University of Alberta

Spare Parts Provisioning Decision Support Model for Long Lead Time Spares

By

Amit S. Aulakh

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science
in
Engineering Management

Department of Mechanical Engineering

Edmonton, Alberta

ABSTRACT

Large corporations have a significant amount of working capital tied into the acquisition and storage of spare parts. In the industry, spare parts inventory policies and strategies are often developed in isolation from reliability centered maintenance practices – this results in significant wasteful direct and indirect cost attached to spare parts management for the equipment operator. This report will focus on developing a methodology for minimizing lifecycle indirect and direct cost that comes from storing long lead time spares. A combined Monte-Carlo and Genetic Algorithm based optimization approach to finding the optimal spare parts storage strategy is proposed. In this study, the indirect and direct cost of having a spare part in the storage facility will be balanced against the cost of lost opportunity that results from decreased availability - a consequence of not having the required spare part available when an equipment failure event occurs. The results of this study present the benefits of optimizing long lead time spares through a joint Monte-Carlo & Genetic Algorithm based approach.

Table of Contents

Table of Figures

Table of Acronyms & Definitions

1.	CSP	-	Conservative Sparing Policy
2.	GA	-	Genetic Algorithm
3.	MC	-	Monte-Carlo Simulation
4.	MSP	-	Minimal Sparing Policy
5.	Normal Distribution	-	Normal continuous probability distribution $Normal\ PDF = f(x;\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{\left(\frac{-(x-\mu)^2}{2\sigma^2}\right)}$ $\mu = Mean$ $\sigma = Standard\ Deviation$
6.	OOP	-	Object Oriented Programming
7.	PDF	-	Probability Density Function
8.	RBD	-	Reliability Block Diagram
9.	RAM	-	Reliability, Availability and Maintainability
10.	Weibull Distribution	-	Weibull continuous probability distribution. $Weibull\ PDF = f(x;c,h) = \frac{h}{c}\left(\frac{x}{c}\right)^{h-1} e^{-\left(\frac{x}{c}\right)^h}$ $h = Shape\ parameter\ (h > 0)$ $c = Scale\ parameter\ (c > 0)$

1 Introduction

Based on the author’s personal experience[1] working in the oil sands & pipeline industry, large corporations have a significant amount of working capital tied into the acquisition and storage of spare parts. Spare parts inventory policies and strategies are often developed in isolation from reliability based maintenance practices – this results in significant direct and indirect cost attached to spare parts management for the equipment operator. This study proposes a combined Monte-Carlo and Genetic Algorithm based optimization approach to finding an optimal spare parts strategy, through simulation, that can result in significant cost savings for the equipment operator over the lifecycle of the facility.

In this report, the indirect & direct cost of having a spare part in the storage facility, onsite or offsite, will be balanced against the cost of lost opportunity that results from decreased availability - a consequence of not having a required spare part available when an equipment failure event occurs. In other words, the optimization objective/fitness function will have competing priorities, on the one hand we want to maximize plant availability by having a sufficient number of spares available, and on the other hand we want to minimize the number of spares in the storage facility as that results in ongoing indirect cost (cost of storage, electricity, warehouse personnel etc.).

Mathematical programming and simulation modeling are currently the most frequently used techniques for developing a spare parts provisioning decision support model in academia and

[1] 6 years with Syncrude Canada Ltd. & 3 years with Enbridge Pipelines Inc.

industry (1). Mathematical programming involves development of mathematical models based on goal programming/preemptive prioritization and linear programming techniques. Simulation modeling is frequently used in the industry for RAM[2] analysis and spare parts inventory management lifecycle cost simulation. A number of commercial simulation software packages are available for RBD[3] simulation - packages like *Reliasoft BlockSim*, *ARINC Raptor*, and *AVSIM* put the power of simulation modeling in the hands of reliability engineers in the industry, however, simulation modeling is not an optimization technique (2). To optimize a spare parts provisioning decision support model, it is necessary to integrate the simulation model with a global optimization technique such as *Genetic Algorithms*, *Simulated Annealing* (3) (4), or *Tabu Search* (5).

This report, consisting of six chapters, will present a case study for using joint monte-carlo and genetic algorithm based approach for finding the optimal spare parts storage policy that can minimize facility lifecycle operating cost. The second chapter of this report will review Monte-Carlo methods and their areas of application. Numerical examples will be provided to clarify the concept. Chapter 3 will review genetic algorithms and compare them to classical optimization algorithms. The system definition will be given in Chapter 4, key assumptions as well as constraints that apply to this study will be discussed and the programming logic/model will be validated. Chapter 5 will present the numerical application of the optimization concept along with actual constraints that were used in the optimization algorithm. The study will close with Chapter 6 which provides the conclusion and discusses the significant cost savings that could

[2] Reliability Availability Maintainability
[3] Reliability Block Diagram

result from switching to an optimal sparing policy as compared to using an ad-hoc or non-optimal sparing strategy.

2 Review – Monte Carlo Methods

The term “Monte Carlo” is associated with numerical computational algorithms that use repeated distribution based random sampling to calculate approximate solutions to probabilistic problems. Monte Carlo methods are used when it is not possible (or very difficult) to obtain an exact solution to a problem that involves a high degree of uncertainty/variability (6). Due to their numerical/iterative nature and dependence on random/pseudo-random numbers, Monte Carlo methods are appropriate when the calculation will be performed with the assistance of a computer. The general approach used in Monte Carlo algorithms is as follows:

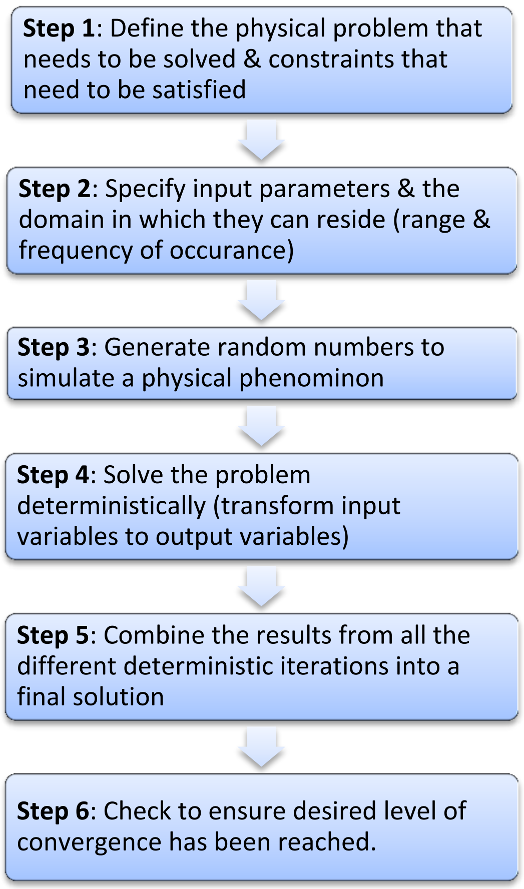
Step 1: Define the physical problem that needs to be solved & constraints that need to be satisfied
Step 2: Specify input parameters & the domain in which they can reside (range & frequency of occurance)
Step 3: Generate random numbers to simulate a physical phenominon
Step 4: Solve the problem deterministically (transform input variables to output variables)
Step 5: Combine the results from all the different deterministic iterations into a final solution
Step 6: Check to ensure desired level of convergence has been reached.

Figure 1: Monte Carlo Methods - Six Step Approach (7)

2.1 Background

With the advent of the first electronic computer near the end of World War II, the technique which had been known previously as “statistical sampling” became a feasible method of solving complex mathematical problems. The ENIAC, now known as the first electronic computer, was developed at the University of Pennsylvania in Philadelphia in 1945 (8). Among the first problems solved by the newly developed computer was a computational model of a thermonuclear reaction (particle diffusion problem) developed by Metropolis, Frankel, and Turkevich (8). The origins of the modern Monte Carlo method can be traced back to a letter written by a scientist by the name of John von Neumann in 1947, outlining a statistical approach to solving the problem of neutron diffusion in fissionable materials (8). It is claimed by many sources that physicist Enrico Fermi had already developed and used the Monte Carlo method as early as the 1930’s, Fermi however did not publish any papers detailing the use of his method (9) (10) (8). The term “Monte Carlo” was first used in the 1940’s by physicists working on the nuclear bomb at the Los Alamos National Laboratory (6) (11).

2.2 Areas of application

Since Monte Carlo simulation methods can be used to model physical phenomenon with a high degree of uncertainty, they have found much use in the area of business risk/decision analysis (6). Monte Carlo methods are widely used today in the industry for cost/contingency calculation (12), reliability modeling, spare parts optimization (13), and construction schedule optimization. In addition to being used in the engineering industry, Monte Carlo methods are also used for:

- Decision making/risk analysis – Monte Carlo methods are commonly used to simulate the probability of a risk event happening and the possible impact of that event based on an impact distribution (6).
- 3D Modeling – Monte Carlo based “global illumination” techniques are used for rendering photorealistic 3D models in the video games/motion film industry (a competing methodology is “radiosity” based global illumination) (14).
- Finance – Monte Carlo based methods are used for business valuation and evaluation of financial derivatives. Random sampling is used to simulate high market share & high margin sales to low market share and low margin sales distributions to come up with a fair value for intellectual property. Statistical sampling techniques are also heavily used in the insurance industry (15).
- Artificial Intelligence – Monte Carlo methods have found some use in the video games industry as an alternative method for programming artificial intelligence (16).
- Telecommunications – Monte Carlo based methods are used for analyzing network node performance and reliability (17).
- Mathematics – Monte Carlo methods are widely used for calculation of multidimensional integrals with complex boundary conditions. Experimental/statistical Monte Carlo based methods are also widely used in the fields of operational research and nuclear physics (11).

2.3 Numerical Examples

The following numerical examples will demonstrate practical usage of the Monte Carlo approach in calculating:

1. The value of Pi (π) by inscribing a circle of known dimensions inside of a square.
2. Area of an irregular shape by using statistical sampling (Appendix B).

Complete code listing is attached for each numerical example in the appendix. *Matlab* numerical computing language has been used for all programming samples.

2.3.1 Approximating the value of Pi (π)

If we inscribe a circle inside a square, as shown in Figure 2, we can calculate the ratio of the area of the square to the circle as:

$$\frac{Area\ of\ Circle}{Area\ of\ Square} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

Knowing this relationship, the value of Pi (π) can be calculated as the ratio of the area of the circle to the area of the square multiplied by 4.

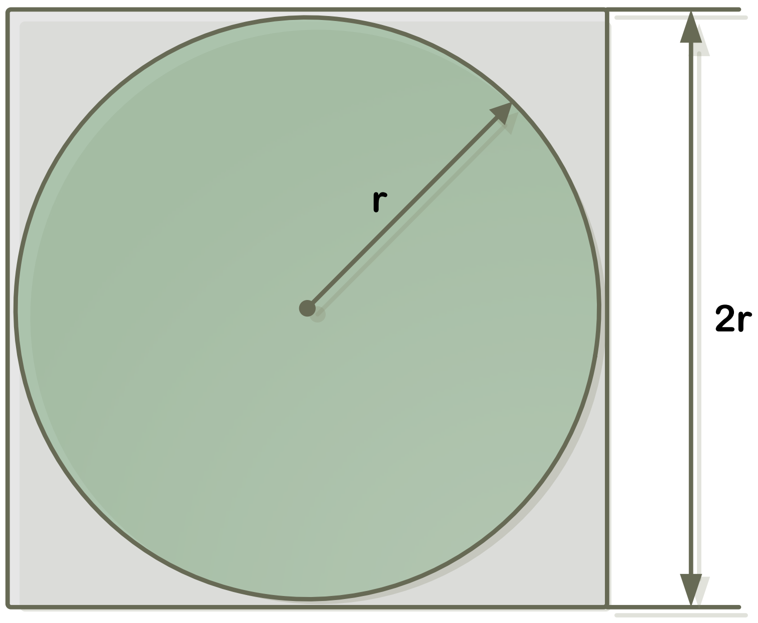

Figure 2: Calculating the Value of PI

This logic can be simulated using a simple Monte Carlo algorithm to approximate the value of Pi (π) - Figure 3 shows a flow diagram for the Monte Carlo algorithm, code listing is attached under Appendix A. The attached code listing uses a circle radius (r) of 5 units and the starting value for n is set at 100 points (number of iterations). The number of iterations was then incremented by 100 points during each run till sample size of 1000 points was reached.

Now since we have calculated the value of Pi (π) using the Monte-Carlo method, the percent deviation of the calculated value of Pi (π) from the theoretical value of Pi (π) can be approximated as:

$$Error\ (deviation\ from\ actual) = \frac{\pi_{Calculated} - \pi_{Theoretical}}{\pi_{Theoretical}} \times 100$$

At 1000 iterations, algorithm execution was stopped as the deviation from actual Pi (π) value was less than 5%. Pi (π) value built into the *Matlab* package was used as the theoretical value for error checking. The number of randomly generated points dropped inside the square during each run is referred to as the "number of iterations"; the dimensions of the square are derived from the circle diameter to get a square with minimum side length while fully inscribing the circle.

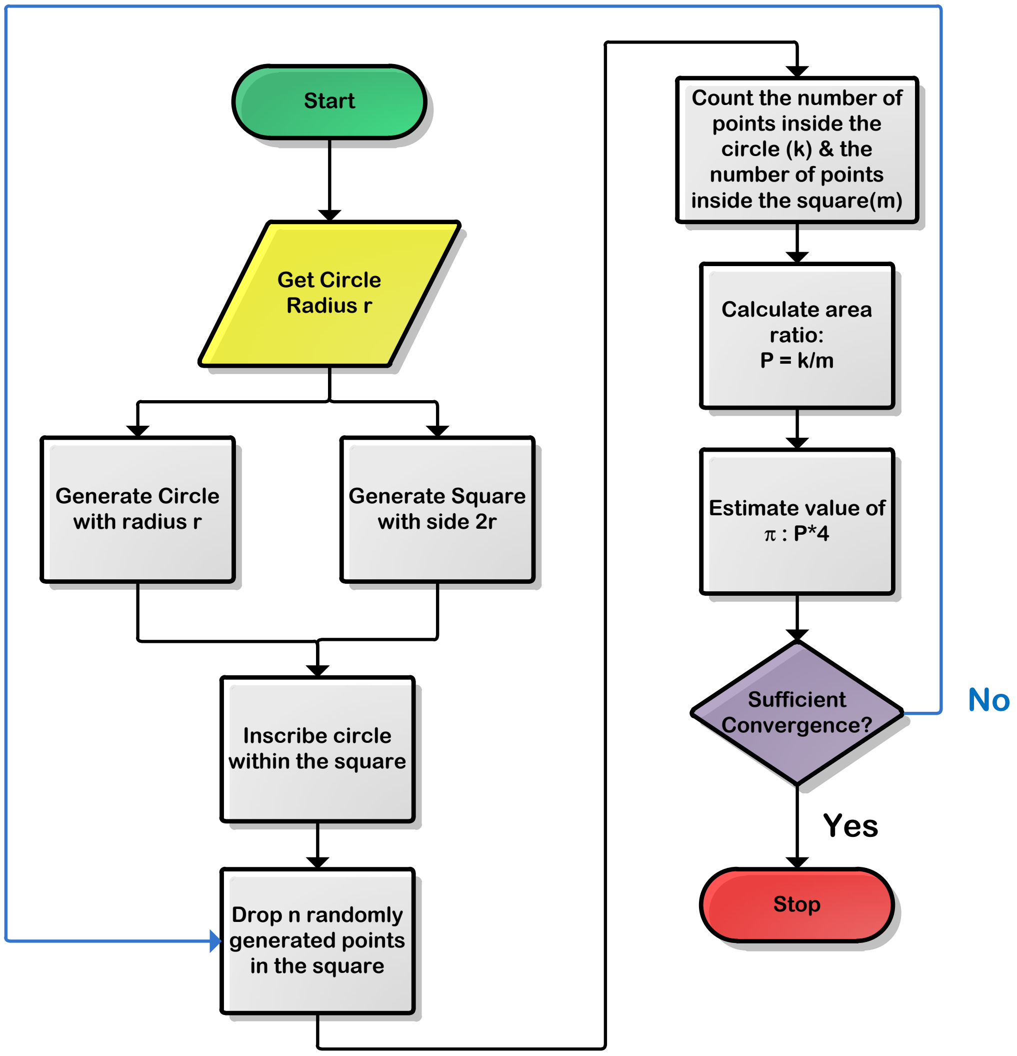
Start
Get Circle Radius r
Generate Circle with radius r
Generate Square with side 2r
Inscribe circle within the square
Drop n randomly generated points in the square
Count the number of points inside the circle (k) & the number of points inside the square(m)
Calculate area ratio: P = k/m
Estimate value of π : P*4
Sufficient Convergence?
No
Yes
Stop

Figure 3: Flowchart - Approximating the Value of Pi (π)

Figure 4 shows the results of the simulation run, the calculated value of Pi (π) was compared against the actual value of Pi (π) to chart the percentage deviation of the calculated value from the actual value. While using the Monte-Carlo methodology, a few general important points should be kept in mind:

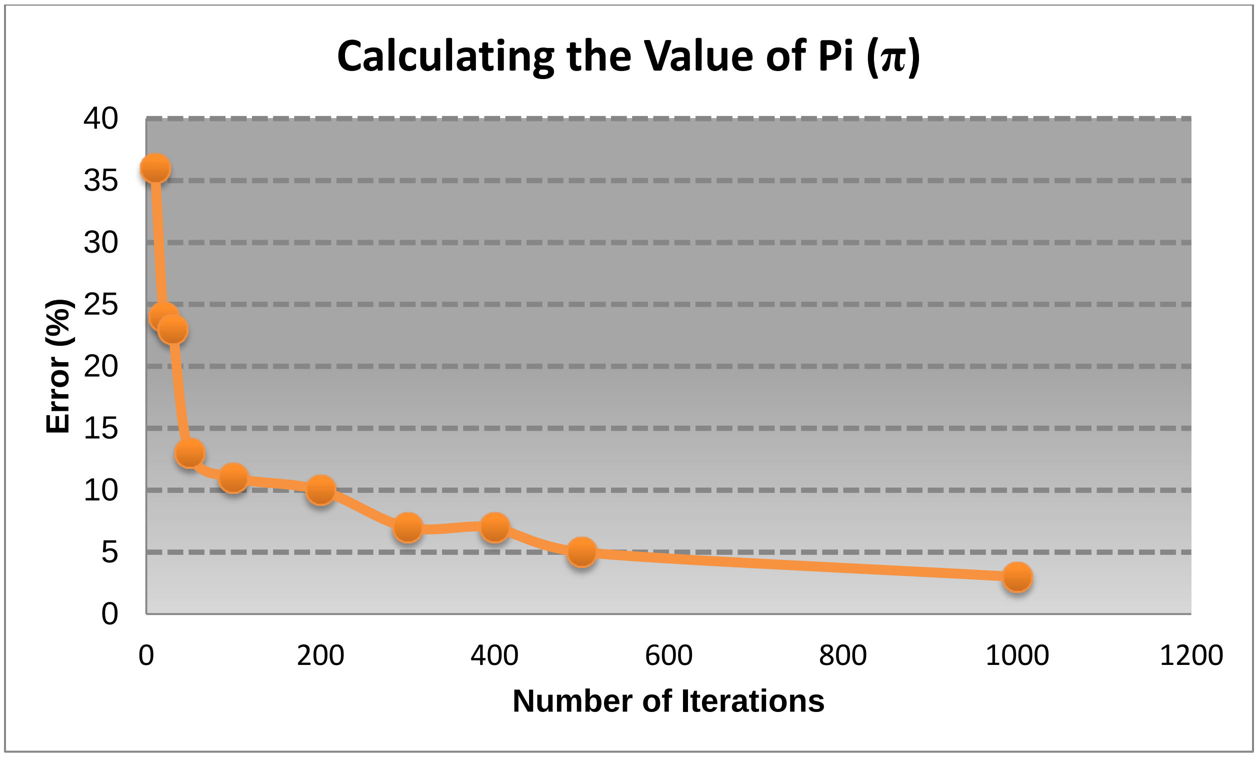

Figure 4: Simulation Results - Calculating the Value of Pi (π)

1. Monte Carlo method relies on good quality random/pseudo random numbers. If the random numbers are not reliable, the results of the analysis won't be reliable either. Within the context of a physical simulation, this means that the variables being simulated must not only be high quality random numbers, but that they must also closely follow the physical distribution that we are attempting to simulate.

