8-1-2010

Analysis and optimization methods of graph based meta-models for data flow simulation

Jeffrey Harrison Goldsmith

Analysis and Optimization Methods of Graph Based Meta-Models for Data Flow Simulation

by

Jeffrey Harrison Goldsmith

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science
in Industrial Engineering

Supervised by

Professor Dr. Michael E Kuhl
Department of Industrial and Systems Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
August 2010

Approved by:

Dr. Michael E Kuhl, Professor
Thesis Advisor, Department of Industrial and Systems Engineering

---

Dr. Moises Sudit, Adjunct Associate Professor
Committee Member, Department of Industrial and Systems Engineering

Thesis Release Permission Form

Rochester Institute of Technology
Kate Gleason College of Engineering

Title:

Analysis and Optimization Methods of Graph Based Meta-Models for Data Flow Simulation

I, Jeffrey Harrison Goldsmith, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Jeffrey Harrison Goldsmith

Date

Dedication

To my parents, this research would not have been possible without your support.

Acknowledgments

I must acknowledge my research advisers: Dr. Michael Kuhl and Dr. Moises Sudit. Dr. Michael Kuhl has spent many hours working with me to create the methods outlined in this document, and many more hours helping me edit and express the content of this thesis! Dr. Moises Sudit's experience and knowledge of operations research helped me complete my work and achieve successful results. I am grateful that both you were on my thesis committee!

This work would not have been possible without previous research completed by Greg Tauer, I appreciate the time Greg has spent helping me understand his work. Additionally, I must acknowledge Kevin Costantini; his work writing software to extract the graph structure from Ptolemy II models has enabled much of the experimentation detailed in this thesis.

Finally, I must also thank my friends who have helped me throughout this research.

Abstract

Analysis and Optimization Methods of Graph Based Meta-Models for Data Flow Simulation

Jeffrey Harrison Goldsmith

Supervising Professor: Dr. Michael E Kuhl

Data flow simulation models are often used for the modeling and analysis of complex dynamic systems. Although traditional analysis methods such as design of experiments or optimization methods can be applied directly to data flow simulation models, applying these techniques to complex systems with large numbers of controllable inputs and performance measures may not be able to be completed in an acceptable amount of time. This research focuses on the development of optimization and analysis methods applied to graph-based meta-models of data flow simulation models. The goal of this research is to create a method that can efficiently determine the values of controllable system input variables that will yield user-specified system output performance measure values. The methodology utilizes an existing graph-based meta-modeling technique that elicits the graph structure of the underlying data flow simulation model. To enable goal-oriented optimization on the elicited graph, edge weights are determined by performing experimental sampling and utilizing a regression model to each of the nodes in the elicited graph. In the case of nonlinear input-output relationships, a method which provided piecewise linear edge weights is used. Finally, mathematical programming formulations are developed to conduct the goal-oriented optimization. An experimental performance evaluation is conducted and illustrates the capability of the method to efficiently provide estimates of system inputs that will result in desired values of system performance measures.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Analysis of large real life systems is often a complicated process. A useful tool for analysis of these systems is computer simulation. In general, a simulation can be viewed as a collection of inputs that produce one or more outputs. By modeling these real life systems accurately as simulations, one can perform experiments on the model that reflect the behavior of the real system. There are benefits associated with analyzing a simulation model instead of a real life system. In particular, experimenting on simulation models is useful when the corresponding real system experiment is expensive or dangerous.

Generally, simulations are used to answer "what if" questions. **What** happens to a some outputs **if** some inputs are changed, is a general "what if" question for simulations. Analysis of this type has strong practical significance, as the effects of an input on an output are not always clear.

Consider a manufacturing system with many operations where raw materials are machined and heat treated multiple times until a complete finished product is produced. Plant management can control many aspects of the production process. For example the system capacity, types of materials, processing rate of machines and temperature of ovens can all have an effect on the final product. In a simulation model of the system, all of these parameters would most likely be controllable input factors.

At the end of the system where final product rolls off the production line operators often measure a variety of performance metrics such as the total number of parts produced, the number of defective parts, and the total time parts spent in the system. Upper management

might be concerned with the cost per part and the total cost of the system. All of these performance metrics would typically be reported as outputs of the simulation model.

Changing the input values to the simulation should result in varied output values. Changing an input to effect an output value is common practice in simulation modeling. In the manufacturing example, perhaps the modeler wants to see if more parts can be manufactured by changing machine settings. Maybe the modeler wants to investigate the effects of cost reduction efforts on production capacity. Clearly, these are "what if" questions.

