [[0 for j in range(parameters.no_specialties)] for i in range(parameters.time_horizon)]
        utilization_elective.insert(0, [0])
        time_elective = copy.deepcopy(utilization_elective)
        time_semi_urgent = copy.deepcopy(utilization_elective)
        Decision.time_semi_urgent.extend(time_semi_urgent)
        Decision.time_elective.extend(time_elective)
        Decision.utilization_elective.extend(utilization_elective)
        Decision.utilization_semi_urgent.extend(utilization_semi_urgent)
```

---

106
```python
Decision.p.extend(p)
Decision.y.extend(y)
Decision.x.extend(x)
Decision.z.extend(z)

@staticmethod
def delete_decision():
    Decision.z = []
    Decision.p = []
    Decision.x = []
    Decision.y = []
    Decision.utilization_elective = []
    Decision.utilization_semi_urgent = []
    Decision.time_elective = []
    Decision.time_semi_urgent = []

class DecisionVariable:
    x = [[0 for i in range(G.dimension)] for j in range(G.iteration_max)]
    th_su = [[0 for j in range(parameters.no_specialties)] for i in range(parameters.time_horizon)] for k in range(G.iteration_max)]
    th_su_p = copy.deepcopy(th_su)
    th_e_p = copy.deepcopy(th_su)

# random number generator for each seed
def u16807d_randomnumber(seed):
    urv = 0
    while urv <= 0 or urv >= 1:
        temp = seed * 16807
        seed = temp % 2147483647
        urv = seed / 2147483648
    return urv, seed

def surgery_scheduling(number, decision_vector):
    z = copy.deepcopy(Decision.x)
    p = copy.deepcopy(Decision.p)
    y = copy.deepcopy(Decision.y)
    x = copy.deepcopy(Decision.x)
    s_cancellation = copy.deepcopy(Decision.x)
    # number of postponed semi-urgent cases
    mps = copy.deepcopy(Decision.x)
    # number of postponed elective cases
    mpe = copy.deepcopy(Decision.x)
    # duration elective cases
    elective_scenario_duration = copy.deepcopy(Decision.x)
    postponed_scenario_duration = copy.deepcopy(Decision.x)
    semi_urgent_scenario_duration = copy.deepcopy(Decision.x)
    over_time_scenario = copy.deepcopy(Decision.x)
    cost_postponement = copy.deepcopy(Decision.x)
    cost_regular = copy.deepcopy(Decision.x)
    cost_capadd = copy.deepcopy(Decision.x)
    cost_cancellation = [0 for i in range(parameters.no_specialties)]
    total_cost = [0 for i in range(parameters.time_horizon)]
    cum_cost = [0 for i in range(parameters.time_horizon)]
    # calculation of total regular time cost
    total_capacity = [[0 for j in range(parameters.no_specialties)] for i in range(parameters.time_horizon)]
    avg_duration = [0 for j in range(parameters.no_specialties)]
    if parameters.dist_duration == 'log_normal':
        for i in range(parameters.time_horizon):
            for j in range(parameters.no_specialties):
                for k in range(parameters.no_uses):
                    cost_postponement[i][j] = u16807d_randomnumber(parameters.no_specialties)
                    cost_regular[i][j] = u16807d_randomnumber(parameters.no_specialties)
                    cost_capadd[i][j] = u16807d_randomnumber(parameters.no_specialties)
                    cost_cancellation[i][j] = u16807d_randomnumber(parameters.no_specialties)
                    total_cost[i][j] += u16807d_randomnumber(parameters.no_specialties)
                    cum_cost[i][j] += u16807d_randomnumber(parameters.no_specialties)
                    total_capacity[i][j] += u16807d_randomnumber(parameters.no_specialties)
                    avg_duration[i][j] += u16807d_randomnumber(parameters.no_specialties)
    ```
total_capacity[i][j] = parameters.z0[i][j] * math.exp(parameters.duration_log_normal[j][0] + 0.5 * parameters.duration_log_normal[j][1]**2)
if i == 0:
    avg_duration[] = math.exp(parameters.duration_log_normal[j][0] + 0.5 * parameters.duration_log_normal[j][1]**2)

elif parameters.dist_duration == 'uniform':
    for i in range(parameters.time_horizon):
        for j in range(parameters.no_specialties):
            total_capacity[i][j] = parameters.z0[i][j] * 0.5 * (parameters.duration_uniform[j][0] + parameters.duration_uniform[j][1])
            if i == 0:
                avg_duration[] = 0.5 * (parameters.duration_uniform[j][0] + parameters.duration_uniform[j][1])

for j in range(1, parameters.time_horizon+1):
    cost_regular[j] = np.dot(parameters.regular_cost[:parameters.no_specialties], total_capacity[j-1])

for j in range(1, parameters.time_horizon + 1):
    # parameters.time_horizon+1
    # start of module to calculate the utilization at stage 'j'
    for i in range(parameters.no_specialties):
        nps[j][i] = 0
        npe[j][i] = 0
        if j > 1:
            for l in range(1, j):
                npe[j][i] += p[l][j - l - 1][i]
                if j - l <= 2:
                    nps[j][i] += y[l][j - l][i]
        # end of module to calculate utilization at stage 'j'

    # start of module to optimize stage 'j' considering average duration of revealed scenarios
    optimal_result = optimization_module(j, x, y, z, p, npe, nps, number, decision_vector)
    z = optimal_result[0]
    x = optimal_result[1]
    y = optimal_result[2]
    p = optimal_result[3]
    s_cancellation[j] = optimal_result[5]

    # print "stage ".j
    # print "capacity allocation to the future = ". z
    # print "electives to be performed = ", x
    # print "semi_urgent schedule = ", y
    # print "postponement of electives = ", p