2. Monte Carlo method requires that the variables that are being simulated randomly are independent and do not have a high level of correlation with other variables being simulated in the analysis (6). As an example, if atmospheric pressure and elevation are being simulated, the results of the simulation will not mimic the actual phenomenon as atmospheric pressure generally changes with a change in elevation. In other words atmospheric pressure is strongly correlated to elevation and thus these two variables cannot be independently and randomly sampled within a Monte Carlo analysis application. In the case of the PI (π) value calculation example discussed above, two variables are being generated randomly – the x and y coordinates of the points that are dropped inside the square.
3. Monte Carlo methods converge to a better solution as the number of iterations is increased. As can be seen from Figure 4, as the number of iterations was increased from 10 to 1000, the percent deviation from actual Pi (π) value went from ~35% to ~1% range.

3 Review – Genetic Algorithms

Genetic algorithms are a subset of Evolutionary Algorithms that use principals of biology to search the solution space for optimal solutions (19). Genetic algorithms use mutation, random selection, crossover, and elitism to solve constrained and unconstrained optimization problems. GA’s are typically used when the optimization objective function is discontinuous, non-deterministic, non-linear, and or non-differentiable (20). Figure 7 shows a process flow diagram for a typical genetic algorithm (21).

Key terms and concepts used in genetic/evolutionary algorithms are as follows:

- Individual – A set of variables that result in a solution of the optimization problem (not necessarily the best solution).

- Population – A set of individuals. If the objective function has n decision variables and the population size is m, then the population is an m by n matrix. Each new generation is "evolved" from the previous generation, in other words each new generation is genetically linked to the initial starting population. The generation of a population of individuals as opposed to generating a single point at each iteration is what separates GA's from classical optimization algorithms like hill-climbing and derivative focused techniques (22).

- Fitness – Fitness of an individual/organism is simply a measure of how well the objective function is maximized or minimized by the given set of variables.

- Parents – Individuals selected by GA for reproduction. The individuals are typically selected based on their fitness values.

- Children – New individuals created via mutation, crossover, and elitism. Children make the next generation that is evaluated by the GA for ranking their fitness values.

The algorithm shown in Figure 7 starts off with a randomly generated initial population; the initial population size (or the number of individuals/organisms in the population) is specified as a starting point. Each individual is then evaluated against the fitness function, the fitness function is the value of the objective function that is being optimized (typically being minimized in the case of cost & maximized in the case of productivity). The initial population is then scored based on the fitness test and "parent" individuals are selected based on their fitness scores. A set number of "elite" individuals are also selected from the initial population, these "elite" individuals are passed on to the next generation without any modification. The elite individuals usually have the best fitness scores, some elites are randomly selected ignoring the fitness score to keep some degree of randomness/diversity in the subsequent population generations. The "parent" individuals are then mated using mutation and crossover rules, Figure 5 & Figure 6 show a binary representation of mutation and crossover process as it applies to two selected individuals from a population. The "children" individuals that come out of the mutation/crossover process as well as the elites make up the next generation population – this process continues till one of the stopping conditions are triggered. Commonly used algorithm stopping criteria are as follows (23):

1. Number of Generations – a set number of generations after which the algorithm terminates. Genetic algorithms can be extremely time consuming, if a single iteration of the optimization function takes 5 minutes. That translates into about 7 days of run time to reach 100 generations (using 20 individuals per generation).

2. Time Limit Threshold – a set time limit after which the algorithm execution is terminated (CPU execution time).

3. Fitness function tolerance – a set tolerance value of the fitness function, the algorithm terminates when the mean change in objective function value is less than a fitness threshold.

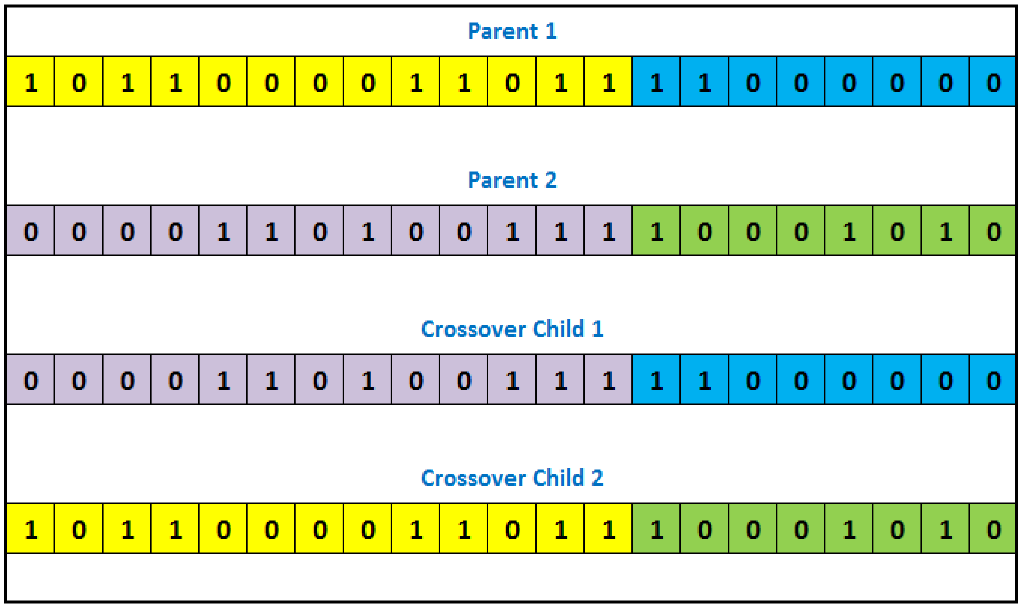

Figure 5: Concept – Crossover

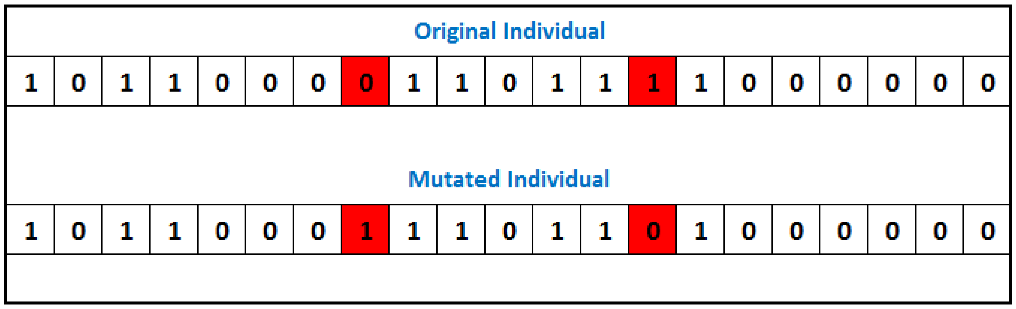

Figure 6: Concept – Mutation

Although GA's can often quickly find good solutions even in situations where the solution space is difficult for classical optimization algorithms (22), it should be noted that good solutions are good only in comparison to other solutions evaluated by the Genetic Algorithm.

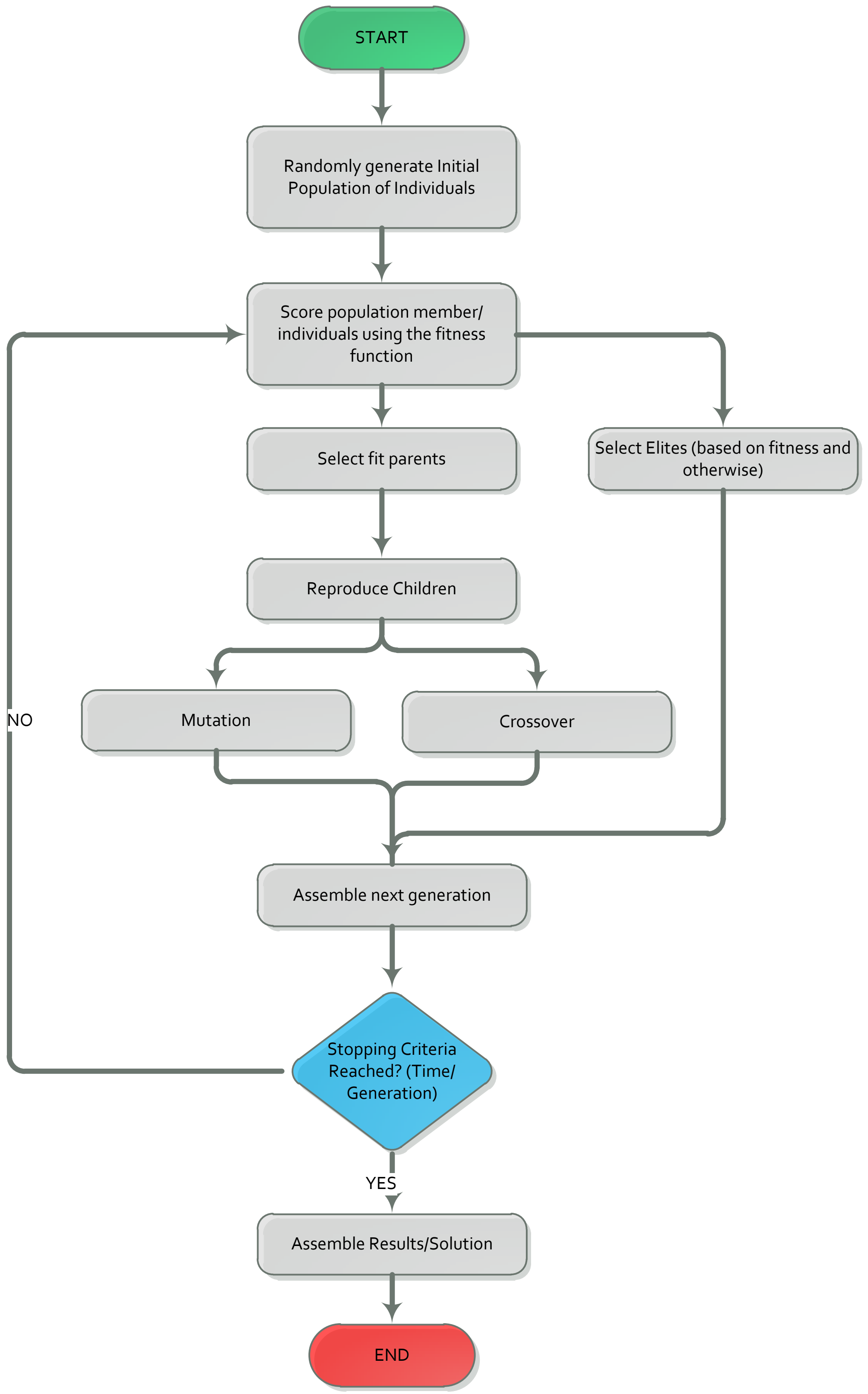
START
Randomly generate Initial Population of Individuals
Score population member/ individuals using the fitness function
Select fit parents
Select Elites (based on fitness and otherwise)
Reproduce Children
Mutation
Crossover
NO
Assemble next generation
Stopping Criteria Reached? (Time/ Generation)
YES
Assemble Results/Solution
END

Figure 7: Genetic Algorithm Flow Diagram

3.1 Genetic Algorithm versus Classical Optimization Algorithms

GA's are algorithms derived from principals of biological evolution; as such they differ from classical optimization algorithms in the following ways (22):

1. GA's use probabilistic functions to transition from one generation/iteration to the next – each successive population generation is derived based on probabilistic crossover/mutation rules.

2. GA's use a population of individuals at each iteration/generation as opposed to using a single point. In other words the best individual in the population approaches global optimal solution.

3. GA's do not make use of derived functions to reach optimal solution – only the fitness function is evaluated for each individual in the population.

4. GA's often use encoded parameters, such as binary representation of the variables for ease of implementing crossover and mutation rules.

4 System Definition

Before getting into the problem that will be solved in this report, let us consider a real world system consisting of multiple tailings centrifugal pumps arranged as shown in Figure 8. This is a typical arrangement that would be seen in a mining operation, be it in the oil sands industry or the potash mining sector. Tailings pumps form a critical part of the mining operation as they transport rejected material from mining/screening operations - tailings are lighter materials that are separated for disposal as a result of milling/crushing/purifying operations. Tailings are typically abrasive slurries that cause accelerated wear of pump internal components.

Figure 8 shows a tailings disposal system process flow diagram. Fifteen identical tailings pumps are spread across 3 trains each having 5 pumps in series configuration. In this system, all three trains are required to be operational for the system to be considered fully functional. Since tailings are abrasive slurries that cause rapid failure of pump components through wear, components such as impellers, suction liners, and seals would have to be stored on site as spares so that when an actual failure event occurs, the component can be replaced as quickly as possible without causing much downtime.

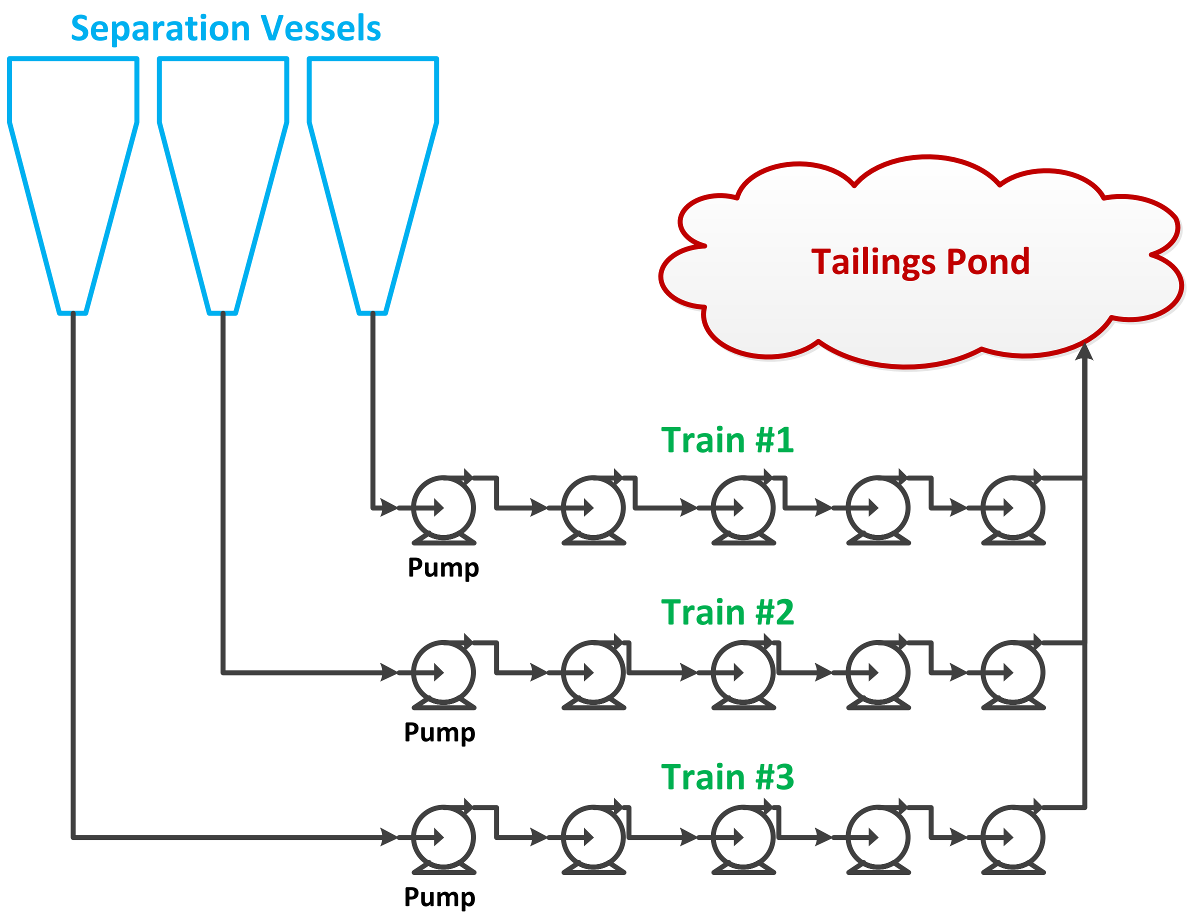

Figure 8: Tailings System Process Flow Diagram

Since the pump components in question are long lead time items, the impellers for example can take anywhere from eight to ten months from the time when the order is placed with the manufacturer to when they are actually delivered on site. The maintenance manager of this facility would be faced with the following critical decisions:

1. How many impellers, suction liners, and seals should be kept in stock as the facility starts up?
2. When should more impellers, suction liners, and seals be ordered?