Another question common in the domain of operations research is "how to". These questions refer to optimization methods; finding the best solution to a problem subject to constraints. **How** do I change the input values of my simulation **to** achieve a specific output, is a general "how to" question for simulations.

Simulation optimization methods can be used to find combinations of input values that yield a specific output. It can be said that simulation optimization allows the analyst to answer not only "what if", but also "how to" [1]. Instead of manually trying many different input combinations, one can run an optimization algorithm to try and find a good solution. In the manufacturing example, the modeler may want to minimize total system cost while producing at least as many parts as the current system. This is a "how to" question; the modeler is asking how does one decrease cost while maintaining production rates.

Data flow simulation is one of many computer simulation modeling paradigms. A variety of social, economic, and engineering systems can be accurately represented using data flow models. A notable characteristic of data flow models is that their underlying structure is a directed graph. This structure is notable as many efficient algorithms and analysis methods have been developed specifically for graphs in the domain of graph theory.

This thesis outlines an optimization method for data flow simulations based on their underlying graph structure. There are many steps to the method implemented. First, the

graph structure is elicited from the data flow simulation. Then, then a fast sampling experiment is performed for each node on the graph. A weight is then generated that relates the magnitude of inputs to a node to its own output. Next, all the gathered data is put through an optimization model, a linear or mixed integer program. Finally, the results are verified to assess the performance of the method.

As stated above, a large variety of complex real life systems can be modeled as data flow simulations. An optimization method for data flow simulations should enhance the ability of analysts who model their systems as data flow simulations. If the optimization method is efficient, then it will hold high practical significance.

Chapter 2

Problem Statement

Large, complex simulation models often take a relatively long time to run. Trying to obtain a specific output from the model can be difficult as the effects of various inputs on various outputs are often unclear. Methods that can predict input values that yield a specific output are desirable, as the time required to analyze large simulation models is often excessive. Techniques that accomplish this task are generally referred to as simulation optimization methods.

In small simulation models, one can change the input values and execute the simulation until a desired output is obtained. This iterative process is acceptable in small models, since the execution time of the model is low and there are few user controllable input values. However, in large models this process would likely be far too time consuming.

Other popular methods used to optimize simulations include meta-heuristics such as simulated annealing, genetic algorithms, and tabu search [1, 11]. Statistical processes like response surface methods, or designed experiments are also common. Most efficient heuristics still require multiple executions of the entire model. Executing large simulation models can take a long time so unfortunately even efficient heuristics may still result in running times that are too long to be practically useful. Statistical methods such as designed experiments will also require many simulation executions for large models due to the number of run configurations needed for the experiment.

Most existing simulation optimization methods have long run times when applied to

large, complex simulation models. As such, there is a need for efficient simulation optimization techniques capable of quickly solving large models.

The goal of this thesis is to produce an efficient simulation optimization method for graph based meta-models of data flow simulations. Data flow simulation models all have the same underlying structure, a directed graph. The purpose of the graph based meta-model is efficiency. Using the graph based meta-model should be efficient as unnecessary complexities from the data flow model will be removed prior to optimization. Mathematical programs, either linear or mixed integer will be used to optimize the meta-models generated from data flow simulations. The purpose of the optimization methods is to find input values to data flow models that yield, or get as close as possible to, user specified goal values.

The quality of the methods suggested in this thesis will be measured by three parameters: run time, precision, and accuracy. An efficient method should have a low run time. The faster the optimization methods execute, the better! A precise method will produce output values close to the user specified values. A fast optimization method that provides useless results would most likely not help anyone. An accurate method will provide reliable results. Accuracy refers to the predicted input values actually yielding the predicted output. An accurate, precise, and fast method will be a valuable analysis tool for data flow simulation models.

Chapter 3

Literature Review

3.1 Overview

The methods in this research use techniques from many different fields. Simulation optimization methods are examined first and a variety of existing techniques are discussed. The majority of existing simulation optimization procedures are designed for discrete event simulation; the adaptation of these methods for data flow simulation are additionally discussed. Data flow simulations are investigated, and the differences between data flow and synchronous data flow paradigms are examined. Designed experiments and regression analysis are briefly explained, as they directly relate to the information abstraction portion of the methodology. The last review section covers mathematical programming techniques that are used in later in this thesis. In this chapter, the gap between existing research and the need for this work is shown.