    # cost calculation across all stages and specialties
    for i in range(parameters.no_specialties):
        cost_over_time[j][i] = max(over_time_scenario[j][i], 0) * parameters.over_time_cost[i]

        cost_capadd[j][i] = parameters.capadd_cost[i] * z[j][i]
        if j < parameters.time_horizon:
            cost_postponement[j][i] += y[j][i][1] * parameters.postponement_cost[i]
        if j < parameters.time_horizon - 1:
            cost_postponement[j][i] += 2 * y[j][2][i] * parameters.postponement_cost[i]
        for l in range(parameters.time_horizon - j):
            cost_postponement[j][i] += (p[j][l][i] * (l+1)) * parameters.
postponement_cost[i]

# cost of cancellation
for i in range(parameters.no_specialties):
    cost_cancellation[i] = p(parameters.time_horizon)[0][i] * parameters.
    cancellation_cost[i] + sum(s_cancellation[j][i] for j in range(1, parameters.
    time_horizon+1)) * parameters.s_cancellation_cost[i]

# calculation of elective, postponed elective and semi-urgent duration across all
# scenarios and stages
for i in range(parameters.no_specialties):
    temp = SurgeryGeneration.Scenarios.time_elective[number][i::parameters.
    no_specialties]
    temp_semi_urgent = SurgeryGeneration.Scenarios.time_semi_urgent[number][i::
    parameters.no_specialties]
    for j in range(1, parameters.time_horizon + 1):
        elective_scenario_duration[j][i] = sum(temp[j - 1][i](int(x[j][i]))
        extra = int(x[j][i])
        if j < parameters.time_horizon:
            for k in range(j + 1, parameters.time_horizon + 1):
                postponed_scenario_duration[k][i] += sum(temp[j - 1][extra:extra + int(p
                [j][k - j - 1][i]))
                extra += int(p[j][k - j - 1][i])
        if 2 < j <= parameters.time_horizon:
            semi_urgent_scenario_duration[j][i] = sum(temp_semi_urgent[j - 1][i](int(y[j
            ][0][i])) + sum(
            temp_semi_urgent[j - 2][int(y[j][0][i]):int(y[j][0][i]) + int(y[j -
            1][1][i])))
            semi_urgent_scenario_duration[j][i] += sum(temp_semi_urgent[j - 3][
            int(y[j][0][i]) + int(y[j -
            1][1][i]):int(y[j][0][1]) + int(y[j - 2][2][i]))
        elif j == 1:
            semi_urgent_scenario_duration[j][i] = sum(temp_semi_urgent[j - 1][i](int(y[j
            ][0][i]))
        elif j == 2:
            semi_urgent_scenario_duration[j][i] = sum(temp_semi_urgent[j - 1][i](int(y[j
            ][0][i])) + sum(
            temp_semi_urgent[j - 2][int(y[j - 1][0][i]):int(y[j - 1][0][i]) + int(y[j -
            1][1][i]))
    # calculation of over time across all stages and specialties
    for j in range(1, parameters.time_horizon + 1):
        over_time_scenario[j][i] = max(0, semi_urgent_scenario_duration[j][i] + 
        elective_scenario_duration[j][i] + 
        postponed_scenario_duration[j][i] - parameters.z0[j -1][i] * avg_duration[i-1] - z[j][i] * avg_duration[i-1])

# total cost at various stages
for j in range(1, parameters.time_horizon + 1):
    total_cost[j - 1] = cost_regular[j] + sum(cost_postponement[j]) + sum(cost_capadd[j
    ])
    + n.p.dot(over_time_scenario[j].parameters.overtime_cost)
    if j == 1:
        cum_cost[0] = total_cost[0]
    else:
        cum_cost[j - 1] = total_cost[j - 1] + cum_cost[j - 2]
```python
def optimization_module(stagn, x, y, z, p, npe, nps, number, decision_vector):
    # DECISION VARIABLES FOR THIS PROBLEM (SEPARATE THEM OUT FROM THE SINGLE LIST
    # CHANGE THIS EVERY TIME THERE IS AN INCREASE OR DECREASE IN THE NUMBER OF PARAMETERS
    th_su = copy.deepcopy(decision_vector[0: parameters.time_horizon])
    th_su_p = copy.deepcopy(decision_vector[1 * parameters.time_horizon: 2 * parameters.time_horizon])
    th_e_p = copy.deepcopy(decision_vector[2 * parameters.time_horizon: 3 * parameters.time_horizon])
    # Decision variables for this problem (separate them out from the single list
    # Change this every time there is an increase or decrease in the number of parameters
    # th_su = copy.deepcopy(decision_vector[0: parameters.time_horizon])
    # th_su_p = copy.deepcopy(decision_vector[1 * parameters.time_horizon: 2 * parameters.time_horizon])
    # th_e_p = copy.deepcopy(decision_vector[2 * parameters.time_horizon: 3 * parameters.time_horizon])
    # change this every time there is an increase or decrease in the number of parameters
    # initialize all variables for one stage of the heuristic
    semu_pst = [0 for j in range(parameters.no_specialties)]
    semu_cnc = [0 for j in range(parameters.no_specialties)]
    elctivable_to_pst = [0 for j in range(parameters.no_specialties)]
    elctivable_to_cnc = [0 for j in range(parameters.no_specialties)]
    cap_elctibles = [0 for j in range(parameters.no_specialties)]
    cap_sume_urgent = [0 for j in range(parameters.no_specialties)]
    semu_pst = [0 for j in range(parameters.no_specialties)]
    semu_cnc = [0 for j in range(parameters.no_specialties)]
    elctivable_to_pst = [0 for j in range(parameters.no_specialties)]
    elctivable_to_cnc = [0 for j in range(parameters.no_specialties)]
    cap_elctibles = [0 for j in range(parameters.no_specialties)]
    cap_sume_urgent = [0 for j in range(parameters.no_specialties)]
    for i in range(parameters.no_specialties):
        # schedule semi-urgent surgeries for the same day
        cap_sume_urgent[i] = max(th_su[stagn - 1][i] + z[stagn][i] - npe[i] - nps[i],
                                  parameters.z0[stagn - 1][i] - parameters.
                                  elective_surgeries[stagn - 1][i]
                                  + x[stagn][i] - npe[i] - nps[i][i])
        y[stag][0][i] = min(SurgeryGeneration.Scenarios.sume_urgent[number][stagn - 1][i],
                               cap_sume_urgent[i])
        # schedule elective surgeries on this stage
        cap_elctible[i] = max(0, parameters.z0[stagn - 1][i] + z[stagn][i] - npe[i] - nps[i][i][i]
                               - y[stagn][0][i])
        cap_elctible[i] = max(0, parameters.z0[stagn - 1][i] + z[stagn][i] - npe[i] - nps[i][i][i]
                               - th_su_p[stagn - 1][i])
        x[stagn][i] = min(parameters.elective_surgeries[stagn - 1][i], cap_elctible[i])
        # semi-urgent and elective cases to be postponed
        semu_to_pst[i] = SurgeryGeneration.Scenarios.sume_urgent[number][stagn - 1][i] - y[stag][0][i]
        elctivable_to_pst[i] = parameters.elective_surgeries[stagn - 1][i] - x[stag][i]
        if stag == parameters.time_horizon:
            if elctivable_to_pst[i] > 0:
                elctivable_to_cnc[i] = copy.deepcopy(elctivable_to_pst[i])
                elctivable_to_pst[i] = 0
            if semu_to_pst[i] > 0:
                y[stag][1][i] = copy.deepcopy(semu_to_pst[i])
                semu_to_cnc[i] = copy.deepcopy(semu_to_pst[i])
                semu_to_pst[i] = 0
        # schedule the postponed semi-urgent cases
        if semu_to_pst[i] > 0:
            if stag == 1:
                y[stag][1][i] = min(semu_to_pst[i], th_su_p[stag][i])
```