3. How many impellers, suction liners, and seals should be ordered when it is time to restock?

The maintenance management questions posed above are critical to the profitability of this sample mining operation. Managing the right inventory level for each spare parts pool (where impellers can be called “pool 1”, suction lines can be called “pool 2”, and seals can be called “pool 3”) is critical to the return on investment for the mining operation as not having adequate spares in stock will bring the operation down and will have a negative impact on plant availability. Conversely, having too many spares on hand will also have a negative impact on profitability as each spare part that is in the inventory takes up warehouse room. There will be a cost associated with maintaining the warehouse, having warehouse personnel, inventory management expenses, electricity cost for hot storage, etc.

The question then comes down to what is the right level of spare parts inventory for this facility? Knowing that pump seals fail four times more frequently than pump impellers and suction lines fail two times more frequently than pump impellers – it becomes clear that each failure mode on the pump has its own failure distribution that should be taken into account while answering the questions posed above.

Based on authors personal experience working in the oil sands industry, sparing decisions in the facility discussed above would probably be made as follows:

1. There are 15 pumps in operation, so the facility manager could subjectively keep 5 impellers, 5 suction liners, and 5 seals in the warehouse (one third of all working components as spares based on subjective “rules of thumb”).

2. To cut down on the number of spares in the warehouse and minimize the working capital invested in spares, the facility manager may choose to restock when 1 of each spare is left in stock (after the initial facility startup).
3. The facility manager may decide to order an arbitrary number of spares per purchase order to start off, with the aim of modifying the restocking policy as he or she gains more experience running the equipment.

It is clear from the logic presented above that the actual failure characteristics of each failure mode are ignored when the sparing policy is made. The fact that each failure mode has its own failure distribution, for example the seals fail four times more frequently compared to the impellers, is not reflected in the sparing policy. In other words the spare parts policy is being made in isolation from the reliability characteristics of the equipment in question – granted that as time goes on and the facility manager gains more experience, the spare parts policy will change and evolve.

The point being made in the above practical example is that in the industry currently, spare parts related decisions are made arbitrarily with no consideration given to what the optimal policy might be. The real question in the problem presented above is, what sparing policy will result in the minimum lifecycle cost for the facility? Given that on the one hand we want to always have a spare part in stock to minimize downtime, and on the other hand we want to avoid large direct and indirect cost that comes with having spare parts sitting in the warehouse. The question of "what is the optimal spare parts policy?" cannot be answered without using stochastic reliability based simulation that takes component failure characteristics into account.

In this report, a combined genetic algorithm and Monte-Carlo based approach will be used to optimize the sparing policy for a mixed component series/parallel system. Figure 9 shows a system consisting of 5 types of components, component A is in parallel configuration where one of the two blocks (A1, A2) have to be online for the system to be functional (active redundancy). Component B is in parallel configuration where one of the three blocks (B1, B2, B3) have to be online for the system to be functional. Components C, D, and E are in series configuration – meaning if any one of these three blocks fails, the system will fail.

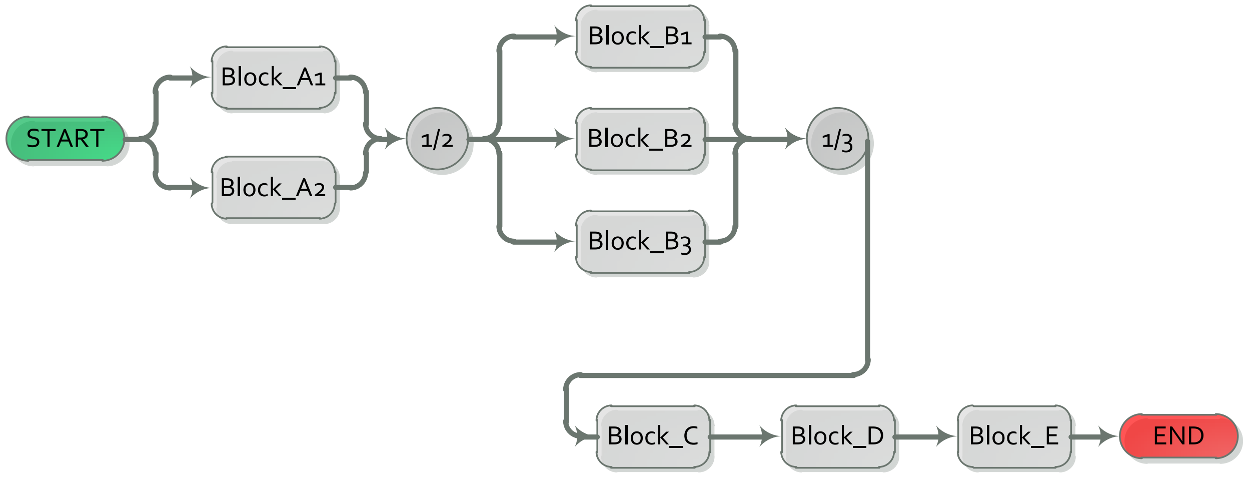

Figure 9: System Reliability Block Diagram

The reliability (R) of the system can be calculated as:

$$R_{sys} = \{1 - [(1 - R_{A1})(1 - R_{A2})]\} \times \{1 - [(1 - R_{B1})(1 - R_{B2})(1 - R_{B3})]\} \times (R_C)(R_D)(R_E)$$

For the purposes of conducting a stochastic analysis, weibull (2 parameter) continuous probability distribution will be used to model the lifetime of each block. Weibull distribution will be presented in the form *W(c,h)* throughout the rest of this report, where c is the scale parameter and h is the shape parameter. Weibull distribution is a very flexible probability distribution that is frequently used to model component failures, where a shape parameter of less than one indicates infant mortality (failure rate decreases over time), shape parameter equal to 1 indicates that random events are causing failure (constant failure rate), and a shape parameter greater than 1 indicates that a component is "aging" and will have an increasing probability of failure as time goes on (increasing failure rate with time).

Normal continuous probability distribution will be used to model component corrective maintenance intervals, normal distributions will be presented in the form *N(μ,ɵ)* where *μ* is the mean and *ɵ* is the standard deviation of the distribution. In other words, 68.2% of the corrective repair time population will be one standard deviation away from the mean, 95.4% of the population will be within two standard deviations from the mean, and 99.6% of the repair time population will be within three standard deviations from the mean. Normal distributions have been used to model corrective repair time in this report for the sake of simplicity and easier understanding of the repair data. Based on the author's personal experience working in the oil sands industry, field repair data typically tends to be slightly skewed towards repairs taking longer than expected, this situation can be modeled better with Log-Normal distributions. Normal distribution will also be used to model stock arrival delay time; the arrival delay is the time elapsed between when the order is placed and when the spare parts arrive on site.

A genetic algorithm based approach will be used to find the optimal spare parts policy which will balance the cost of carrying spare parts in stock against the need to maintain the highest availability possible while targeting the lowest lifecycle operating cost. The objective of this study is to find an optimal sparing strategy, something between the following two extremes:

1. Keeping all the spare parts that the system will ever need in stock from the very start. In this case the right spare will be available immediately after a failure event, but consequently we will have to pay high indirect cost to maintain a large spare parts inventory. This strategy will be referred to as the “conservative” sparing policy (CSP).
2. Not keeping any spares in stock and ordering the spares just as they are needed, in this case the facility will be down while we wait for the right spares to arrive (availability impact – will result in lost production). But on the flip side we would save money by not maintaining an extensive spare parts inventory on site. This strategy will be referred to as the “Minimal Sparing Policy” or (MSP).

The following information will define a complete sparing strategy (decision variables):

1. Knowing what the restock trigger amount should be – in other words the target inventory level at which a new restock order should be placed.
2. Knowing the amount of spare parts to order during a restock event.
3. Knowing what the initial stock level should be for each spare parts pool – this is the number of items that should be in each spare parts pool at the start of the analysis (or the number of spares that we should have in our inventory when the facility starts operation).

Figure 10 shows a visualization of how the stock spare level will vary with time within each spare parts pool – each spare parts pool will start off with an initial stock level, then as equipment failures occur, the number of spare parts in the pool will decrease to the restock trigger amount – at this time a new parts order will be placed with the suppliers. As the newly ordered spare parts arrive on site (subject to the stock arrival time delay distribution), the pool level will rise to the maximum pool capacity and so on. It should be noted that after an order is placed for new parts, they do not arrive on site immediately. The pool level continues to decline as we are waiting for parts and may reach the zero level resulting in facility downtime.

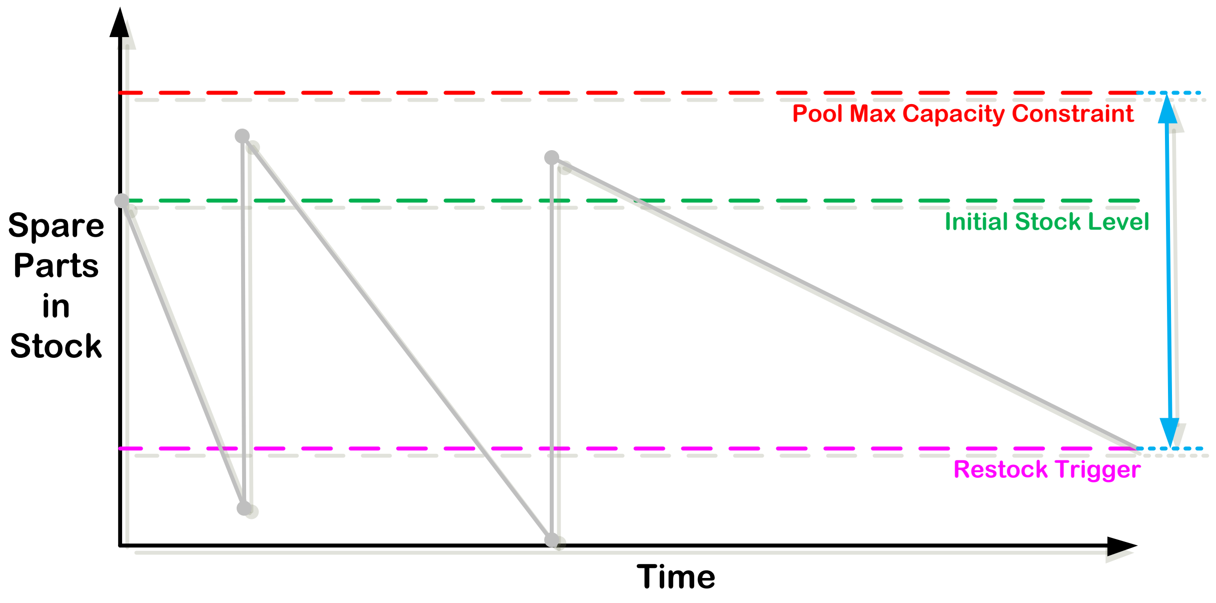

Figure 10: Sparing Strategy Diagram

4.1 Model Development

Object oriented programming (OOP) methodology has been used to develop an algorithm for simulating the lifecycle[4] cost of implementing different spare parts storage policies - taking into account the failure and corrective repair distributions of the components in the model. The algorithm has been developed (by the author) in Matlab[5] programming language, class based approach has been used extensively in the development of the algorithm – this allows for reliable and easy scaling of the algorithm based on the number of spare part pools as well as the number of reliability blocks being used in the model. While many commercial programs[6] exist for performing reliability based spare parts analysis, none of them can be integrated into a genetic algorithm based optimization methodology to allow for selection of an optimal spare parts policy (minimization of lifecycle spare parts cost). The need for developing a Monte-Carlo based reliability/spare parts cost simulation algorithm is demonstrated by Figure 11 - the Monte-Carlo algorithm needs to integrate seamlessly with an outer loop of a genetic algorithm based optimization algorithm. New spare parts policies will be randomly generated by the genetic algorithm and passed on to the Monte-Carlo algorithm for lifecycle cost simulation. The lifecycle cost then gets passed back to the Genetic Algorithm for evaluation and optimization, the simulation is finished when we have found the optimal solution (when algorithm stopping criteria is triggered).

[4] In this report, a fixed lifecycle of 10 years will be examined
[5] MathWorks Matlab v2009b
[6] Reliasoft BlockSim, Isograph AvSim, ARINC Raptor etc.

The Monte-Carlo algorithm takes the following factors into account:

1. Reliability block failure distribution – a failure distribution that is typically derived from real maintenance data that has been observed in the field. *Weibull* distribution will be used to model failure times.

$$Weibull\ Probability\ Density\ Function = f(x; c, h) = \frac{h}{c}\left(\frac{x}{c}\right)^{h-1} e^{-\left(\frac{x}{c}\right)^{h}}$$

$$h = Shape\ parameter\ (h > 0)$$

$$c = Scale\ parameter\ (c > 0)$$

2. Corrective maintenance time distribution – time required for maintenance activities, corrective maintenance will be performed on a block that is in a failed state. Individual block availability counts towards the overall system availability based on the system reliability block diagram, a single block may or may not make the system unavailable based on the block configuration (series or parallel). Corrective repair time will be imposed from the point where the spare part is available - *Normal* distribution will be used to simulate corrective repair duration.

$$Normal\ Probability\ Density\ Function = f(x; \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{\left(\frac{-(x-\mu)^2}{2\sigma^2}\right)}$$

$$\mu = Mean$$

$$\sigma = Standard\ Deviation$$

3. Direct cost per spare part – this is the acquisition cost of each spare part. Spare parts pool class maintains a complete log of how many spare parts were used during the

simulation run. The direct cost of spares is one of the two components that make the total spare parts policy cost (the other the indirect cost).

4. Indirect cost per spare part – Indirect cost is the cost of keeping each spare part in the pool per unit time (1 hour). Indirect cost is essentially the cost of the storage facility normalized per spare per unit time. Spare parts pool class maintains a complete log of how many parts are in the pool during the simulation lifecycle and calculates the total indirect cost of all spare parts in the pool at the end of each simulation hour. Total cost of a sparing policy is calculated as indirect cost plus direct cost over the simulated lifecycle.
5. Initial stock level – Initial stock level specifies the number of spares that are in the pool when the simulation is started. As the stock level drops to the restock trigger, the restock policy is initiated to replenish the pool. It is possible for the pool level to drop below the specified restock trigger without having the restock policy triggered – this happens when multiple blocks fail at the same time and require the same spare part for corrective repair. In these cases the restock policy is triggered during the next iteration (1hr later).
6. Items to add during restock – this specifies the number of items that should be added to the pool when the restocking policy is triggered.
7. Restock trigger – Restock trigger initiates the restocking policy when the number of spare parts in the pool is equal to or drops below the specified amount (as opposed to having a scheduled restock frequency).

8. Stock arrival time delay distribution – Normal distribution random numbers are used to simulate stock arrival delay. The arrival delay is the time elapsed between when the order is placed and when the spare parts arrive. It is assumed that repair cannot begin till the required spare parts are available and that once the spare parts are available, they are in the field immediately (in other words no spare part acquisition logistical delay).

Table 1 shows a list of all the variables and factors, and their associated units, used in the analysis. Unless otherwise specified, all time related figures are in hours. The algorithm is unit neutral, time could be specified in minutes or even seconds but that will have a significant impact on how long each Monte-Carlo and genetic algorithm iteration takes.

Parameter	Variable/Constant	Unit
Failure Time	Random Variable	Time (hrs.)
Repair Time	Random Variable	Time (hrs.)
Direct cost of spares	Constant	Money ($)
Indirect cost of spares	Constant	Money ($)
Restock trigger	Decision Variable	Number of units
Restock amount	Decision Variable	Number of units
Initial stock level	Decision Variable	Number of units
Stock arrival delay	Random Variable	Time (hrs.)

Table 1: Unit Reference

Figure 12 shows a simplified process flow diagram for the reliability block class. The reliability block class models a piece of equipment in the field, each block has a designated spare parts pool – a pool from which the block will draw spare parts when it fails. Multiple blocks can draw from the same spare parts pool. The block class is responsible for managing block availability statistics and passing them on to the main Monte-Carlo simulation algorithm.

Figure 13 and Figure 14 show a simplified process flow diagram for the spare parts pool class. The spare parts pool class maintains a log of all the spare parts assigned to different reliability blocks during the course of the lifecycle simulation. Upon completion of the simulation, the spare part pools class returns the direct, indirect, and total cost of the sparing strategy. Figure 15 shows a process flow diagram for the main Monte-Carlo simulation algorithm, the simulation algorithm advances the simulation time for all attached reliability blocks and spare parts classes. The simulation algorithm also implements the reliability block diagram logic for the Monte-Carlo analysis and calculates overall system availability based on the availability data returned by all the attached block objects. The actual number of block classes and the number of spare parts pool classes used is dictated by the system RBD.