3.2 Simulation Optimization

Many techniques for optimizing stochastic simulations come from deterministic operations research theory [1]. Some of the procedures are stochastic approximation methods (SAM), response surface methodology (RSM), and heuristic methods. Gradient search methods are general tools used to solve non-linear programs. SAMs use recursive functions to reach maximum or minimum values of theoretical regression equations corresponding to

response surfaces. Each step of the SAM requires a computation of the gradient vector, often a costly computation. SAMs need many simulation replications to compute gradients; they may not be practical in large systems [16]. RSMs fit regression equations of varying degree based on simulation output, and do not require gradients. Although RSMs require fewer replications than SAMs, they can be more sensitive to problem structure [1]. In general, these methods have been directly adapted from a mix of statistics and both non-linear and integer programming theory; using heuristics when complete optimization would be too costly.

Many techniques used in commercial packages are meta-heuristics like genetic algorithms, tabu search, and neural networks. Two key differences in heuristic algorithms that separate the many choices involve deciding when the current solution is optimal and how to proceed to the next point if the current state is not optimal [10].

A different method is to transform discrete event simulation models into mathematical programs [14]. This is not the first attempt at a simulation transformation method. Reduction of event graphs into analytical problems is possible, and analysis techniques from integer programming solves them [18]. Formulations of buffer allocation problems with flow lines and finite capacities are provided as an integer program, linear programming relaxation, and a stochastic program. The results of this research were promising, and a very important concept was the use of the LP relaxation to identify promising areas in the solution space of the more difficult problems.

Clearly a significant amount of research has revolved around simulation optimization, yet the majority is focused on discrete event models. Many systems can be modeled as discrete event simulations, so there is little surprise that most research is based on the discrete event paradigm. Current simulation optimization methods may apply to data flow simulations, but the topic has not yet been addressed.

3.3 Data Flow Simulations

Synchronous data flow simulation is a specific implementation of the data flow languages in simulation [12]. The unique point of synchronous data flow is that the number of tokens produced from an actor is equal to the number that actor consumes. This fact allows the SDF complier to build a schedule of all token flow in advance, significantly reducing the computational overhead commonly associated with data flow simulations. The reduction in required computational overhead can greatly decrease the running time of an SDF compared to a normal data flow model.

Since most SDF models are not sensitive to time, there is no explicit representation of time. However, many SDF models represent time by the number of iterations. One iteration of a data flow simulation consists of all actors firing their required number of tokens once. This concept is necessary when feedback loops are present in the model. Cycles create dependencies that must be broken before a model can start execution. Commonly, a delay type actor is needed to break these dependencies. Without a delay actor, other actors in the model would be waiting for a data token which would never arrive! Delay actors fire a specific token (typically with a value of 0) in the 0th iteration of the model, to break these cycles.

The simulation software used in this thesis is Ptolemy II, a free software package which supports many simulation paradigms including discrete event, data flow, and synchronous data flow [5]. Ptolemy is written in Java and is completely open source. This makes Ptolemy very flexible since anyone can create actors for specific tasks as they need. The flexibility also means a model can be called directly from Java applications, integrating the simulation into other larger or more specific applications.

Linear signal flow graphs (LSFG) are the basis for some constraints in linear programming formulations detailed in chapters later in this thesis. A linear signal flow graph is composed of vertices and edges, and the sum of all input edges to a vertex equal the value

of that vertex [8]. When an SDF model contains actors that only perform linear functions on their inputs, then the SDF model will operate similarly to an LSFG.

3.4 Statistical Analysis

Previous work related to analysis of synchronous data flow simulation models involved performing designed experiments on each actor in the simulation to determine the relative main effect of each input on each output [20]. Two aspects of this work are directly used in this thesis; the graph reduction and experimental sampling. Figure 3.1 shows an example Ptolemy II SDF model, and Figure 3.2 shows that model's elicited graph structure.