sem_urgent_to_postpone[i] = y[stage][1][i]
y[stage][2][i] = min(sem_urgent_to_postpone[i], th_su_p[stage+1][i] + parameters.th_z)
z[stage + 2][i] = max[y[stage][2][i] - th_su_p[stage+1][i], 0]

elif 2 <= stage < parameters.time_horizon - 2:
y[stage][1][i] = min(sem_urgent_to_postpone[i], th_su_p[stage][i] + z[stage + 1][i] - y[stage - 1][2][i])
semi_urgent_to_postpone[i] = y[stage][1][i]
y[stage][2][i] = min(sem_urgent_to_postpone[i], th_su_p[stage+1][i] + parameters.th_z)
z[stage + 2][i] = max(y[stage][2][i] - th_su_p[stage+1][i], 0)
semi_urgent_to_cancel[i] = semi_urgent_to_postpone[i] - y[stage][2][i]

elif stage = parameters.time_horizon - 1:
y[stage][1][i] = min(sem_urgent_to_postpone[i], th_su_p[stage][i] + z[stage + 1][i] - y[stage - 1][2][i])
semi_urgent_to_cancel[i] = semi_urgent_to_postpone[i] - y[stage][1][i]
y[stage][2][i] = semi_urgent_to_cancel[i]
semi_urgent_to_postpone[i] = 0

# schedule the postponed elective cases
if electives_to_postpone[i] > 0 and stage < parameters.time_horizon:
k = stage + 1
while electives_to_postpone[i] > 0:
    if stage == 1 and k < parameters.time_horizon + 1:
        p[stage][k-stage-1][i] = min(electives_to_postpone[i], max(0, th_e_p[k-1][i] - parameters.elective_surgeries[k-1][i]))
electives_to_postpone[i] -= p[stage][k-stage-1][i]
    elif stage > 1 and k < parameters.time_horizon + 1:
        posted_electives -= 0
        for l in range(1, stage):
            posted_electives += p[l][k-1-1][i]
        p[stage][k-stage-1][i] = min(electives_to_postpone[i], max(0, th_e_p[k-1][i] - parameters.elective_surgeries[k-1][i] - posted_electives))
electives_to_postpone[i] -= p[stage][k-stage-1][i]

k += 1

if k == parameters.time_horizon + 1:
electives_to_cancel[i] = copy.deepcopy(electives_to_postpone[i])
electives_to_postpone[i] = 0

return z, x, y, p, electives_to_cancel, semi_urgent_to_cancel

# print semi_urgent_to_postpone
# print x
# print y

def oracle_surgery_scheduling(x_input_oracle, total_scenarios, current_sample):
    feasibility = 1
    est_value = 0
    est_var = float('Inf')
    true_value = 0
    ind_values = np.array([0] * total_scenarios)
    ind_regular = np.array([0] * total_scenarios)
    ind_overtime = np.array([0] * total_scenarios)
    ind_capadd = np.array([0] * total_scenarios)
    ind_postponement = np.array([0] * total_scenarios)
    ind_cancelation = np.array([0] * total_scenarios)
    total_cost_matrix = []
    cum_cost_matrix = []
    x_input = np.copy(x_input_oracle)
if all(0 <= i <= 10 for i in x_input_oracle):
    # enforce equality of the postponement of semi-urgent parameter
    final = [ii-jj for ii, jj in zip(sum(parameters.z0[i]), x_input_oracle)]
    x_input.extend(x_input_oracle)
    col = parameters.no_specialties
    row = parameters.time_horizon * parameters.threshold_no
    x_oracle = [x_input[col * i: col * (i + 1)] for i in range(row)]
    for i in range(total_scenarios):
        cost_scenario = surgery_scheduling(i + current_sample, x_oracle)
        total_cost_matrix.append(cost_scenario[1])
        cum_cost_matrix.append(cost_scenario[0])
        ind_values[i] = cum_cost_matrix[i][parameters.time_horizon - 1]
        ind_cancellation[i] = cost_scenario[3]
    est_value = np.mean(ind_values)
    if not total_scenarios == 1:
        est_var = np.var(ind_values, ddf=1)
    else:
        feasibility = 0
        # print "Estimated Total Cost = ", est_value, "Estimated Variance=", est_var
    return feasibility, est_value, est_var, true_value, ind_values