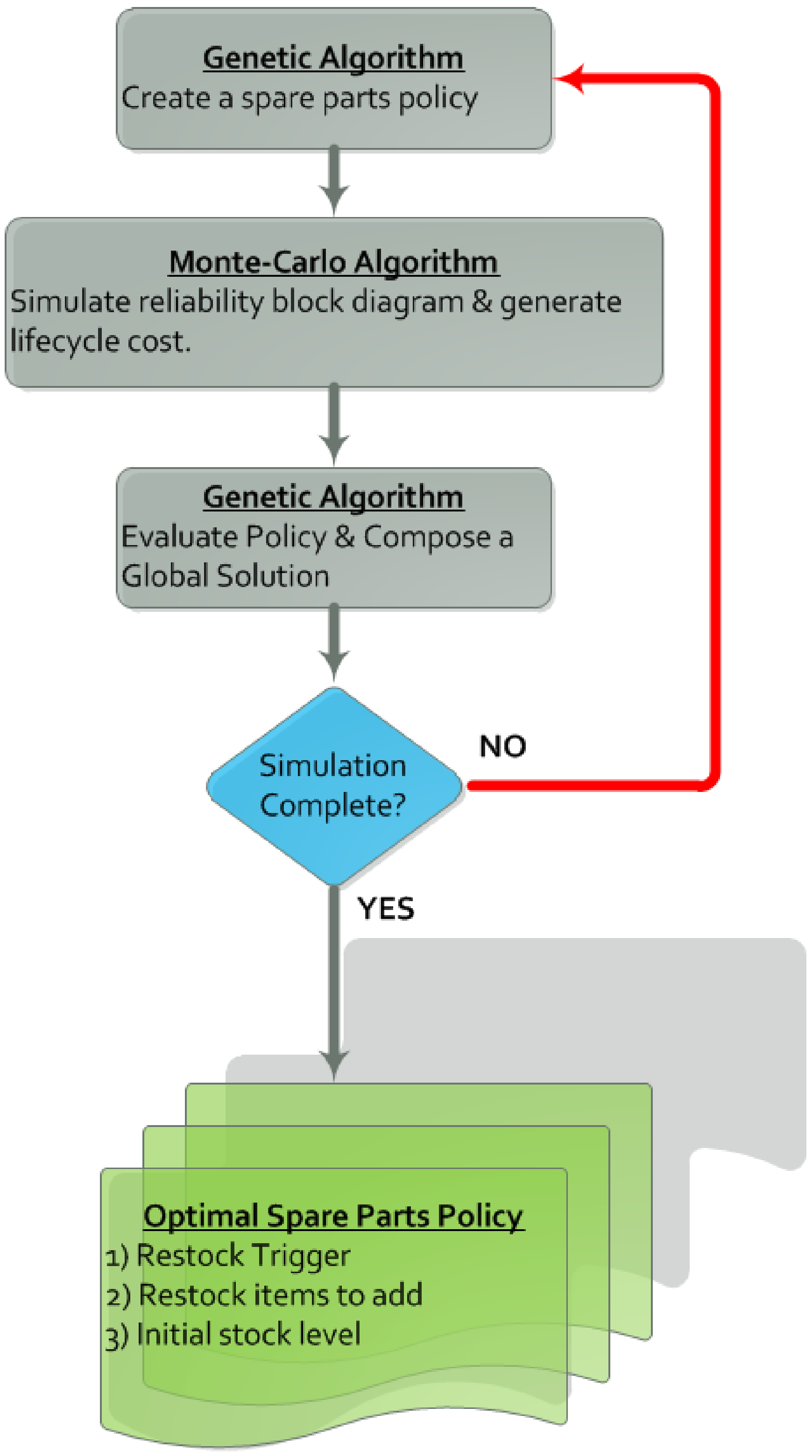
Genetic Algorithm
Create a spare parts policy
Monte-Carlo Algorithm
Simulate reliability block diagram & generate lifecycle cost.
Genetic Algorithm
Evaluate Policy & Compose a Global Solution
Simulation Complete?
NO
YES
Optimal Spare Parts Policy
1) Restock Trigger
2) Restock items to add
3) Initial stock level

Figure 11: Spare Parts Optimization Model

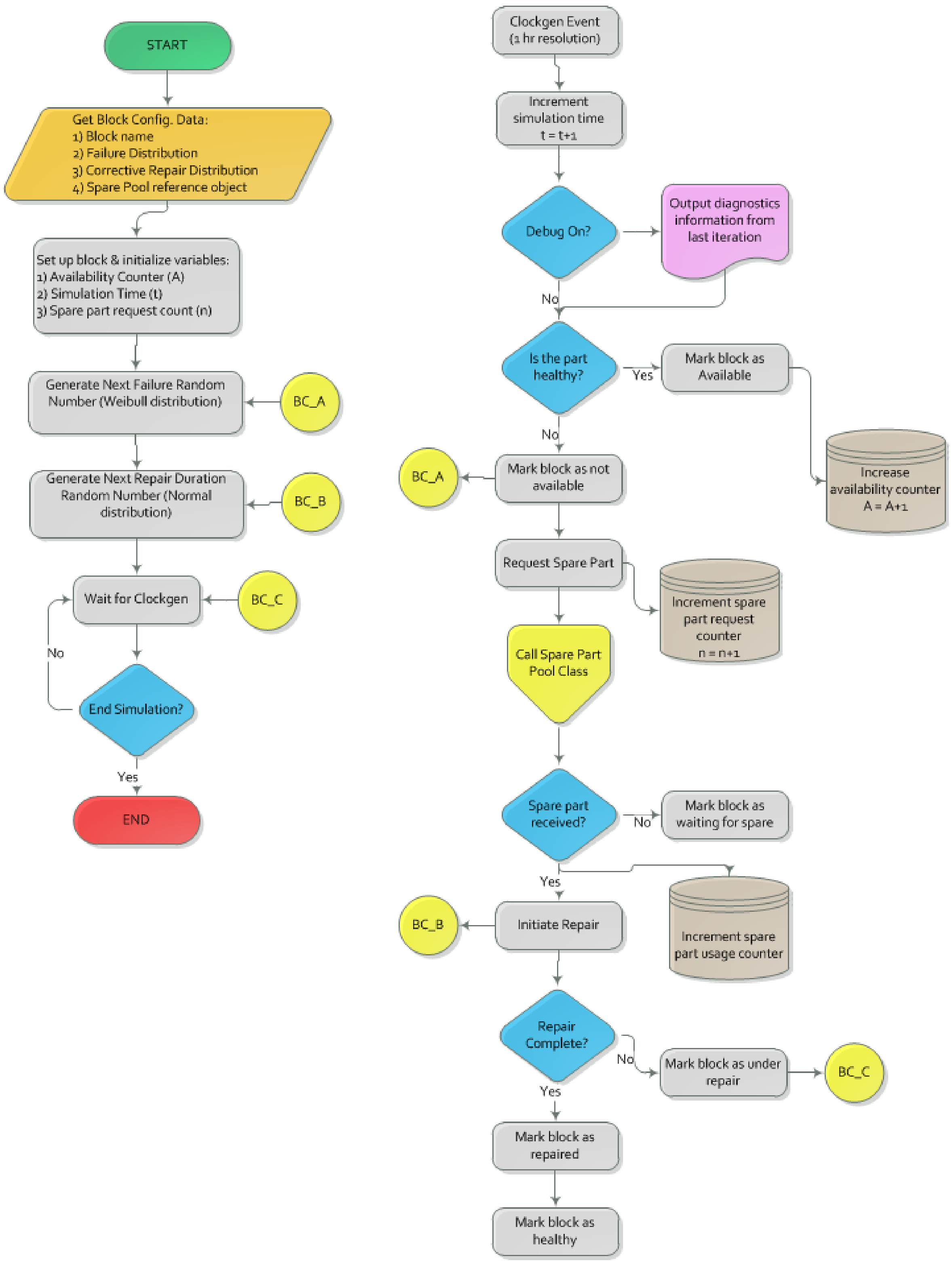
START
Get Block Config. Data:
1) Block name
2) Failure Distribution
3) Corrective Repair Distribution
4) Spare Pool reference object
Set up block & initialize variables:
1) Availability Counter (A)
2) Simulation Time (t)
3) Spare part request count (n)
Generate Next Failure Random Number (Weibull distribution)
BC_A
Generate Next Repair Duration Random Number (Normal distribution)
BC_B
Wait for Clockgen
BC_C
No
End Simulation?
Yes
END
Clockgen Event (1 hr resolution)
Increment simulation time t = t+1
Debug On?
Output diagnostics information from last iteration
No
Is the part healthy?
Yes
Mark block as Available
Increase availability counter A = A+1
No
BC_A
Mark block as not available
Request Spare Part
Increment spare part request counter n = n+1
Call Spare Part Pool Class
Spare part received?
No
Mark block as waiting for spare
Yes
BC_B
Initiate Repair
Increment spare part usage counter
Repair Complete?
No
Mark block as under repair
BC_C
Yes
Mark block as repaired
Mark block as healthy

Figure 12: Reliability Block Class

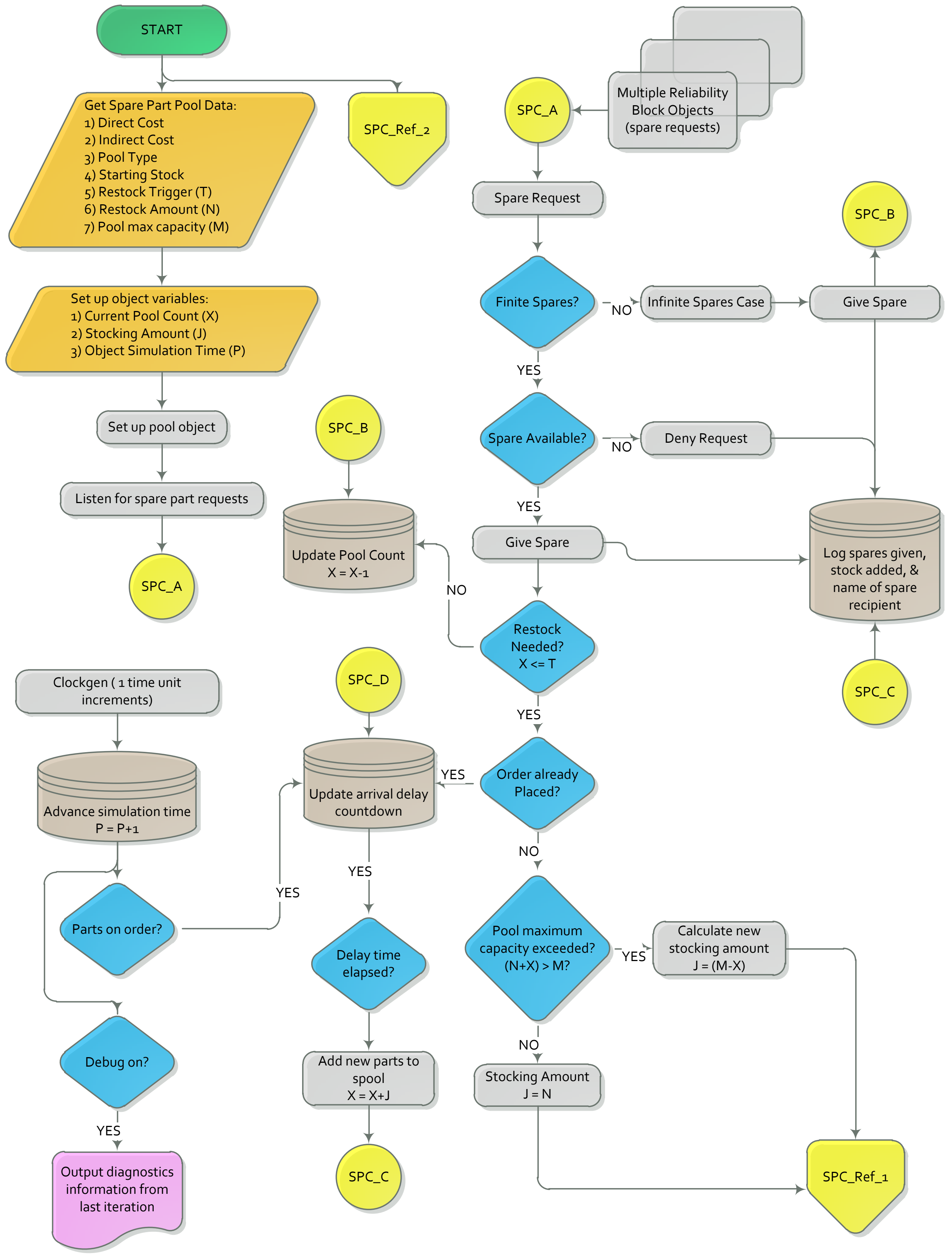
START
Get Spare Part Pool Data:
1) Direct Cost
2) Indirect Cost
3) Pool Type
4) Starting Stock
5) Restock Trigger (T)
6) Restock Amount (N)
7) Pool max capacity (M)
SPC_Ref_2
Set up object variables:
1) Current Pool Count (X)
2) Stocking Amount (J)
3) Object Simulation Time (P)
Set up pool object
Listen for spare part requests
SPC_A
Multiple Reliability Block Objects (spare requests)
SPC_A
Spare Request
Finite Spares?
NO
Infinite Spares Case
Give Spare
SPC_B
YES
Spare Available?
NO
Deny Request
YES
Give Spare
Log spares given, stock added, & name of spare recipient
SPC_C
SPC_B
Update Pool Count X = X-1
NO
Restock Needed? X <= T
YES
Order already Placed?
YES
NO
Pool maximum capacity exceeded? (N+X) > M?
YES
Calculate new stocking amount J = (M-X)
NO
Stocking Amount J = N
SPC_Ref_1
SPC_D
Update arrival delay countdown
YES
Delay time elapsed?
Add new parts to spool X = X+J
SPC_C
Clockgen (1 time unit increments)
Advance simulation time P = P+1
Parts on order?
YES
Debug on?
YES
Output diagnostics information from last iteration

Figure 13: Spare Parts Pool Class – Part 1

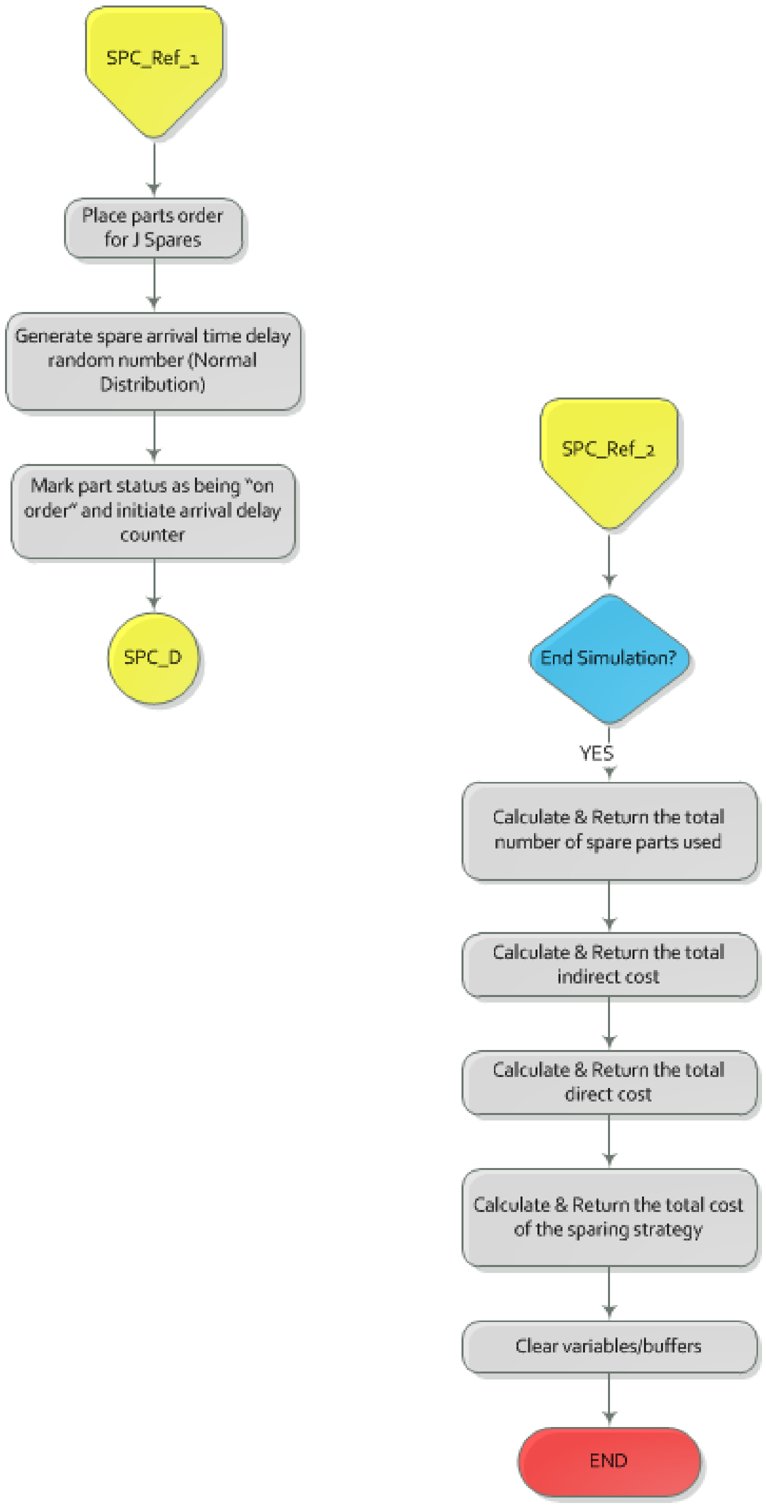
SPC_Ref_1
Place parts order for J Spares
Generate spare arrival time delay random number (Normal Distribution)
Mark part status as being "on order" and initiate arrival delay counter
SPC_D
SPC_Ref_2
End Simulation?
YES
Calculate & Return the total number of spare parts used
Calculate & Return the total indirect cost
Calculate & Return the total direct cost
Calculate & Return the total cost of the sparing strategy
Clear variables/buffers
END