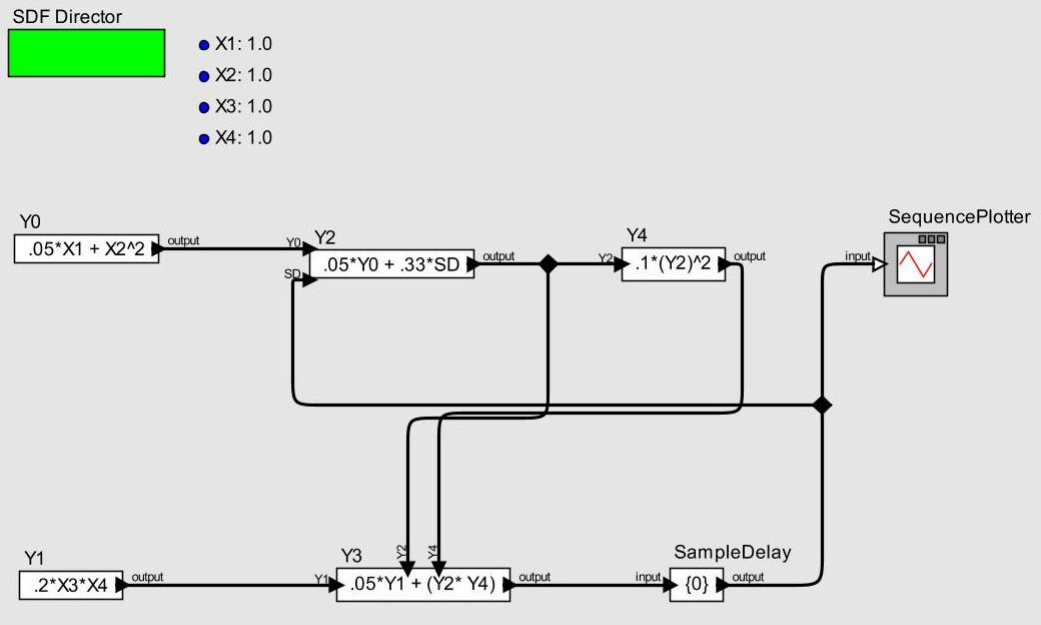

Figure 3.1: Example Ptolemy II Model

Regression analysis is a technique used to fit curves to data[15]. Given a matrix of control variables and a vector of response variables, linear least squares regression finds coefficients for each control variable that minimize the sum of squared error. These fitted

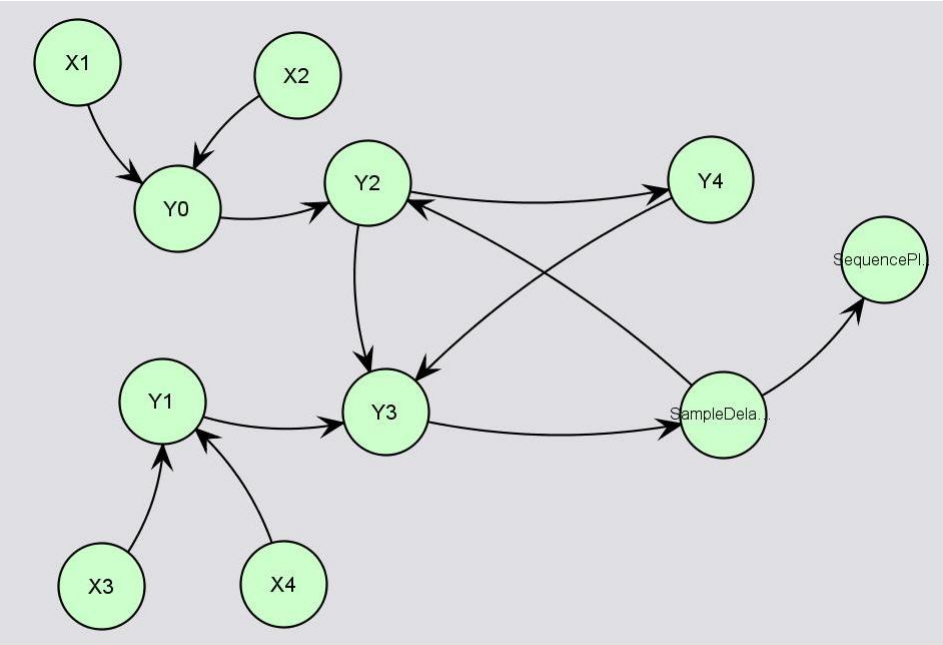

Figure 3.2: Graph Elicitation of Figure 3.1

coefficients can then be used to predict a response, within the range of the control data. Although one can use the regression coefficients to predict values outside the observed range, there is no guarantee the coefficients accuracy. The generic form of a regression equation is shown below equation (3.1):

$$y = \beta_0 + \beta_1 x_1 + \cdots + \beta_n x_n + \epsilon \tag{3.1}$$

The intercept is β_0, control variables $x_1 \ldots x_n$, fitted regression coefficients $\beta_1 \ldots \beta_n$, and an error term ϵ . An important concept is the coefficient of determination, or R^2; which gauges how well a regression model is fit. High R^2 values indicate a good fit where as low values do not explain the data well. The adjusted R^2 is a modified R^2 that takes into account the number of variables fitted. This can be important in large models since the R^2 value is artificially inflated by adding variables.