Threshold Optimization using CPLEX

def mip_surgery(total_scenarios, start):
    total_capacity = [[0 for j in range(parameters.no_specialties)]
    avg_duration = [0 for j in range(parameters.no_specialties)]
    if parameters.dist_duration == 'log_normal':
        for i in range(parameters.time_horizon):
            total_capacity[i][j] = parameters.z0[i][j] * math.exp(parameters.
            duration_log_normal[j][0] + 0.5 * parameters.duration_log_normal[j][1]
    if i == 0:
        avg_duration[j] = math.exp(parameters.duration_log_normal[j][0] + 0.5 *
        parameters.duration_log_normal[j][1] ** 2)
    elif parameters.dist_duration == 'uniform':
        for i in range(parameters.time_horizon):
            total_capacity[i][j] = parameters.z0[i][j] * 0.5 * (parameters.
            duration_uniform[j][0] + parameters.duration_uniform[j][1])
    if i == 0:
        avg_duration[j] = 0.5 * (parameters.duration_uniform[j][0] + parameters.
        duration_uniform[j][1])
    reg_cost = [0] * parameters.time_horizon
    for j in range(parameters.time_horizon):
        reg_cost[j] = np.dot(parameters.regular_cost[:, parameters.no_specialties],
        total_capacity[j])
        # print reg_cost
    # not releasing unused semi-urgent time
    assumption = 0
    # create an LP problem
    model = pulp.LpProblem("surgery_scheduling", LpMinimize)
    # total number of scenarios in this problem
    n = total_scenarios
    # total number of days in the time horizon
t = parameters.time_horizon
# total number of specialties
sp = parameters.no_specialties
# tuple of scenarios
t_scenario = [(i,) for i in range(1, n + 1)]
# tuple of specialties
t_specialties = [(i,) for i in range(1, sp + 1)]
# tuple of days
t_days = [(i,) for i in range(1, t + 1)]
# create all combinations for postponement of elective surgeries
p_electives = [tuple(p) for p in itertools.combinations(range(1, t + 2), 2)]
# create all combinations for scheduling of elective surgeries
x_electives = []
[[x_electives.append((t_scenario[i] + t_specialties[j] + p_electives dummy[p] for p in range(len(p_electives dummy)))] for j in range(len(t_specialties))] for i in range(len(t_scenario))]
# create all combinations for scheduling semi-elective surgeries
y_semi_urgent = []
y_semi_urgent dummy = []
[[y_semi_urgent dummy.append((t_scenario[i] + t_specialties[j] + y_semi_urgent dummy[p] for p in range(len(y_semi_urgent dummy)))] for j in range(len(t_specialties))] for i in range(len(t_scenario))]
# create all combinations for capacity addition in the hospital
z dummy = []
z_capacity = []
[[z dummy.append((t_scenario[i] + t_specialties[j] + z dummy[p] for p in range(len(z dummy)))] for j in range(len(t_specialties))] for i in range(len(t_scenario))]
# create a tuple for alpha, beta and gamma
threshold = []
[threshold.append((t_specialties[j] + t_days[p] for p in range(len(t_days)))] for j in range(len(t specialties))]
cost_scenarios = [(i, i + 1)]
# add all the tuples created as variables in the problem
x = LPVariable.dicts("x", x_electives, lowBound=0, cat=LPInteger)
a = LPVariable.dicts("a", x_electives, lowBound=0, cat=LPInteger)
b = LPVariable.dicts("b", p_electives, lowBound=0, cat=LPInteger)
s = LPVariable.dicts("s", x_electives, lowBound=0, cat=LPInteger)
y = LPVariable.dicts("y", y_semi_urgent, lowBound=0, cat=LPInteger)
p = LPVariable.dicts("p", p_electives, lowBound=0, cat=LPInteger)
z = LPVariable.dicts("z", z_capacity, lowBound=0, cat=LPInteger)
# artificial variables added for reformulation of min or max constraints
g1 = LPVariable.dicts("g1", x_electives, lowBound=0, cat=LPBinary)
h1 = LPVariable.dicts("h1", x_electives, lowBound=0, cat=LPBinary)
g2 = LPVariable.dicts("g2", x_electives, lowBound=0, cat=LPBinary)
h2 = LPVariable.dicts("h2", x_electives, lowBound=0, cat=LPBinary)
g3 = LPVariable.dicts("g3", x_electives, lowBound=0, cat=LPBinary)
h3 = LPVariable.dicts("h3", x_electives, lowBound=0, cat=LPBinary)
g4 = LPVariable.dicts("g4", x_electives, lowBound=0, cat=LPBinary)
h4 = LPVariable.dicts("h4", x_electives, lowBound=0, cat=LPBinary)
g5 = LPVariable.dicts("g5", p_electives, lowBound=0, cat=LPBinary)
h5 = LPVariable.dicts("h5", p_electives, lowBound=0, cat=LPBinary)
```python
g6 = lpVariable.dicts("G6", p_electives, lowBound=0, cat=lpBinary)
h6 = lpVariable.dicts("H6", p_electives, lowBound=0, cat=lpBinary)
g7 = lpVariable.dicts("G7", z_capacity, lowBound=0, cat=lpBinary)
h7 = lpVariable.dicts("H7", z_capacity, lowBound=0, cat=lpBinary)
g8 = lpVariable.dicts("G8", z_capacity, lowBound=0, cat=lpBinary)
h8 = lpVariable.dicts("H8", z_capacity, lowBound=0, cat=lpBinary)
cost = lpVariable.dicts("C", cost_scenarios, lowBound=0, cat=lpInteger)

# decision variables in the formulation
alpha_sem_i_urgent = lpVariable.dicts("TY", threshold, lowBound=0, cat=lpInteger)
beta_sem_i_urgent = lpVariable.dicts("TP", threshold, lowBound=0, cat=lpInteger)
gamma_elective = lpVariable.dicts("TPE", threshold, lowBound=0, cat=lpInteger)

def ADD_OBJECTIVE_FUNCTION
  model += lpSum(lpSum(lpSum(parameters.cancellation_cost[i-1]* p[s,i,j,t+1] for j in range(1, t+1) for i in range(1, len(t_specialties)+1)) for s in range(1, len(t_scenarios)+1))
  + lpSum(lpSum(lpSum(parameters.s_cancellation_cost[i-1] * (y[s,i,parameters.time_horizon-1].parameters.time_horizon+1) + \n    y[s,i,parameters.time_horizon].parameters.time_horizon*1) for i in range(1, len(t_specialties)+1)) for s in range(1, len(t_scenarios)+1))
  + lpSum(lpSum(lpSum(parameters.postponement_cost[i-1] * (k-j) * p[s,i,j,k] for k in range(j+1, t+1)) for j in range(1, t) for i in range(1, len(t_specialties)+1)) for s in range(1, len(t_scenarios)+1))
  + lpSum(lpSum(lpSum(parameters.capadd_cost[i-1] * (z[s,i,j+2] for j in range(1, t-1)) for i in range(1, len(t_specialties)+1)) for s in range(1, len(t_scenarios)+1))
  + lpSum(lpSum(lpSum(parameters.postponement_cost[i-1] * (k-j) * p[s,i,j,k] for k in range(1, t-1) for i in range(1, len(t_specialties)+1)) for s in range(1, len(t_scenarios)+1))
  + lpSum(lpSum(parameters.cancellation_cost[i-1] * sc[s,i,k] for k in range(1, t+1)) for i in range(1, len(t_specialties)+1)) for s in range(1, len(t_scenarios)+1))

# ADD CONSTRAINTS

# for loop for adding constraints on the decision variables
for i in range(1, len(t_specialties)+1):
  for j in range(1, len(t_days)+1):
    model += alpha_sem_i_urgent[i, j] <= parameters.z0[j - 1][i - 1]
    model += beta_sem_i_urgent[i, j] <= alpha_sem_i_urgent[i, j]
    model += gamma_electives[i, j] <= parameters.z0[j - 1][i - 1] - alpha_sem_i_urgent[i, j] * "ThPE%/%s" % (i, j)

for s in range(1, len(t_scenarios)+1):
  for i in range(1, len(t_specialties)+1):
    for j in range(1, t+1):
      # scheduling of semi-urgent surgeries
      if j < t - 1:
        model += lpSum(y[s, i, j, k] for k in range(j, j + 3)) + sc[s,i,j] <= SurgeryGeneration.Scenarios.sem_i_urgent[s + start - 1][j - 1][i - 1] * "TotalY%/%s" % (s, i, j)
      elif j == t - 1:
        model += lpSum(y[s, i, j, k] for k in range(j, j + 3)) <= SurgeryGeneration.Scenarios.sem_i_urgent[s + start - 1][j - 1][i - 1] * "TotalY%/%s" % (s, i, j)
      elif j == t:
        pass
```