Figure 14: Spare Parts Pool Class - Part 2

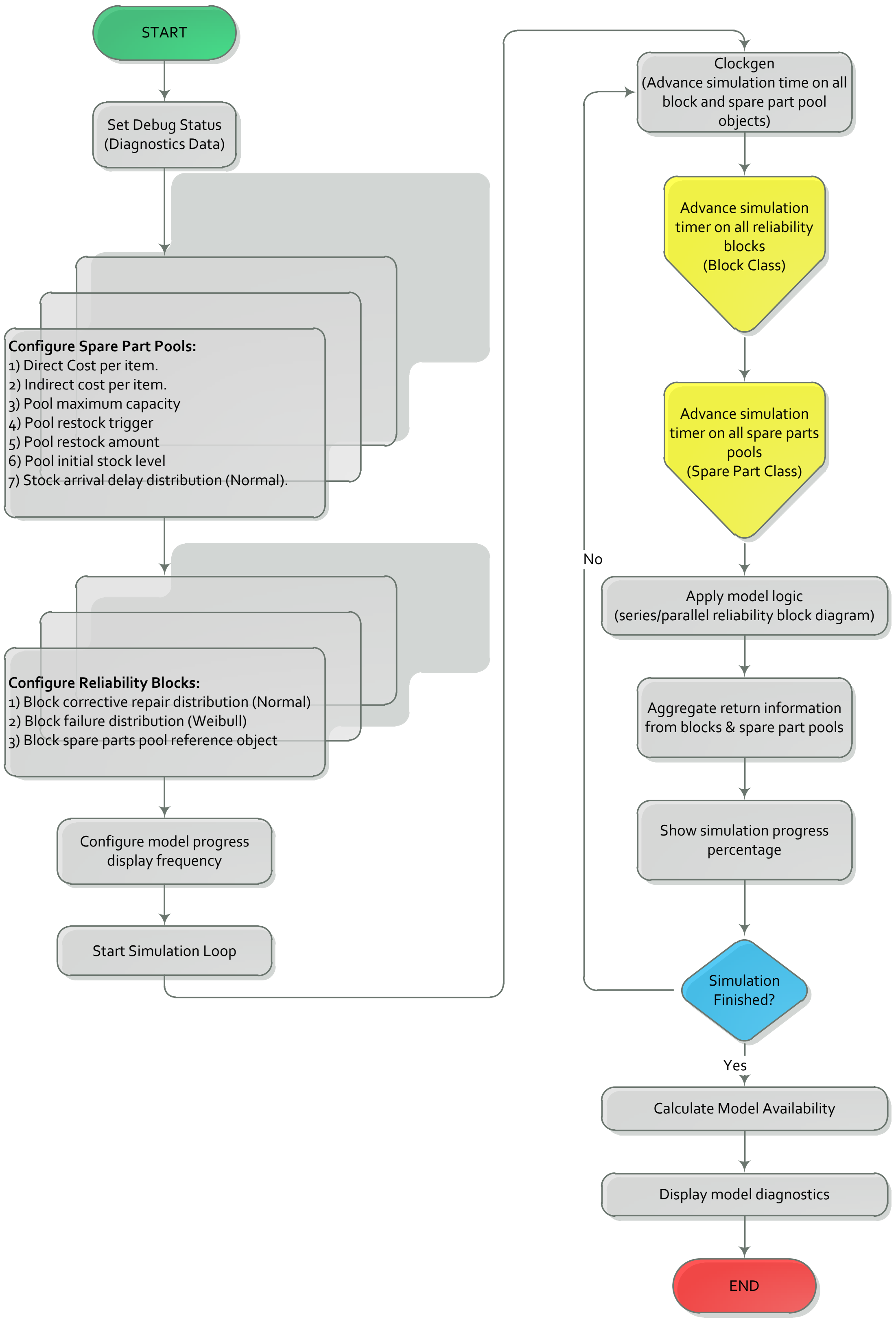
START
Set Debug Status (Diagnostics Data)
Configure Spare Part Pools:
1) Direct Cost per item.
2) Indirect cost per item.
3) Pool maximum capacity
4) Pool restock trigger
5) Pool restock amount
6) Pool initial stock level
7) Stock arrival delay distribution (Normal).
Configure Reliability Blocks:
1) Block corrective repair distribution (Normal)
2) Block failure distribution (Weibull)
3) Block spare parts pool reference object
Configure model progress display frequency
Start Simulation Loop
Clockgen (Advance simulation time on all block and spare part pool objects)
Advance simulation timer on all reliability blocks (Block Class)
Advance simulation timer on all spare parts pools (Spare Part Class)
Apply model logic (series/parallel reliability block diagram)
Aggregate return information from blocks & spare part pools
Show simulation progress percentage
Simulation Finished?
No
Yes
Calculate Model Availability
Display model diagnostics
END

Figure 15: Monte-Carlo Simulation Algorithm Process Flow Diagram

4.2 Key Assumptions

The following assumptions apply to the optimization algorithm presented in this report:

1. Simulation resolution of 1 hr. is sufficient for realistically simulating the spare parts pool policy lifecycle cost. The algorithm presented in this report is time unit independent; one minute resolution can be used if required. To save computation time, 1 hr. resolution will be used.
2. Equipment maintenance work orders are issued without delay upon failure.
3. Block maintenance priorities are in effect (first part to fail will get the spare part if multiple blocks are waiting for the same spare) – this means that "first come first served" priority sequence is in effect, the part that enters the repair queue first gets repaired first.
4. Repair time delay is applicable from the point when a spare part is available.
5. Direct cost includes initial stock level.
6. Indirect cost includes all spare parts in the pool at the end of each time unit (1 hr.). This includes the initial stock level.
7. Weibull failure distribution is used for modeling equipment failure time. Normal distributions are used for modeling corrective repair and stock arrival delay time.
    a. Most commonly used distributions in the field of reliability engineering are used in the model; it is possible to change distributions by making relatively small changes in the model.

8. Corrective repair time & stock arrival delay time normal distributions are trimmed to remove the possibility of generating negative numbers.
9. Failure time distribution, corrective repair time distribution, and stock arrival delay distributions are independent of each other.
10. Corrective maintenance results in equipment going back to "as good as new" state.
11. Equipment repair cannot commence till the required spare part is available.
12. Unlimited repair crews are available (repair is carried out at once after the spare part is available). If multiple parts are in failed state, repair is carried out simultaneously.
    a. Future work might incorporate workforce levels as an input to the optimization algorithm.
13. There is no delay in getting a spare part from the pool (no logistical spare part acquisition delay).
14. Time value of money is ignored (no discounted cash flow analysis for the purposes of calculating the sparing policy cost).
15. Removing a spare part from the stocking facility actually results in indirect cost reduction (such as would be the case with having offsite spares). In the case of onsite spares, sometimes removing the spare from storage facility actually does not have any impact on the indirect cost, the cost simply gets shifted to other spares stored in the facility (cost related to personnel, electricity etc. is still incurred as long as the onsite storage facility is functional).
16. A subsystem can be repaired while the system is in operation - repair on a subsystem can begin as soon as the appropriate spare parts are available.

17. No volume discounts are available for large spare part orders.

18. Sensitivity analysis around the cost of downtime will not be conducted.

    a. The cost of downtime (the unavailability penalty) can change with fluctuation in commodity prices (price of crude oil or potash as an example). Fluctuating prices can change the optimal spares level, this might be included future iterations of the model.

19. Imperfect spares or older discarded parts can be used in situations when the plant is waiting for new spares to arrive and the warehouse inventory level is at zero. This maintenance strategy is outside the scope of work for the current project (older discarded parts are normally only used as a stop-gap solution for short periods of time).

4.3 Key Constraints

The Monte-Carlo algorithm is subject to the following constraints:

1. The number of items that are added during the restock event is not less than the restock trigger amount.
2. Initial stock level is greater than the restock trigger amount.

4.4 Model Validation

Two simple test cases will be developed to validate the Monte-Carlo algorithm. A single block test case will be simulated for limited and unlimited spare scenarios and the results will be compared against *Reliasoft BlockSim* commercial reliability block simulation software (please see Appendix D). A multi-block test case will be simulated for a lifecycle of 500 hrs. for manual code validation – this entails going through the results manually to make sure that the desired output is being generated.

4.4.1 Multiple Reliability Block Test Case

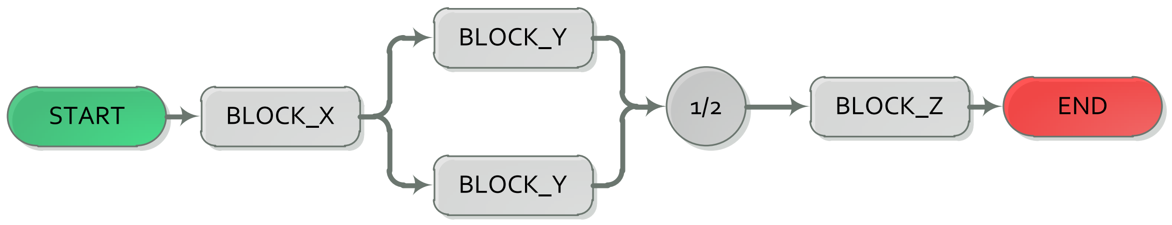

Figure 16: Multi-Block Test Case - Reliability Block Diagram

A multi-block test case was simulated for model validation using the following parameters:

	Block X	Block Y	Block Z
Failure distribution	Weibull (10,1.5)	Weibull (5,1.3)	Weibull (7,1.8)
Repair distribution	Normal (4,2)	Normal (3,1)	Normal (5,3)
Direct Cost of Spares	\$150	\$175	\$200
Indirect Cost	\$15	\$10	\$7
Restock trigger	3	2	1
Restock amount	5	4	3
Initial stock level	10	5	4
Stock arrival delay	N/A	N/A	N/A

Table 2: Multiple Reliability Block Case Parameters

Figure 16 shows a reliability block diagram for the problem; Figure 17 shows a fault tree model.

For the system to fail, either block X, or Block Z, or both Block Y's have to fail. Short failure and repair distributions are selected to allow for easy manual algorithm validation. A 500 iteration simulation output for both the limited and unlimited spare scenario is attached in the appendix E and has been manually checked for correctness by the author.

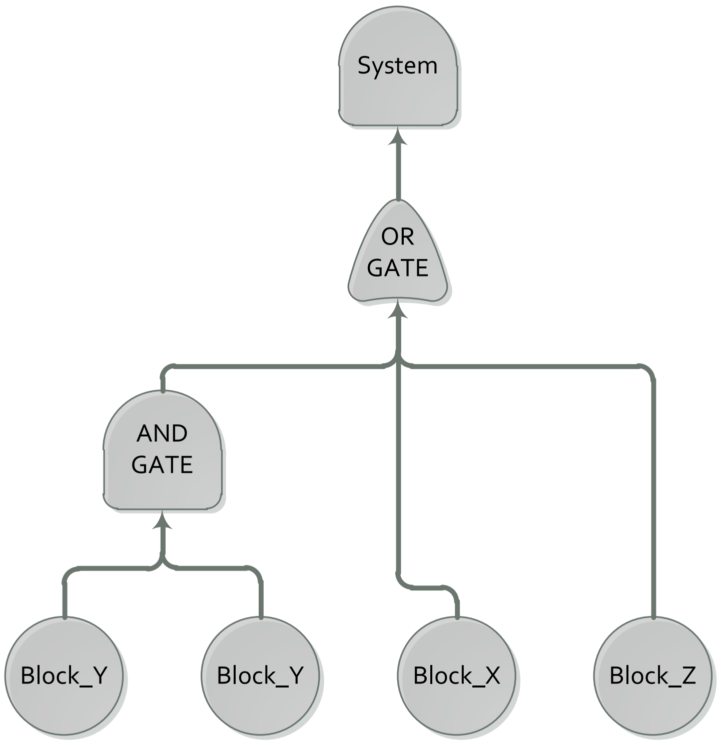

Figure 17: Multi-Block Test Case - Fault Tree Diagram

5 Numerical Application

The optimization model proposed in this report will be applied to a hypothetical high reliability system that contains redundant parts. The application will clearly demonstrate how a combined Monte-Carlo and Genetic Algorithm based approach can be taken to minimize lifecycle cost associated with spare parts storage. Figure 18 & Figure 19 show the reliability block diagram and the fault tree for the problem system.

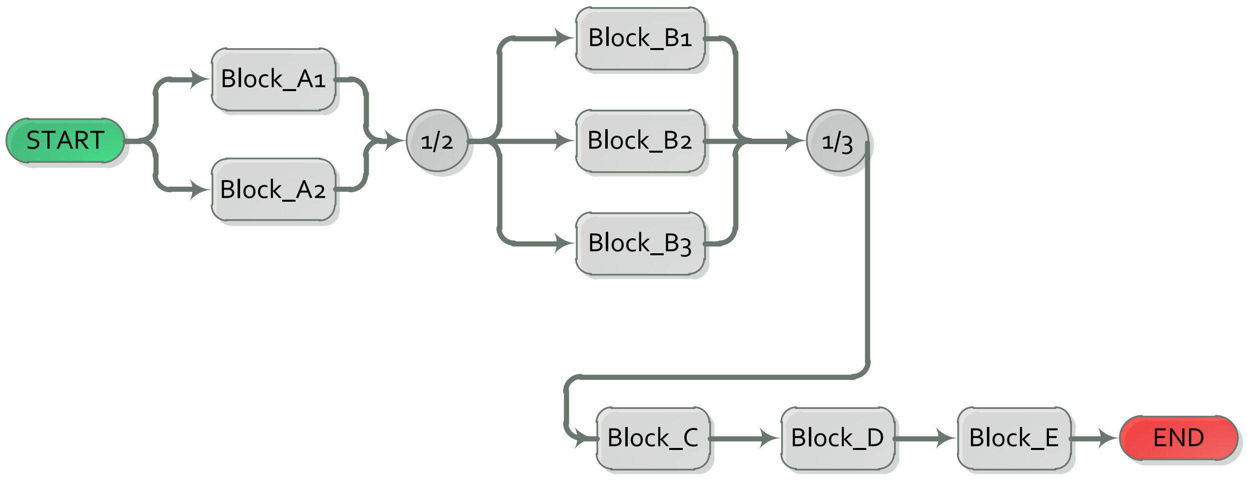

Figure 18: Model - Reliability Block Diagram

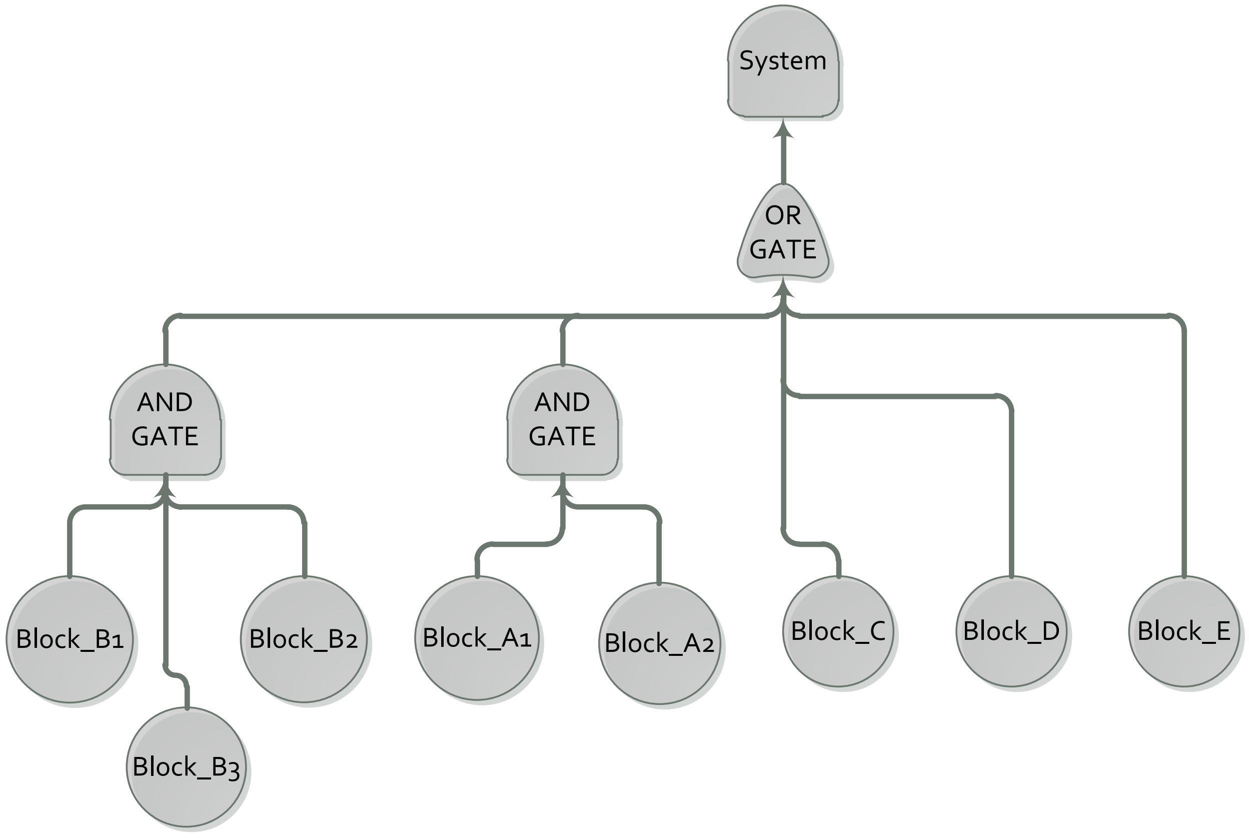

Figure 19: Model - Fault Tree Diagram

The following parameters will be used in the analysis:

	Block A	Block B	Block C	Block D	Block E
Failure distribution	W(2000,1.5)	W(1500,1.3)	W(3000,1.2)	W(3700,1.4)	W(5000,2)
Repair distribution	N(8,4)	N(6,2)	N(14,4)	N(12,8)	N(24,4)
Direct Cost of Spares	$500	$420	$4000	$5600	$6200
Indirect Cost ($/hr)	$0.80	$0.95	$1.90	$2.10	$3
Stock arrival delay	N(1650,200)	N(1900,300)	N(1750,230)	N(2000,200)	N(2100,230)
Cost of Downtime	$300 per hr.				

Table 3: Model Reliability Parameters

To get a baseline of how many spares would be used over the 10 year lifecycle, an unlimited spares case was simulated. The results show that the system would have 98.55%[7] availability and the spare usage profile would be as follows:

- Block A = 97
- Block B = 194
- Block C = 25
- Block D = 27
- Block E = 20

Knowing these results, we can estimate the total cost of running a "conservative" spares strategy by starting out with the baseline as our initial stock level. Table 4 shows the details of a conservative sparing strategy that starts out with all the required spare parts in stock, new spares will be added when the stock level is depleted to one unit.