All statistical calculations necessary for this work are executed using R, a free language for statistical computing [21]. An extension to R called the JRI was written to allow Java programs to call R functions, and retrieve results.

3.5 Mathematical Programming

Operations research (OR) refers to many tools and methods that are generally used to optimize systems. One of the main methods to within OR is mathematical programming. This technique involves representing a system as a set of decision variables, subject to constraints. Most math programs have objective functions; these functions determine when a solution is optimal. Some of the most common types of math programs are shown in table 3.1:

Table 3.1: Common Types of Mathematical Programs

- Linear Program (LP)
- Integer Program (IP)
- Binary Integer Program (BIP)
- Mixed Integer Program (MIP)
- Non-Linear Program (NLP)
- Non-Linear Mixed Integer Program (NLMIP)

A linear program refers to a math program where all variables are real (continuous) and constraints and objective functions are linear (the general formulation is shown below in (3.2)) [17]. LP's are the easiest optimization problems to solve (out of those listed in table 3.1). Many efficient algorithms exist to solve LP's and a globally optimal solution is guaranteed to be found. A globally optimal solution is the set values for the decision variables that produce the smallest minimum or largest maximum, subject to the defined constraints. A feasible solution is one that satisfies all defined constraints. Constraints in math programs can have one of the following three operators: $\geq$, $\leq$, or $=$. Typically

variables are listed on the left hand side of the operator, with data parameters on the right hand side.

Integer programs have the same restrictions as LP's, however they include variables that constrained to be integral (discrete). Many authors abbreviate the set of all positive integers $\mathcal{Z}^+$ [22]. Integer programs are considerably more difficult to solve than LP's. Binary integer programs are a special case of IP's, the variables in a BIP are all constrained to exist as either a one or zero. The positive binary set is commonly abbreviated as $\mathcal{B}^+$ [22]. BIP's are easier to solve than general IP's, however they are still harder to solve than LP's. A mixed integer program is an IP that also contains some real variables. MIP's are perhaps the most difficult type of IP to solve, since algorithms that are used to solve IP's are often designed for only integer variables.

Non linear programs are math programs where some or all of the constraints or objective are non linear functions [17]. The majority of NLP's can not be solved efficiently, with the notable exception of convex quadratic functions. A non linear program where some or all of the variables are constrained to be integers is called a non linear mixed integer program. NLMIP's are very difficult optimization problems to solve.

The general form of an LP can be expressed by the following formulation (3.2):

$$\begin{aligned} &\text{Maximize } \boldsymbol{c'x} \\ &\text{Subject To } \boldsymbol{Ax} \leq \boldsymbol{b} \\ &\boldsymbol{x} \geq \mathbf{0} \end{aligned} \tag{3.2}$$

Where x is a vector of real variables, c is the vector of costs, A is the matrix of constraint coefficients and b is the vector of constraint limits.

Many optimization techniques and theory from linear, integer, and non-linear programming may be able to generate input values for a specific output of an SDF model. An

important concept that has been a key part of this thesis is goal programming. The idea of goal programming is to create non-negative deviation variables corresponding to positive and negative deviations of goals. The objective function of a goal-programming problem minimizes the sum of the deviation variables, balancing the trade-offs associated with meeting certain goals [17]. Table 3.2 details generic goal constraints with non-negative deviation variables, criterion functions, and goal values.

Table 3.2: Goal Programming Constraint Description

Constraint Operator	**Constraint Formulation**
Greater Than	(criterion function) + (negative deviation variable) $\geq$ goal value
Less Than	(criterion function) - (positive deviation variable) $\leq$ goal value
Equality	(criterion function) - (positive deviation variable) + (negative deviation variable) = goal value

3.5.1 Piecewise Linear Functions in Mathematical Models

Piecewise linear functions are collections of lines "pieced" together over different ranges. An example is shown in equation (3.3):

$$f(x) = \begin{cases} 5x & \text{if } 0 \leq x < 4 \\ 3x & \text{if } 4 \leq x < 9 \\ 2x & \text{if } 9 \leq x \leq 16 \end{cases} \tag{3.3}$$

Common uses for piecewise linear functions are cost changes with order quantity, such as a volume discount. Certain network flow and logistics models can benefit from the tiered structure of piecewise functions. Sometimes, non-linear functions can be approximated as piecewise linear functions.