model += lpSum(y[s, i, j, k] for k in range(j, j + 2)) == SurgeryGeneration.Scenarios.semi_urgent[s + start - 1][j - 1][i - 1]. "Total # of same day semi-urgent surgeries"

scheduling of same day semi-urgent surgery

# y[s, i, j] - min(U^s, i, j, max(alpha_i-j - previous y, capacity - n_ij - postponed + added capacity))

model += y[s, i, j] <= SurgeryGeneration.Scenarios.semi_urgent[s + start - 1][j - 1][i - 1] <= y[s, i, j] + (1-g1[s, i, j])*1000
model += a[s, i, j] <= y[s, i, j] + (1-h1[s, i, j])*1000
model += g1[s, i, j] + h1[s, i, j] == 1

# code module for scheduling elective cases
if j >= 3:
  model += a[s, i, j] >= alpha_semi_urgent[i, j] - lpSum(y[s, i, k, j] for k in range(j - 2, j)) + z[s, i, j - 2, j]
  model += a[s, i, j] >= parameters.z0[j - 1][i - 1] - parameters.elective_surgeries[j-1][i-1] - lpSum(p[s, i, k, j] for k in range(1, j)) + z[s, i, j - 2, j] - lpSum(y[s, i, k, j] for k in range(1, j))
  model += alpha_semi_urgent[i, j] - lpSum(y[s, i, k, j] for k in range(j - 2, j)) + z[s, i, j - 2, j] - lpSum(y[s, i, k, j] for k in range(j - 2, j))
  model += parameters.z0[j - 1][i - 1] - parameters.elective_surgeries[j - 1][i - 1] - lpSum(p[s, i, k, j] for k in range(1, j)) - alpha_semi_urgent[i, j] + z[s, i, j - 2, j] - lpSum(y[s, i, k, j] for k in range(j - 2, j))
  model += g2[s, i, j] + h2[s, i, j] == 1