[7] The same model yielded a 98.62% availability in Reliasoft BlockSim software (deviation of 0.1% from calculated results)

Spare Parts Pool Policy 1: Conservative					
	Block A	Block B	Block C	Block D	Block E
Restock trigger	1	1	1	1	1
Restock amount	10	15	5	5	5
Initial stock level	97	194	25	27	20

Table 4: Conservative Sparing Strategy Parameters

Next we can simulate an aggressive minimal sparing policy. The simulation will start off with a single spare part in stock for each spare parts pool. More spares will be ordered when the pool is empty. Table 5 shows the details of the “minimal” Sparing policy, the total cost of the minimal sparing policy is ~$800k compared to ~$17.7 million for the conservative sparing strategy (MSP is much lower than CSP because MSP does not have much indirect cost & at this stage the unavailability penalty is set at $0/hr.).

Figure 20 and Figure 21 show a comparison of the pool direct and indirect cost for both the conservative and minimal sparing policies. It can be seen that significant savings come from the minimal sparing strategy as indirect costs are dramatically reduced because of not having a large number of spare parts in storage for extended periods of time. The significant cost savings however come at a cost of having reduced availability, the conservative sparing strategy results in 98.54% availability as compared to 66.19% availability with the minimal sparing strategy. Figure 22 shows a total cost comparison between conservative and minimal sparing strategies – these two cases show two extremes of what can be done in terms of managing spares, the optimal or “best” case scenario is somewhere in the middle. The best case scenario is a scenario that balances indirect cost, which comes from having spares in the pool, and system availability. Genetic algorithms will be used to find a sparing strategy that gives us maximum availability, while maintaining a low pool indirect cost.

Spare Parts Pool Policy 2: Minimal Sparing Strategy					
	Block A	Block B	Block C	Block D	Block E
Restock trigger	0	0	0	0	0
Restock amount	2	3	1	1	1
Initial stock level	1	1	1	1	1

Table 5: Minimal Sparing Strategy Parameters

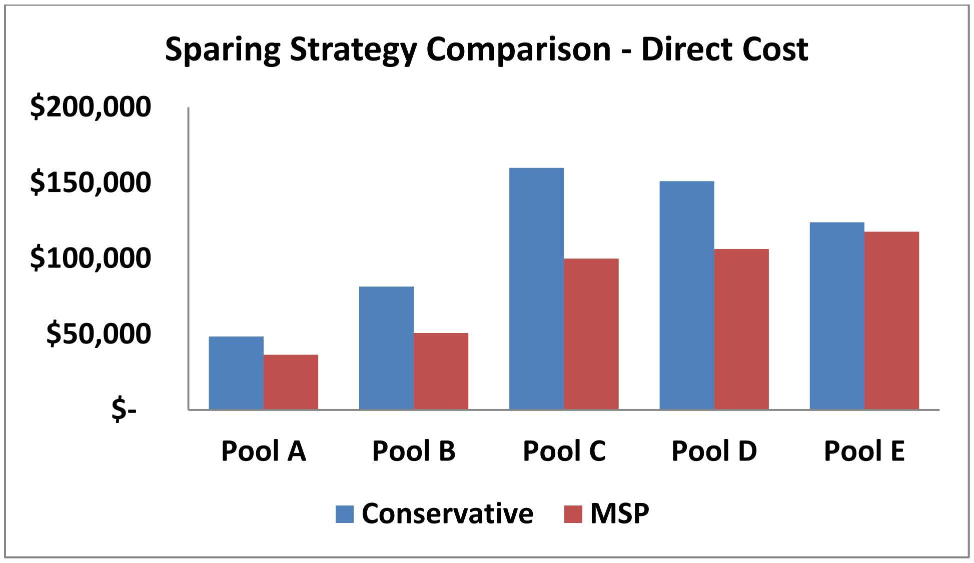

Figure 20: Sparing Strategy Comparison - Direct Cost

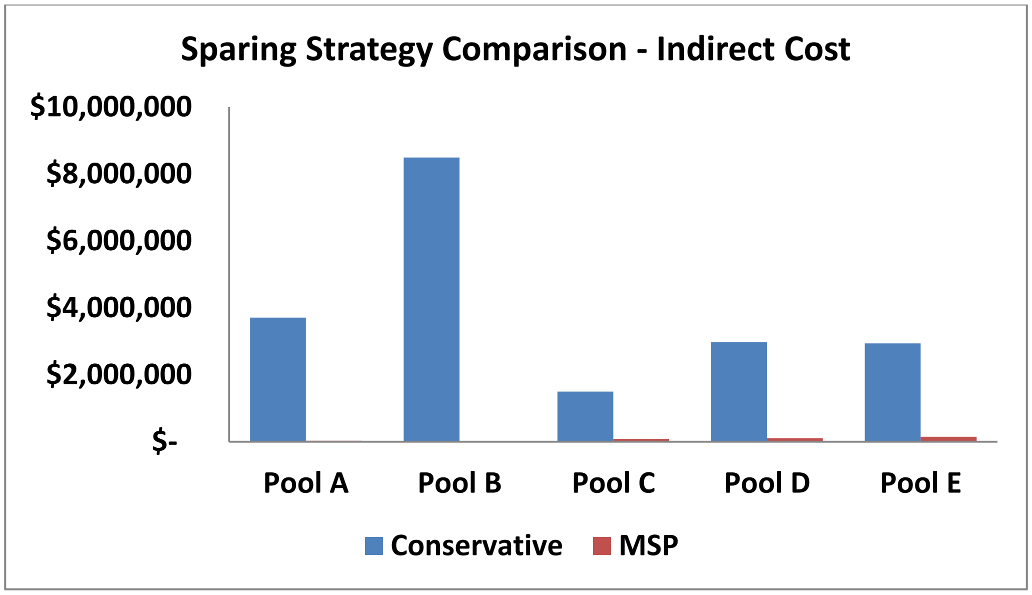

Figure 21: Sparing Strategy Comparison - Indirect Cost

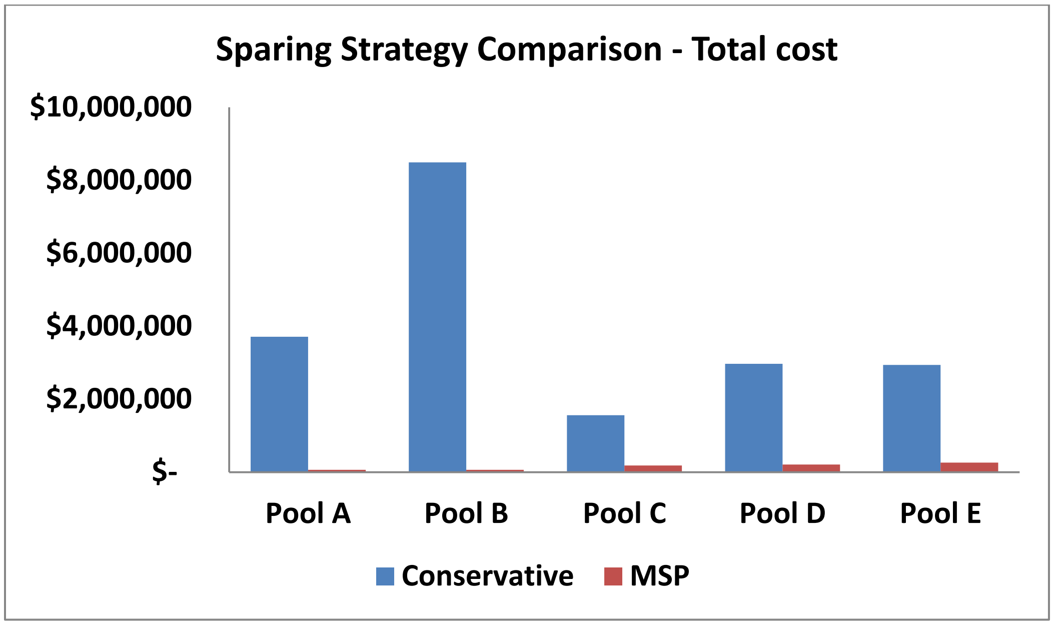

Figure 22: Sparing Strategy Comparison - Total Cost

5.1 Genetic Algorithm Constraints & Bounds

For the purposes of implementing the Genetic Algorithm optimization function, the following variable designations will be used:

	Pool A	Pool B	Pool C	Pool D	Pool E
Restock trigger	x(2)	x(6)	x(10)	x(14)	x(18)
Restock event – number of items to add	x(3)	x(7)	x(11)	x(15)	x(19)
Initial stock level	x(4)	x(8)	x(12)	x(16)	x(20)

Table 6: GA Variable Designations

Monte-Carlo algorithm constraints will result in the following linear inequality GA constraints:

$$-x(3) + x(2) \leq 1$$
$$-x(7) + x(6) \leq 1$$
$$-x(11) + x(10) \leq 1$$
$$-x(15) + x(14) \leq 1$$
$$-x(19) + x(18) \leq 1$$

$$-x(4) + x(2) \leq 1$$
$$-x(8) + x(6) \leq 1$$
$$-x(12) + x(10) \leq 1$$
$$-x(16) + x(14) \leq 1$$
$$-x(20) + x(18) \leq 1$$

The following boundary conditions will be used:

$$1 \leq x(2) \leq 97$$
$$1 \leq x(3) \leq 97$$
$$1 \leq x(4) \leq 97$$

$$1 \leq x(6) \leq 194$$
$$1 \leq x(7) \leq 194$$
$$1 \leq x(8) \leq 194$$

$$1 \leq x(10) \leq 25$$
$$1 \leq x(11) \leq 25$$
$$1 \leq x(12) \leq 25$$

$$1 \leq x(14) \leq 27$$
$$1 \leq x(15) \leq 27$$
$$1 \leq x(16) \leq 27$$

$$1 \leq x(18) \leq 20$$
$$1 \leq x(19) \leq 20$$
$$1 \leq x(20) \leq 20$$

The boundary conditions for each spare parts pool is derived from the unlimited spares case, based on the maximum number of spares that would be used if unlimited number of parts were available.

5.2 Results

Using a random starting population of 20 individuals, a crossover rate of 0.8, constraint dependent mutation rate, and an elite count of 2 individuals, a test case was simulated with cost of lost opportunity resulting from unavailability set at $0/hr. Algorithm stopping criteria of 60 hours was used to generate simulation results for 50 generations. Figure 23 shows the best, worst, and mean fitness scores coming out of the simulation – it can be seen that the mean objective function scores are converging around ~$820k. Table 7 shows the best individual - spare pool restock trigger, number of items added during restock event, and initial stock level are all converging towards 1. These results are quite predictable as this is essentially the minimal spare parts ordering scenario. With the penalty of decreasing availability set to $0/hour of unavailability, the algorithm should and does converge towards the minimal sparing strategy as there are no competing goals for the objective function. With no availability penalty in place, lower indirect cost simply results in better fitness which eventually leads to the lower boundary condition of 1 spare part in stock for each pool.

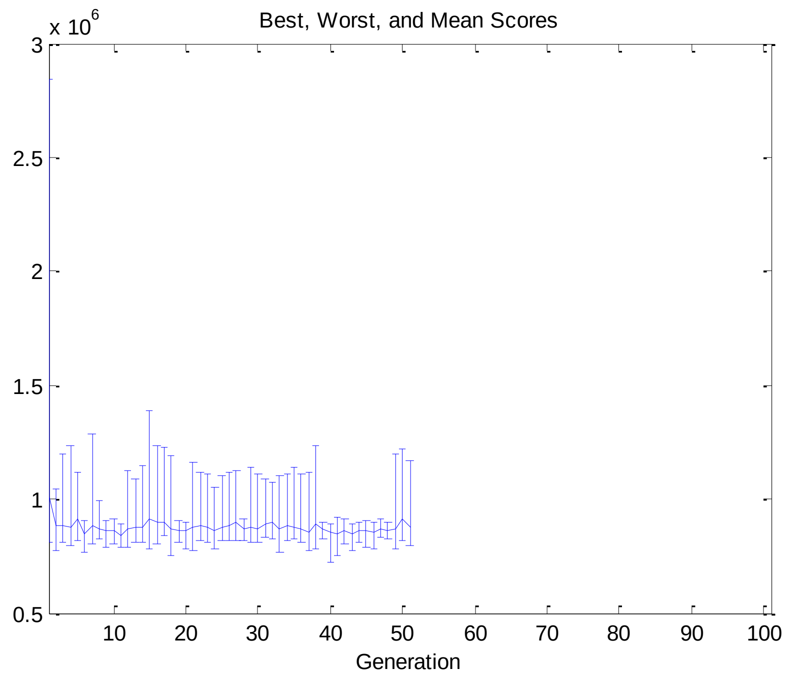

Figure 23: Test Case GA Results – Fitness Function Best, Worst, and Mean Scores

	Pool A	Pool B	Pool C	Pool D	Pool E
Restock trigger	1	1	2	1	1
Restock event – number of items to add	1	1	1	1	1
Initial stock level	1	1	3	1	1

Table 7: Test Case GA Results - Best Individual

A second case with unavailability penalty set at $300/hr. was simulated. Figure 24 shows the best, worst, & mean fitness function scores – it can be seen that the algorithm is approaching a mean of $5M as the total cost of the optimal[8] sparing policy with the best solution giving a value of $4M as the total cost of the optimal sparing strategy. With the cost of lost opportunity taken into account, the optimal sparing strategy results in ~35% savings over the minimal sparing scenario and ~80% savings over the conservative sparing strategy over a 10 year facility

[8] While the policy might not be global optimal (excessive amount of computation time would be required to run the simulation for 100+ generations), it is certainly a "very good" sparing strategy – significantly better than the minimal or conservative cases.

lifecycle. Based on these results, it is clear that significant savings can result from optimizing the spare parts strategy by using simulation in combination with global optimization methods such as genetic algorithms. In absence of having an optimal sparing policy available, in most cases the equipment operator would end up using the minimal sparing policy which is not always the best choice for long lead time spare parts.

Table 8 shows the optimal sparing strategy, taking the cost of lost opportunity into account – using an unavailability penalty of $300/hr. The optimal sparing policy specifies exactly when new spare parts should be ordered. Initial or starting out stock level is also specified along with the number of items to add during each restocking event. This sparing strategy is considerably more sophisticated than simply initiating spare part restocking based on a pre-existing calendar based stocking schedule (industry norm).

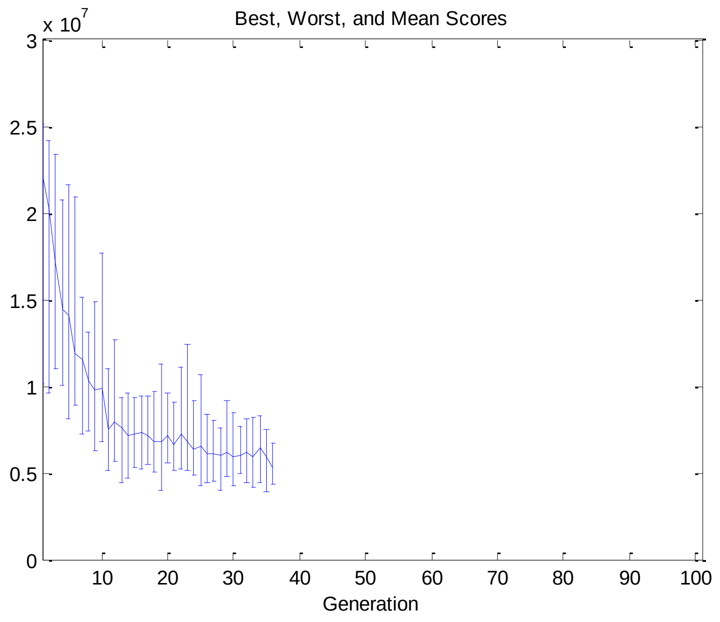

Figure 24: GA Results - Best, Worst, & Mean Scores

	Pool A	Pool B	Pool C	Pool D	Pool E
Restock trigger	1	1	2	1	1
Restock event – number of items to add	4	3	3	3	2
Initial stock level	2	5	3	2	2

Table 8: Optimal Sparing Strategy

6 Conclusion

In this report, a combined Mont-Carlo and Genetic Algorithm based optimization approach to finding an optimal spare parts policy has been presented, such a policy can result in significant lifecycle cost savings for equipment/plant operators. The proposed approach takes into account both the indirect and direct cost of having a spare part in the storage facility and balances the cost of lost opportunity resulting from decreased plant availability against the cost of maintaining spares in the warehouse. The algorithm presented in this report provides a simulation based stochastic solution to the problem of finding an optimal spare parts storage policy that can maximize profits by recommending the right level of spares that should be maintained to lower the spares storage cost while maximizing plant availability.

The proposed Monte-Carlo and Genetic Algorithm based optimization technique was applied to a combined series parallel system consisting of eight components; it was found that significant cost savings can result from following the proposed methodology. The lifecycle cost of conservative sparing strategy was ~$20M, lifecycle cost of minimal sparing strategy was ~$6M, and the lifecycle cost of the optimal sparing strategy was ~$4M. Table 9 shows the minimal sparing policy where the restocking is triggered when there are no spares left in the storage facility, minimal restocking amounts based on equipment count are used – for example there are two blocks of type A in the system, therefore the restocking amount is 2 units. Initial stock level of 1 unit is used to minimize equipment storage costs. Table 10 shows the conservative sparing strategy, this strategy calls for storing all the required spares up front, as such the initial stock level is high enough to accommodate all the failures the equipment will see during the mission time/lifecycle. Table 11 shows the optimal sparing strategy derived from Genetic Algorithms that results in maximum profits – minimal lifecycle spares cost while maintaining maximum plant availability.

	Block A	Block B	Block C	Block D	Block E
Restock trigger	0	0	0	0	0
Restock amount	2	3	1	1	1
Initial stock level	1	1	1	1	1

Table 9: Minimal Sparing Strategy

	Block A	Block B	Block C	Block D	Block E
Restock trigger	1	1	1	1	1
Restock amount	10	15	5	5	5
Initial stock level	97	194	25	27	20

Table 10: Conservative Sparing Strategy

	Block A	Block B	Block C	Block D	Block E
Restock trigger	1	1	2	1	1
Restock amount	4	3	3	3	2
Initial stock level	2	5	3	2	2

Table 11: Optimal Sparing Strategy

In the absence of having an optimal sparing strategy available, most equipment/plant operators tend to utilize a subjective sparing policy that is based on the minimal sparing strategy presented in this report. The optimal sparing strategy, which can be derived using the approach presented in this report, however can result in significant cost savings - 35% savings over minimal sparing strategy (and 80% savings over the conservative sparing strategy). These numbers can translate into millions of dollars of savings every year as the working capital invested in storage of spare parts can be quite significant for most large companies.

7 Bibliography

1. **Patterson, W J, Fredendall, L D and Kennedy, W J.** An Overview of Recent Literature on Spare Parts Inventories. *International Journal of Production Economics.* 2002, Vol. 76.

2. **Simchi-Levi, David, Kaminsky, Philip and Simchi-Levi, Edith.** *Designing & Managing the Supply Chain.* USA : McGraw-Hill Higher Education, 2003. pp. 38-40. 0072492562.

3. **Haddock, J and Mittenthal, J.** Simulation Optimization using Simulated Annealing. *Computers & Industrial Enginering Journal.* 1992, Vol. 22.

4. **Punniyamoorthy, Ganesh.** Optimization of Continuous-Time Production Planning using Hybrid Genetic Algorithms-Simulated Annealing. *International Journal of Advanced Manufacturing Technology.* 2004, Vol. 26.

5. **Yang, T and Kuo, Y.** Tabu Search Simulation Optimization Approach for Flow-Shop Scheduling with Multiple Processors. *International Journal of Production Research.* 2004, Vol. 42.

6. **Koller, Glenn.** *Risk Assessment and Decision Making in Business and Industry 2nd Edition.* Boca Raton : Chapman & Hall, 2005.

7. **Dimov, Ivan T.** *Monte Carlo methods for applied scientists.* Singapore : World Scientific Publishing Co. Pte. Ltd., 2008.

8. *The Beginning of the Monte Carlo Method.* **Metropolis, N.** 1987, JNL, pp. 125-130.

9. **Newman, M. E. and Barkema, G. T.** *Monte Carlo Methods in Statistical Physics.* New York : Oxford University Press, 1999. p. 26.

10. **Segre, Emilio.** *Enrico Fermi Physicist.* Chicago : The University of Chicago Press, 1970. p. 86.

11. **Hammersley, J. M. and Handscomb, D. C.** *Monte Carlo Methods.* Norwich : Fletcher & Son Ltd, 1964.

12. **Humphreys, Kenneth K.** *Project and Cost Engineers' Handbook, Fourth Edition.* New York : Marcel Dekker, 2005. p. 224.

13. **Cigolini, Roberto D. and Desmukh, Abhijit V.** *Recent Advances in Maintenance and Infrastructure Management.* London : Springer-Verlog, 2009.

14. **Sillion, Francois X. and Puech, Claude.** *Radiosity & Global Illumination.* San Francisco : Morgan Kaufmann Publishers, Inc., 1994.

15. **Reilly, Robert F. and Schweihs, Robert P.** *The Handbook of Business Valuation and Intellectual Property Analysis.* New York : McGraw Hill, 2004.

16. **Charles, Darryl, et al., et al.** *Biologically Inspired Artificial Intelligence for Computer Games.* Hershey : IGI Publishing, 2008.

17. **Sanso, Brunilde and Soriano, Patrick.** *Telecommunications Network Planning.* Norwell : Kluwer Acedemic Publishers, 1999. pp. 138-139.

18. **Sobol, Llya M.** *A Primer for the Monte Carlo Method.* London : CRC Press, 1994.

19. **Mitchell, Melanie.** *An Introduction to Genetic Algorithms.* New York : Massachusetts Institute of Technology, 1998. 0262133164.

20. **Gen, Mitsuo and Cheng, Runwei.** *Genetic Algorithms & Engineering Optimization.* United States : John Wiley & Sons, 2000. 047131531.

21. **Steele, Nigel C. and Neruda, Roman.** *Artificial Neural Nets and Genetic Algorithms.* New York : Springer Computer Science, 2001. 3211836519.

22. **Sivanandam, S N and Deepa, S N.** *Introduction to Genetic Algorithms.* New York : Springer Berlin Heidelberg, 2009. 3642092241.

23. **Wright, Alden H. and Vose, Michael D.** *Foundations of Genetic Algorithm.* New York : Springer-Verlag, 2005. 3540272372.

24. **Rardin, Ronald L.** *Optimization in Operations Research.* USA : Prentice Hall, 1997. 0023984155.

25. **Hillier, Frederick.** *Introduction to Operations Research.* USA : McGraw-Hill Science, 2009. 0077298349.

26. **Haupt, Randy L. and Haupt, Sue Ellen.** *Practical Genetic Algorithms.* USA : Wiley-Interscience, 2004. 0471455652.

27. **Langdon, William B. and Poli, Riccardo.** *Foundations of Genetic Programming.* USA : Springer, 2002. 3540424512.

28. **Coley, David A.** *An Introduction to Genetic Algorithms for Scientists and Engineers.* USA : World Scientific Publishing Company, 1997. 9810236026.

29. **Kelton, W D and Law, A M.** *Simulation Modeling and Analysis.* New York : McGraw-Hill Higher Education, 1991.

30. *Multiobjective spare part allocation by means of genetic algorithms and Monte Carlo simulation.* **Marseguerra, Marzio, Zio, Enrico and Podofillini, Luca.** Milan, Italy : Reliability Engineering and System Safety, 2004, Vol. 87.

31. *A stocking policy for spare part provisioning under age based preventative replacement.* **Kabir, Zohrul and Al-Olayan, Ahmed S.** Riyadh, Saudi Arabia : European Journal of Operational Research, 1993, Vol. 90.

APPENDICES

Appendix A: Approximating the Value of PI Using a Monte Carlo Approach

```
%Monte Carlo Simulation - Calculating the Value of PI

%Amit S. Aulakh - Jan 1, 2010


%USER DATA

Circle_Radius = 5;

Circle_Center = [0,0];

nSampleSize = 100; %number of points to drop


%VARS

Axis_Limit = Circle_Radius; %square side is 2xradius

iPointsInCircle = 0; %count of points inside circle

iPointsOutsideCircle = 0; %count of points outside the circle (ie in the square but not in circle)

mnCircleProperties = [Circle_Center(1,1) Circle_Center(1,2) Circle_Radius];


%create a figure window

figure


%set axis limits

axis(Axis_Limit*[-1 1 -1 1]);
```

```
%lock square axis

axis('square');


hold on;

grid on;


%draw axes + in solid black line ('-k')

plot(xlim,[0 0],'-k',[0 0],ylim,'-k');


%draw the circle

t=0:.01:2*pi;

xc=Circle_Radius*cos(t)+Circle_Center(1);

yc=Circle_Radius*sin(t)+Circle_Center(2);

plot(xc,yc,'y-','erasemode','none','linewidth',2)

fill(xc,yc,'y'); %fill with yellow color


%draw a square - (x,y) pairs define connectiong points

plot(Axis_Limit*[-1 -1 1 1 -1],Axis_Limit*[1 -1 -1 1 1],'-k','linewidth',2)



%drop points & count how many land inside & outside of the circle

for i=1:nSampleSize
```

```
%generate random x,y coordinates

[x,y]= Gmtry_Generate_Random_XY_Coord(-1*Axis_Limit, Axis_Limit, -1*Axis_Limit,

Axis_Limit);


%check if point is inside circle

nRetVal = Gmtry_IsPointInsideCircle(mnCircleProperties, [x,y]);

if nRetVal == 2

%point in circle

iPointsInCircle = iPointsInCircle+1;


%plot point

plot(x, y, '.r', 'MarkerSize',20 );


elseif nRetVal == 3

%point on circle

iPointsInCircle = iPointsInCircle+1;


plot(x, y, '.b', 'MarkerSize',20 );

elseif nRetVal == 4

%point not in circle

iPointsOutsideCircle = iPointsOutsideCircle+1;
```

```
plot(x, y, '.g', 'MarkerSize',20 );

end


end %for


%calcualte the value of PI

%value of pi = (Area of circle / area of square)*4

calculated_pi = (iPointsInCircle/nSampleSize)*4;


%compare with actual value of PI

CalcPI_deviation = ((calculated_pi - pi)/pi)*100;


%hold off

hold off;
```

Appendix B: Approximating the Area of an Irregular Shape using a Monte Carlo Approach

The Monte-Carlo method can also be used for calculating the area of an irregular shape. This approach is often not used for calculating areas of two dimensional geometry because better estimating techniques exist, namely the "quadrature formulas" (18). However, the same concept can be extended to calculate volumes of multidimensional space, for these applications the Monte Carlo approach is highly effective and is often the only practical numerical method available (18).

The area of an irregular shape can be calculated using the Monte Carlo methodology by inscribing the irregular shape, with an unknown area, inside of a square, or any other shape which allows us to compute the area of the outer object easily. Figure 25 shows an irregular shape S inscribed within a square of known dimensions – statistical sampling can now be used to estimate the area S by dropping K randomly sampled points inside the square. If N points land inside the circle, than the area of the irregular shape (S) can be estimated as:

$$Area\ of\ an\ irregular\ shape\ (S) = \left(\frac{N}{K}\right) \times Area\ of\ Square$$

The accuracy of this approach is highly dependent on the number of points dropped, or the number of iterations. The higher the number of randomly sampled points inside the square, the more accurate the estimated area S will be. For the purpose of numerically demonstrating this concept in action, area of a circle will be calculated – the same technique can apply to any irregular shape, the circle has been used so that we can compare the calculated result against actual known area of a circle. Figure 26 shows the algorithm logic diagram (code listing is attached in Appendix C).

The attached code listing generates a circle radius (r) randomly and then inscribes it in a square of side length 2r. A starting point of 50 iterations was used; the number of iterations was then incremented by 100 till the calculated area had an error of less than 5%. Error was calculated as:

$$Error\ (\%) = \frac{Abs(Area_{Actual} - Area_{Calculated})}{Area_{Actual}} \times 100$$

Figure 27 shows a graphical output from the Monte Carlo algorithm; randomly sampled coordinate pairs inside the circle are shown in red. Figure 28 shows a comparison of the calculated area value against the known actual area value. It can be seen that as the number of iterations was increased from 10 to 500, the percentage deviation of calculated value from the actual value dropped from 50% to 2% mark.

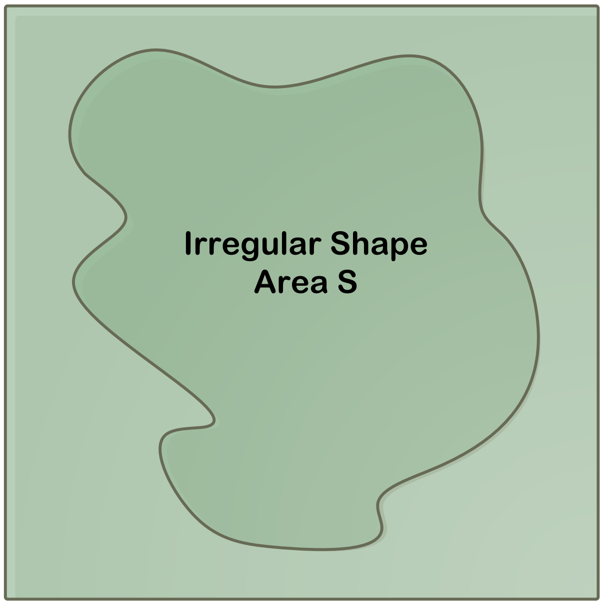

Figure 25: Irregular Shape S Inscribed in a Square of known dimensions

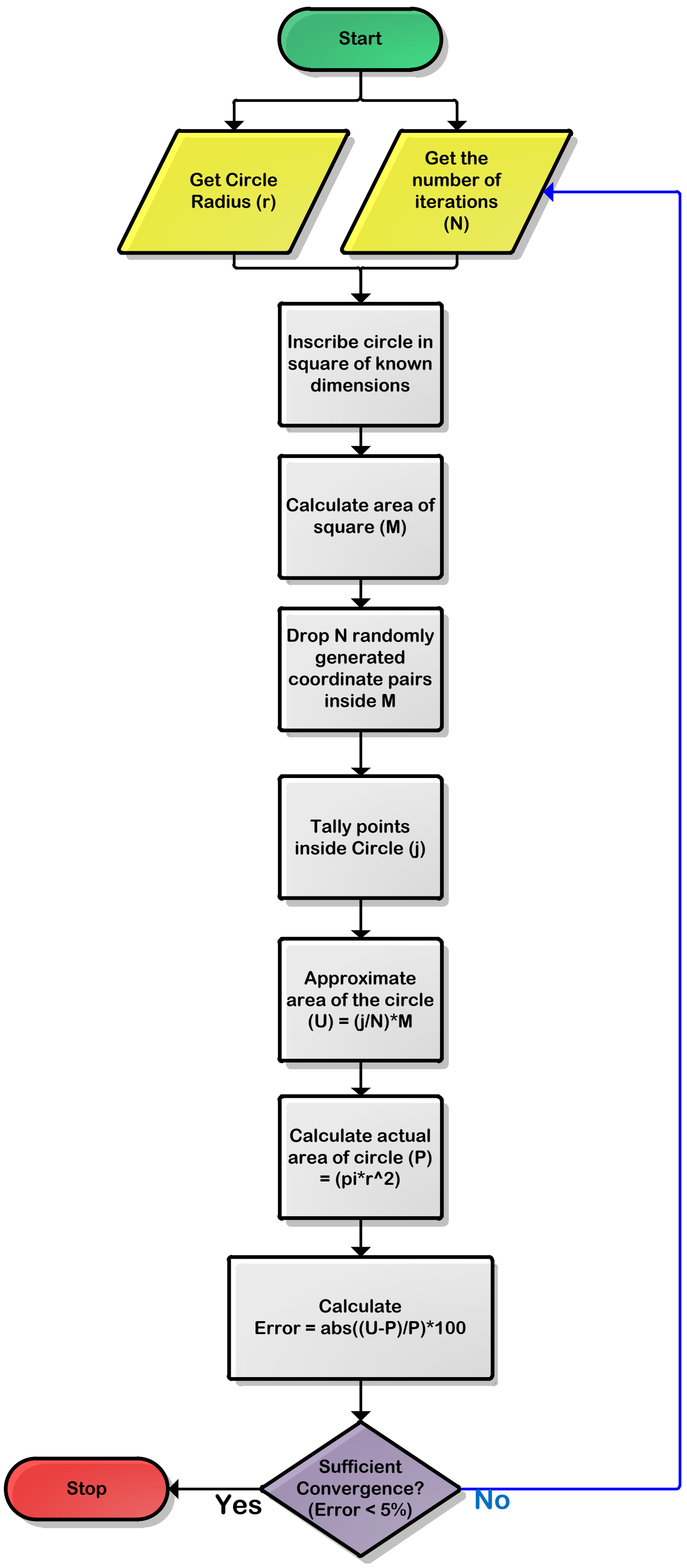
Start
Get Circle Radius (r)
Get the number of iterations (N)
Inscribe circle in square of known dimensions
Calculate area of square (M)
Drop N randomly generated coordinate pairs inside M
Tally points inside Circle (j)
Approximate area of the circle (U) = (j/N)*M
Calculate actual area of circle (P) = (pi*r^2)
Calculate Error = abs((U-P)/P)*100
Sufficient Convergence? (Error < 5%)
Yes
No
Stop

Figure 26: Flowchart - Approximating Area of an Irregular Shape (Circle)