model += x[s, i, j] <= parameters.elective_surgeries[j - 1][i - 1]
model += x[s, i, j] <= parameters.z0[j - 1][i - 1] + z[s, i, j - 2, j]
model += parameters.elective_surgeries[j - 1][i - 1] - lpSum(p[s, i, k, j] for k in range(1, j)) - alpha_semi_urgent[i, j]
model += parameters.elective_surgeries[j - 1][i - 1] <= x[s, i, j] + (1-g3[s, i, j])*1000
model += parameters.z0[j - 1][i - 1] + z[s, i, j - 2, j] - lpSum(p[s, i, k, j] for k in range(1, j))
model += alpha_semi_urgent[i, j] <= x[s, i, j] + (1-h3[s, i, j])*1000
model += g3[s, i, j] + h3[s, i, j] == 1
eelif j == 1:
  model += a[s, i, j] >= alpha_semi_urgent[i, j]
model += a[s, i, j] >= parameters.z0[j - 1][i - 1] - parameters.elective_surgeries[j-1][i-1]
model += alpha_semi_urgent[i, j] - lpSum(p[s, i, k, j] for k in range(1, j)) - alpha_semi_urgent[i, j]
model += parameters.z0[j - 1][i - 1] - parameters.elective_surgeries[j - 1][i - 1] >= a[s, i, j] - (1-g2[s, i, j])*1000
model += parameters.z0[j - 1][i - 1] - parameters.elective_surgeries[j - 1][i - 1] >= a[s, i, j] - (1-g2[s, i, j])*1000
model += g2[s, i, j] + h2[s, i, j] == 1

model += x[s, i, j] <= parameters.elective_surgeries[j - 1][i - 1]
model += x[s, i, j] <= parameters.z0[j - 1][i - 1] - alpha_semi_urgent[i, j]
model += parameters.elective_surgeries[j - 1][i - 1] <= x[s, i, j] + (1-g3[s, i, j])*1000
model += parameters.z0[j - 1][i - 1] - alpha_semi_urgent[i, j] <= x[s, i, j] + (1-h3[s, i, j])*1000
model += g3[s, i, j] + h3[s, i, j] == 1
eelif j == 2:
  model += a[s, i, j] >= alpha_semi_urgent[i, j] - lpSum(y[s, i, k, j] for k in range(j, j + 2))
model += \( a[s, i, j] \) \( \geq \) parameters.z0[j - 1][i - 1] - parameters.elective_surgeries[j - 1][i - 1] - p[s, i, j - 1, j] - y[s, i, j - 1, j]

model += alpha_semi_urgent[i, j] - lpSum(y[s, i, k, j] for k in range(1, j)) \( \geq \) a[s, i, j] - (1-g2[s, i, j]) * 1000

model += parameters.z0[j - 1][i - 1] - parameters.elective_surgeries[j - 1][i - 1] - p[s, i, j - 1, j] - y[s, i, j - 1, j]

# scheduling of elective surgeries

# code module for postponing elective cases

if j < t:
    for k in range(j+1, t+2):
        if k < t+1:
            if j == 1 and k-j == 1:
                model += p[s, i, j, k] \( \leq \) parameters.elective_surgeries[j - 1][i - 1] - x[s, i, j]
                model += b[s, i, j, k] \( \geq \) b[s, i, j, k]
                model += b[s, i, j, k] \( \leq \) p[s, i, j, k] + (1-h6[s, i, j, k]) * 10000
                model += parameters.elective_surgeries[j - 1][i - 1] - x[s, i, j] - lpSum(p[s, i, j, l] for l in range(j+1, k))
                model += b[s, i, j, k] \( \geq \) gamma_electives[i, k] - parameters.elective_surgeries[k - 1][i - 1]

        elif j > 1 and k-j == 1:
            model += p[s, i, j, k] \( \leq \) parameters.elective_surgeries[j - 1][i - 1] - x[s, i, j] - lpSum(p[s, i, j, l] for l in range(j+1, k))
            model += b[s, i, j, k] \( \leq \) b[s, i, j, k]
            model += b[s, i, j, k] \( \leq \) p[s, i, j, k] + (1-h6[s, i, j, k]) * 10000
            model += parameters.elective_surgeries[j - 1][i - 1] - x[s, i, j] - lpSum(p[s, i, j, l] for l in range(j+1, k))
            model += gamma_electives[i, k] - parameters.elective_surgeries[k - 1][i - 1] \( \geq \) b[s, i, j, k] - (1-g6[s, i, j, k]) * 10000
            model += g6[s, i, j, k] + h6[s, i, j, k] \( \geq \) 1

            model += p[s, i, j, k] \( \leq \) parameters.elective_surgeries[j - 1][i - 1] - x[s, i, j] - lpSum(p[s, i, j, l] for l in range(j+1, k))
            model += b[s, i, j, k] \( \leq \) b[s, i, j, k]
            model += b[s, i, j, k] \( \leq \) p[s, i, j, k] + (1-h6[s, i, j, k]) * 10000
            model += parameters.elective_surgeries[j - 1][i - 1] - x[s, i, j] - lpSum(p[s, i, j, l] for l in range(j+1, k)) \( \geq \) p[s, i, j, k] + (1-g6[s, i, j, k]) * 10000
            model += g6[s, i, j, k] + h6[s, i, j, k] \( \geq \) 1

            model += b[s, i, j, k] \( \geq \) gamma_electives[i, k] - parameters.elective_surgeries[k - 1][i - 1]
            model += gamma_electives[i, k] - parameters.elective_surgeries[k - 1][i - 1] \( \geq \) b[s, i, j, k] - (1-g6[s, i, j, k]) * 10000
            model += 0 \( \geq \) b[s, i, j, k] - (1-h6[s, i, j, k]) * 10000
            model += g6[s, i, j, k] + h6[s, i, j, k] \( \geq \) 1