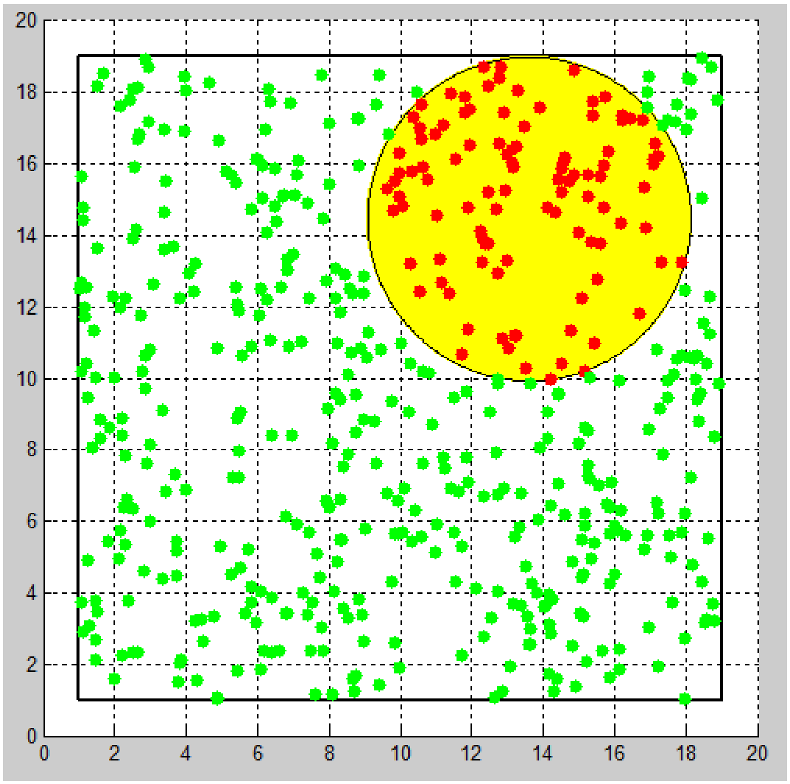

Figure 27: Calculating Area of a Circle using the Monte-Carlo Approach

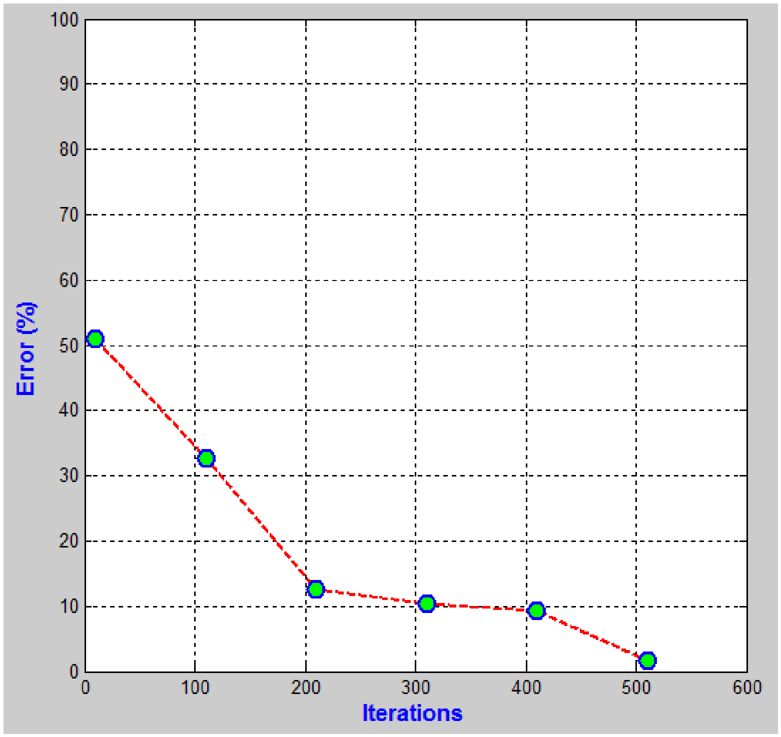

Figure 28: Calculating Area of a Circle - Error vs. Number of Iterations

Appendix C: Irregular Shape Area Approximation (Algorithm)

```
%Monte Carlo Anaysis Demo

%Calculating Area of a irregular shape (circle)

%Amit S. Aulakh - 2009


iAxesLimits = 20;

iLenSquareSide = iAxesLimits-2;

iMinCircRadius = iLenSquareSide/4; %circle diameter must be atleast sidelen/2

iCirclePlcmtAnimationFrames = 15;


%square offsets from x & y axes

xo = 1; %x offset

yo = 1; %y offset


%x & y ranges for points inside the square

nXRange = [xo, iLenSquareSide+xo];

nYRange = [yo, iLenSquareSide+yo];


%eg

%starting sample size = 50

% increment by 50

% number of increments = 3
% Sample size = 50, 100, 150

nStartSampleSize = 10; %this is the starting sample size

nSampleSizeIncrement = 100;

nNumOfSampleSizeIncrements = 6;


%current effective smaple size

nSampleSize = nStartSampleSize; %number of points to drop


%point drop delay

fpointdropdelaysecs = 0; %0.0005;


%==========================================================

% Initialize Vars

%==========================================================

iPointsInCircle = 0;

iPointsOutsideCircle = 0;


%==========================================================

%new figure window

%==========================================================

%create figure object (individual window)

%get screen size
```

```
scrsz = get(0,'ScreenSize');

%full screen

hwndFigure = figure('Name','Monte-Carlo Simulation example by Amit S. Aulakh -

2009','NumberTitle','off', 'OuterPosition',[0 0 scrsz(3) scrsz(4)]);

%turn off figure menubar

set(hwndFigure,'MenuBar','none');


hwndAxesError = subplot(1,2,1);


%set(hwndAxesError,'title','none');

plot([0]); %initialization

title('Error vs Number of Iterations'); %must set title AFTER you've plotted something

xlabel('Iterations');

%set title and axis label fonts & colors

set(get(gca,'Title'),'Color','b', 'FontSize', 16, 'FontWeight', 'bold');

set(get(gca,'XLabel'),'Color','b', 'FontSize', 12, 'FontWeight', 'bold');

set(get(gca,'YLabel'),'Color','b', 'FontSize', 12, 'FontWeight', 'bold');


hwndAxesDisplay = subplot(1,2,2);


%===========================================================================

%animate circle placement
```

```
%==========================================================================

%select axes

axes(hwndAxesDisplay);

for i=1:iCirclePlcmtAnimationFrames


%clear figure

%clf;

cla; %clear axes


%set displayed axes limits

 %axis([xmin xmax ymin ymax])

axis(iAxesLimits*[0 1 0 1]);


% maintaings square axes (ie when u resize window, a circle won't become a

% ellipse) - aspect ratio will be maintained

axis('square');


%Retain current graph in figure (all the commands will apply to the same

%figure) otherwise only the last command will show on figure

hold on

grid on;

%draw axes + in solid black line ('-k')
```

```
plot(xlim,[0 0],'-k',[0 0],ylim,'-k');


%draw a square (in positive xy plane only)

%xo = 1; %x offset

%yo = 1; %y offset

 l = iLenSquareSide; %length of side

plot([xo xo+l xo+l xo xo],[yo yo yo+l yo+l yo],'-k','linewidth',2)


SqLLCorner_x = xo;

SqLLCorner_y = yo;


%draw chart title

title('Iteration # - of -');

set(get(gca,'Title'),'Color','b', 'FontSize', 16, 'FontWeight', 'bold');


%get randoml generated circle center point & radius

[RetVal_CircCntrX RetVal_CircCntrY RetVal_CircRadius] =

fPlaceCircleInsideSquare(SqLLCorner_x, SqLLCorner_y, iLenSquareSide, iMinCircRadius);

%draw a circle

r=RetVal_CircRadius; %radius of circle

center=[RetVal_CircCntrX,RetVal_CircCntrY]; %center of circle

t=0:.01:2*pi;
```

```
xc=r*cos(t)+center(1);

yc=r*sin(t)+center(2);

plot(xc,yc,'y-','erasemode','none','linewidth',2)

 fill(xc,yc,'y'); %fill with yellow color


%pause for 0.30 sec

pause(0.15);

end %for


%get final circle properties

mnCircleProperties = [center(1) center(2) r];


% Run iterations & update error display


%Loop code will be executed 5 times

for n = 1:nNumOfSampleSizeIncrements


%=====================================================================

% Initialize variables for loop

%=====================================================================

%if we're doing first run - clear vars

if n == 1
```

```
clear mcError;

clear mcIterations;

end


iPointsInCircle = 0;

iPointsOutsideCircle = 0;


clear sqArea;

clear circle_radius;

clear ActualCircleArea;

clear ratio_incircle_div_totalpoints;

clear calc_area;

clear error_percent;

%============================================================================

%drop points

%============================================================================

%--------------------------------------------------------------------------------

%Setup display

%--------------------------------------------------------------------------------

%select the right axes

axes(hwndAxesDisplay);

%clear axes
```

```
cla;
%set displayed axes limits
%axis([xmin xmax ymin ymax])
axis(iAxesLimits*[0 1 0 1]);


% maintaings square axes (ie when u resize window, a circle won't become a
% ellipse) - aspect ratio will be maintained
axis('square');
%Retain current graph in figure (all the commands will apply to the same
%figure) otherwise only the last command will show on figure
 hold on
grid on;
%draw axes + in solid black line ('-k')
plot(xlim,[0 0],'-k',[0 0],ylim,'-k');
%draw a square (in positive xy plane only)\
%xo = 1; %x offset
%yo = 1; %y offset
 l = iLenSquareSide; %length of side
plot([xo xo+l xo+l xo xo],[yo yo yo+l yo+l yo],'-k','linewidth',2)
SqLLCorner_x = xo;
SqLLCorner_y = yo;
%draw chart title
```

```
 stemptitle = sprintf('Iteration # %d of %d', n, nNumOfSampleSizeIncrements);

title({stemptitle});

set(get(gca,'Title'),'Color','b', 'FontSize', 16, 'FontWeight', 'bold');

%draw the circle

r=mnCircleProperties(3); %radius of circle

center=[mnCircleProperties(1),mnCircleProperties(2)]; %center of circle

t=0:.01:2*pi;

xc=r*cos(t)+center(1);

yc=r*sin(t)+center(2);

plot(xc,yc,'y-','erasemode','none','linewidth',2)

 fill(xc,yc,'y'); %fill with yellow color

%-----------------------------------------------------------------------

%DROP POINTS

%-----------------------------------------------------------------------

for i=1:nSampleSize


%generate random x,y coordinates

[x,y]= Gmtry_Generate_Random_XY_Coord(nXRange(1,1), nXRange(1,2), nYRange(1,1),

nYRange(1,2));

%check if point is inside circle

nRetVal = Gmtry_IsPointInsideCircle(mnCircleProperties, [x,y]);

if nRetVal == 2
```

```
%point in circle

iPointsInCircle = iPointsInCircle+1;

%plot point

plot(x, y, '.r', 'MarkerSize',20 );

elseif nRetVal == 3

%point on circle

iPointsInCircle = iPointsInCircle+1;


plot(x, y, '.b', 'MarkerSize',20 );

elseif nRetVal == 4

%point not in circle

iPointsOutsideCircle = iPointsOutsideCircle+1;


plot(x, y, '.g', 'MarkerSize',20 );

end

pause(fpointdropdelaysecs);

end %for

%-----------------------------------------------------------------------

%Turn hold off

%-----------------------------------------------------------------------

hold off
```

```
%=============================================================================

% Calculate error

%=============================================================================

sqArea = iLenSquareSide * iLenSquareSide;

circle_radius = mnCircleProperties(3);

ActualCircleArea = pi*circle_radius*circle_radius;

ratio_incircle_div_totalpoints = iPointsInCircle/nSampleSize;

calc_area = ratio_incircle_div_totalpoints * sqArea;

error_percent = (abs(ActualCircleArea-calc_area)/ActualCircleArea)*100;

disp(sprintf('square area = %d', sqArea));

disp(sprintf('poitns in circle = %d', iPointsInCircle));

disp(sprintf('total points = %d', nSampleSize));

disp(sprintf('ratio incircle vs total = %d', ratio_incircle_div_totalpoints));

disp(sprintf('Actual circle area = %d', ActualCircleArea));

disp(sprintf('calculated circle area = %d', calc_area));

disp(sprintf('error prcnt = %d', error_percent));


%=============================================================================

% Store error data

%=============================================================================

%mcError = error_percent; %[10 20 30];

%mcIterations = nSampleSize; %[5 6 7];
```

```
mcError(1,n) = error_percent; %[10 20 30];

mcIterations(1,n) = nSampleSize; %[5 6 7];


%========================================================================

% Plot updated error

data%========================================================================

%select axes

axes(hwndAxesError);

plot(mcIterations, mcError         ,...

'LineStyle'        ,   '--'    ,...

'LineWidth'        ,   2       ,...

 'Marker'          ,   'o'     ,...

'MarkerSize'       ,   10      ,...

'MarkerFaceColor'   ,   'g'     ,...

'MarkerEdgeColor'   ,   'b'     ,...

 'Color'           ,   'r'     ...     %line color

); %x,y


%set title & axes names

title('Error vs Number of Iterations'); %must set title AFTER you've plotted something

xlabel('Iterations');
```

```
ylabel('Error (%)');


%trun grid on

grid on;


%set axes limits

ylim([0 100]);


%set title and axis label fonts & colors

set(get(gca,'Title'),'Color','b', 'FontSize', 16, 'FontWeight', 'bold');

set(get(gca,'XLabel'),'Color','b', 'FontSize', 12, 'FontWeight', 'bold');

set(get(gca,'YLabel'),'Color','b', 'FontSize', 12, 'FontWeight', 'bold');

%=====================================================================

%Update data for next run

%=====================================================================

%increment sample size

nSampleSize = nSampleSize + nSampleSizeIncrement;

disp(sprintf('n = %d', n));


 %break; %to get out of loop

end %for
```

Appendix D: Code Validation – Single Reliability Block Case

Figure 29 shows a reliability block diagram for the single block test case; Figure 30 shows a fault tree representation of the same model. The overall system is represented as an "and" gate, therefore when block 01 fails, the whole system fails.

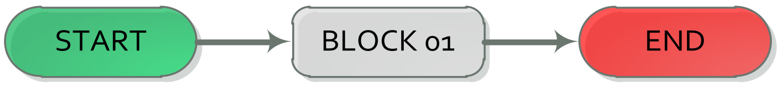

Figure 29: Single Block Test Case: Reliability Block Diagram

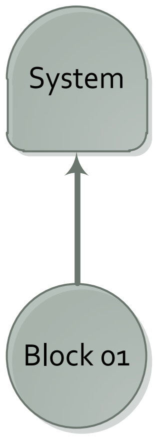

Figure 30: Single Block Test Case: Fault Tree Diagram

An unlimited spares case will be simulated using the following parameters:

- Weibull failure distribution (Eta = 5, Beta = 1.3)
    - An unrealistically short life distribution is being used for the block so that we can use a lot of spare parts during the simulation lifecycle (for code validation).
- Normal corrective repair distribution (Mean = 8, Standard deviation = 4)

- Lifecycle duration of 10 years (87600 hrs.)

Figure 31 shows the results of the simulation run. Availability calculated by the algorithm was 37.21% compared to 36.46% calculated by *BlockSim*, the number of spares used was 6686 by the algorithm, compared to 6924 by *BlockSim*. The overall error is in the range of 2 – 4% which is quite reasonable.

A limited spares scenario was also simulated using the following parameters:

- Normal corrective repair distribution (Mean = 4, Standard deviation = 2)
- Pool maximum capacity of 7 spares
- Restock trigger = 3 spares
- Restock amount = 6 spares
- Initial stock = 6 spares
- Stock arrival delay = 0

Figure 32 shows the results of the test case, availability error was on the order of 0.3%, the number of spare parts used were off by 4%.

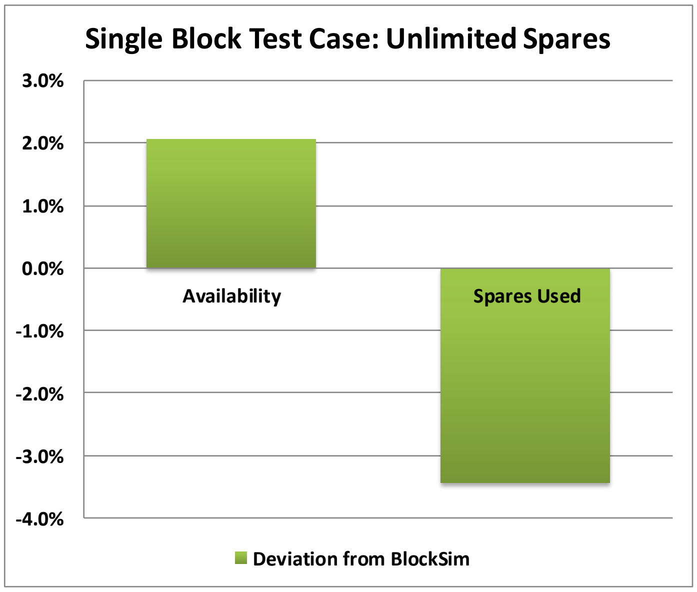

Figure 31: Single Block Test Case – Unlimited Spares validation Results

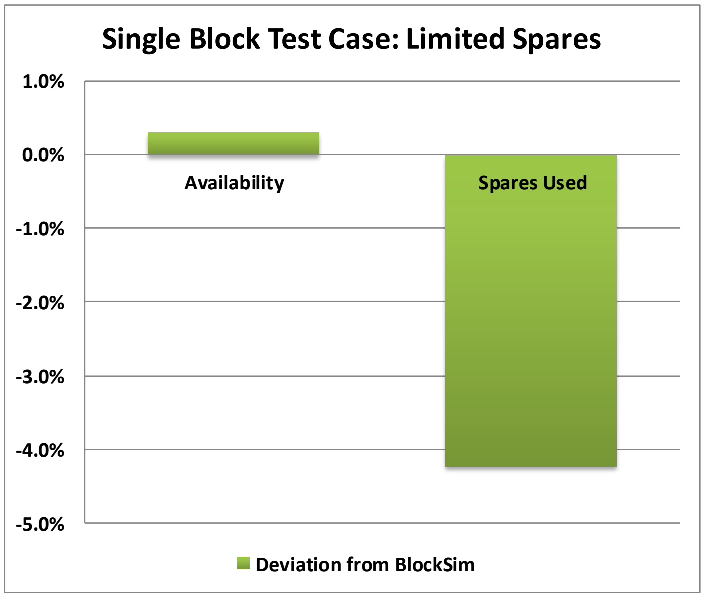

Figure 32: Single Block Test Case - Limited Spares Validation Results

Appendix E: Multiple Reliability Block Case (Output Validation)

Limited Spares Case (output terminated at 60 iterations – code was manually checked for 500 iterations):

Block X	Block Y1	Block Y2	Block Z	Sys Availability
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
0	1	0	1	0
0	1	0	1	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	1	1	0	0
0	0	1	0	0
0	0	1	0	0
1	0	1	0	0
0	0	1	0	0
0	1	0	0	0
0	1	0	1	0
0	0	0	1	0
1	0	0	0	0
0	0	0	0	0
0	1	0	0	0
0	1	1	0	0
0	1	0	0	0
0	1	0	0	0
1	1	0	0	0
1	1	1	0	0
1	1	1	0	0
1	1	1	1	1
0	1	1	1	0
0	1	1	1	0
0	1	0	1	0
0	1	0	1	0
0	1	0	1	0
1	0	1	1	1

1	0	1	1	1
1	0	1	1	1
1	0	1	1	1
1	1	1	0	0
1	1	1	0	0
1	1	1	0	0
1	1	1	0	0
1	1	1	0	0
1	1	1	0	0
1	0	1	0	0
1	0	1	0	0
1	0	0	1	0
1	0	0	1	0
1	1	0	1	1
1	1	0	1	1
1	0	1	1	1
1	0	1	1	1
1	0	1	0	0
1	1	1	0	0
1	1	1	0	0
1	1	1	0	0
1	1	1	1	1
0	0	0	1	0
0	0	0	1	0
1	0	0	1	0
1	0	1	0	0
1	1	1	0	0