            model += p[s, i, j, k] \( \leq \) parameters.elective_surgeries[j - 1][i - 1] - x[s, i, j]
            model += p[s, i, j, k] \( \leq \) b[s, i, j, k]
model <- b[s,i,j,k] <- p[s,i,j,k] + (1-h6[s,i,j,k])*10000
model <- parameters.elective_surgeries[j-1][i-1] - x[s,i,j] <- p[s,i,j,k] + (1-g6[s,i,j,k])*10000
model <- g6[s,i,j,k] + h6[s,i,j,k] == 1

model <- b[s,i,j,k] == gamma_electives[i,k] - parameters.elective_surgeries[k-i][i-1] - lpSum(p[s,i,j,k] for l in range(1,j))
model <- gamma_electives[i,k] - parameters.elective_surgeries[k-i][i-1] - lpSum(p[s,i,j,k] for l in range(1,j)) \ >= b[s,i,j,k] - (1-g6[s,i,j,k])*10000
model <- 0 >= b[s,i,j,k] - (1-h6[s,i,j,k])*10000
model <- g6[s,i,j,k] + h6[s,i,j,k] == 1

else:
  model <- p[s,i,j,k] <- b[s,i,j,k]
  model <- b[s,i,j,k] <- p[s,i,j,k] + (1-h6[s,i,j,k])*10000
  model <- p[s,i,j,k] <- parameters.elective_surgeries[j-1][i-1] - x[s,i,j] - lpSum(p[s,i,j,k] for l in range(j+1,k))
  model <- parameters.elective_surgeries[j-1][i-1] - x[s,i,j] - lpSum(p[s,i,j,k] for l in range(j+1,k)) \ >= b[s,i,j,k] - (1-g6[s,i,j,k])*10000
  model <- 0 >= b[s,i,j,k] - (1-h6[s,i,j,k])*10000
  model <- g6[s,i,j,k] + h6[s,i,j,k] == 1
  elif k == t+1:
    model <- p[s,i,j,k] == parameters.elective_surgeries[j-1][i-1] - x[s,i,j] \ - lpSum(p[s,i,j,k] for l in range(j+1,k)). "P3%ks%ks" %s" %s" %s" %s" %s" %s" %s" %s" %s" %s" %s" %s"

  elif j == t:
    model <- p[s,i,j+1] == parameters.elective_surgeries[j-1][i-1] - x[s,i,j]. "P3%ks%ks" %s" %s" %s" %s" %s" %s" %s" %s" %s" %s" %s" %s"

# code module for postponing semi-urgent cases
if j == 1:
  model <- y[s,i,j+1] <- beta_semi_urgent[i,j+1]
  model <- y[s,i,j+1] <= SurgeryGeneration.Scenarios.semi_urgent[s,start-1][j-1][i-1] - y[s,i,j]
  model <- beta_semi_urgent[i,j+1] <= y[s,i,j+1] + (1-g4[s,i,j])*1000
  model <- SurgeryGeneration.Scenarios.semi_urgent[s,start-1][j-1][i-1] - y[s,i,j] <= y[s,i,j+1] + (1-h4[s,i,j])*1000
  model <- g4[s,i,j] + h4[s,i,j] == 1
  model <- y[s,i,j+2] <= SurgeryGeneration.Scenarios.semi_urgent[s,start-1][j-1][i-1] - y[s,i,j] - y[s,i,j+1]
  model <- y[s,i,j+2] <= beta_semi_urgent[i,j+2] + parameters.th_z
  model <- beta_semi_urgent[i,j+2] + parameters.th_z <= y[s,i,j+2] + (1-g8[s,i,j+2])*10000
  model <- SurgeryGeneration.Scenarios.semi_urgent[s,start-1][j-1][i-1] - y[s,i,j] - y[s,i,j+1] <= y[s,i,j+2] + (1-h8[s,i,j+2])*10000
  model <- g8[s,i,j+2] + h8[s,i,j+2] == 1
  model <- z[s,i,j+2] => y[s,i,j+2] - beta_semi_urgent[i,j+2]
  model <- y[s,i,j+2] - beta_semi_urgent[i,j+2] => z[s,i,j+2] - (1-g7[s,i,j+2])*10000
  model <- 0 >= z[s,i,j+2] - (1-h7[s,i,j+2])*10000
model += g7[s,i,j,j+2] + h7[s,i,j,j+2] == 1
model += sc[s,i,j] -- SurgeryGeneration.Scenarios.semi_urgent[s+start -1][j-1][i-1] - y[s,i,j,j] - y[s,i,j,j+1] - y[s,i,j,j+2]
elif j < t-1:
model += y[s,i,j,j+1] <= beta_semi_urgent[i,j+1] + z[s,i,j-1,j+1] - y[s, i,j-1,j+1]
model += y[s,i,j,j+1] <= SurgeryGeneration.Scenarios.semi_urgent[s+start -1][j-1][i-1] - y[s,i,j,j]
model += beta_semi_urgent[i,j+1] + z[s,i,j,j+1] - y[s,i,j,j+1] <= y[s, i,j+1] + (1-g7[s,i,j]) *1000
model += SurgeryGeneration.Scenarios.semi_urgent[s+start-1][j-1][i-1] - y[s,i,j,j] <= y[s,i,j,j+1] + (1-h8[s,i,j]) *1000
model += g4[s,i,j] + h4[s,i,j] == 1
model += y[s,i,j,j+2] <= SurgeryGeneration.Scenarios.semi_urgent[s+start -1][j-1][i-1] - y[s,i,j,j] - y[s,i,j,j+1] - y[s,i,j,j+2] + (1-h8[s,i,j]) *1000
model += g8[s,i,j,j+2] + h8[s,i,j,j+2] == 1
model += z[s,i,j,j+2] >= y[s,i,j,j+2] - beta_semi_urgent[i,j+2]
model += y[s,i,j,j+2] <= beta_semi_urgent[i,j+2] + parameters.th_z
model += beta_semi_urgent[i,j+2] + parameters.th_z <= y[s,i,j,j+2] + (1-g8[s,i,j,j+2]) *10000
model += SurgeryGeneration.Scenarios.semi_urgent[s+start-1][j-1][i-1] - y[s,i,j,j] - y[s,i,j,j+1] <= y[s,i,j,j+2] + (1-h8[s,i,j,j+2]) *10000
model += g8[s,i,j,j] + h8[s,i,j,j] == 1
model += sc[s,i,j] -- SurgeryGeneration.Scenarios.semi_urgent[s+start -1][j-1][i-1] - y[s,i,j,j] - y[s,i,j,j+1] - y[s,i,j,j+2]
elif j == t-1:
model += y[s,i,j,j+1] <= beta_semi_urgent[i,j+1] + z[s,i,j-1,j+1] - y[s, i,j-1,j+1]
model += y[s,i,j,j+1] <= SurgeryGeneration.Scenarios.semi_urgent[s+start -1][j-1][i-1] - y[s,i,j,j]
model += beta_semi_urgent[i,j+1] + z[s,i,j-1,j+1] - y[s,i,j-j+1] <= y[s, i,j-j+1] + (1-g4[s,i,j]) *1000
model += SurgeryGeneration.Scenarios.semi_urgent[s+start-1][j-1][i-1] - y[s,i,j,j] <= y[s,i,j,j+1] + (1-h4[s,i,j]) *1000
model += g4[s,i,j] + h4[s,i,j] == 1
model += y[s,i,j,j+2] == SurgeryGeneration.Scenarios.semi_urgent[s+start -1][j-1][i-1] - y[s,i,j,j] - y[s,i,j,j+1] - y[s,i,j,j+2]. "Y1%S%S%S" % (s,i,j,j+2)
elif j == t:
model += y[s,i,j,j+1] <= beta_semi_urgent[i,j+1] + z[s,i,j-1,j+1] - y[s, i,j-1,j+1]
model += y[s,i,j,j+1] <= SurgeryGeneration.Scenarios.semi_urgent[s+start -1][j-1][i-1] - y[s,i,j,j]
model += beta_semi_urgent[i,j+1] + z[s,i,j-1,j+1] - y[s,i,j-j+1] <= y[s, i,j-j+1] + (1-g4[s,i,j]) *1000
model += SurgeryGeneration.Scenarios.semi_urgent[s+start-1][j-1][i-1] - y[s,i,j,j] <= y[s,i,j,j+1] + (1-h4[s,i,j]) *1000
model += g4[s,i,j] + h4[s,i,j] == 1
model += y[s,i,j,j+2] == SurgeryGeneration.Scenarios.semi_urgent[s+start -1][j-1][i-1] - y[s,i,j,j] - y[s,i,j,j+1] - y[s,i,j,j+2]. "Y1%S%S%S%S" % (s,i,j,j+2)
# if j <= t-2:
#    model += z[s,i,j,j+2] <= 3
#    filename = "cplex_*" + str(total_scenarios) + str(start//2) + ".lp"
model.writeLP(filename)
return model.numVariables(), model.numConstraints()
for i in range(len(total_scenarios)):
    A = mip.surgery(total_scenarios[i].start+2)
    NumVars[i] = A[0]
    start += 1
    start = 0
for i in range(len(total_scenarios)):
    file_name = "cplex_" + str(total_scenarios[i]) + str(start) + ".lp"
    start += 1
for j in range(len(time_limit)):
    parameters.time = time_limit[j]
    if j>0 and optimality_gap[i][j-1] <= 0.05:
        optimality_gap[i][j] = optimality_gap[i][j-1]
        output[i][j] = output[i][j-1]
    else:
        A = solveMIPCplex(file_name, parameters.time)
        optimality_gap[i][j] = A[1]*100
        output[i][j] = A[0][:20]
        output[i][j] = [round(x) for x in output[i][j]]
        f.write( str(total_scenarios[i]) + ":t " + str(time_limit[j]) + ":t " + str(output[i][j]) + "t " + str(optimality_gap[i][j]) + "\n")
        f.write("Solutions for Sample Size = " + str(total_scenarios[i]) + " are " + str(output[i][j]) + "\n")
    f.close()
return output, optimality_gap

def solveMIPCplex(filename, time_limit):
    c = cplex.Cplex(filename)
    c.parameters.timeline.set(time_limit)
    c.parameters.mip.strategy.file.set(2)
    c.parameters.emphasis.memory.set(1)
    c.parameters.workmem.set(2500)
    c.set_results_stream(None)
    output = [0 for i in range(parameters.no_specialties * parameters.time_horizon)]
    # c.set_results_stream(\"CplexTest.txt\")
    try:
        c.solve()
    except CplexSolverError:
        print "Exception raised during solve"
    return # solution status()
    gap = c.solution.MIP.get_mip_relative_gap()
    # for i in range(len(c.solution.get_values())):
    #    print c.variables.get_names(i), \"=\" , c.solution.get_values(i)
    count = 0
    for i in range(1,parameters.time_horizon+1):
        for j in range(1,parameters.no_specialties+1):
            var_name = \"TY\"(\" + str(j) + \".\" + str(i) + \")\"
            output[count] = c.solution.get_values(var_name)
            count +=1
        # print c.solution.get_values(value(alpha_sen_i_urgent[i,j]) for j in range(1,parameters.time_horizon+1))
    return output, gap