Normally using piecewise linear functions in mathematical models requires the use of

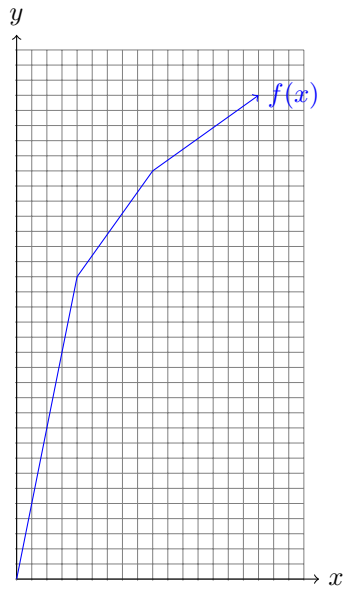

Figure 3.3: Graph of the Piecewise Linear Function (3.3)

binary variables. One binary variable for each piece of the function and constraints restricting the sum of all binary variables to one. The obvious downside to this type of representation is the addition of binary variables! Solving a mixed integer program is more computationally intensive than a pure linear program.

There are two methods commonly applied to formulate piecewise linear functions in mathematical modeling software; the so-called lambda (λ) and delta (δ) forms.

3.5.1.1 The λ Form

The λ form first appeared in a paper by Dantzig [7]. The technique outlined explains how to use binary variables to represent piecewise linear functions. In this case, the binary variables used to switch between sets of adjacent breakpoints within the piecewise function. A continuous variable, λ, exists for each breakpoint in the function. The λ formulation to implement piecewise linear functions into an MIP are shown below in 3.4.

Note, the above assumes that the function $f(z)$ is broken up into points $P_0 ... P_k$ where each P_k has coordinates (a_k, b_k). The δ variables ensure that only adjacent λ variables are

$$
\begin{aligned}
&\text{Maximize} f(z) = \sum_{k=0}^{K} b_k \lambda_k \\
&\text{Subject To:} \\
&\qquad \sum_{k=0}^{K} a_k \lambda_k = z \\
&\qquad \sum_{k=0}^{K} \lambda_k = 1 \\
&\qquad \sum_{k=0}^{K} \delta_k = 1 \\
&\qquad \lambda_k \leq \delta_{k-1} + \delta_k \qquad \forall k \in K \\
&\qquad 0 \leq \lambda \leq 1 \qquad \forall k \in K \\
&\qquad \delta_k \in \mathcal{B}^+ \qquad \forall k \in K
\end{aligned} \tag{3.4}
$$

Figure 3.4: The λ Formulation

selected to be in the basis at any time. Consider a piecewise linear function with 3 breakpoints, perhaps equation (3.3). The expansion of the constraint(3.4) from figure 3.4 yields:

$$
\begin{aligned}
\lambda_0 &\leq \delta_0 \\
\lambda_1 &\leq \delta_0 + \delta_1 \\
\lambda_2 &\leq \delta_1 + \delta_2
\end{aligned} \tag{3.5}
$$

Equation (3.5) shows how the ranges are constrained and the correct λ variables are switched on and off. For example, fixing δ_0 to 1 essentially turns on the first piece of the piecewise linear function, as λ_0 and λ_1 can now be ≥ 0.

Special ordered sets (SOS) are a data structure that can be used to solve piecewise linear optimization problems with higher efficiency [2, 3, 13] . Simply, they add an extra step in the branch and bound algorithm: a node can only be feasible if a certain number of adjacent (typically by index) variables are active. Commonly, two types of SOS are seen: SOS1 and

SOS2. The number following SOS indicates the number of adjacent variables required to be active in the basis. A solver that can use SOS type constraints can prune the branch and bound tree very quickly, and therefore produce optimal solutions much more quickly than the default code.

3.5.1.2 The δ Form

Approximately ten years after [7], the δ formulation for piecewise linear functions was created [2]. The δ name was chosen because the variables are related to differences in the data parameters. The δ form is more efficient than the λ form, because no binary variables are required. The δ formulation to implement piecewise linear functions in LP's is shown in equation (3.5):

$$\text{Minimize} f(z) = \sum_{k=1}^{K} \frac{b_k - b_{k-1}}{a_k - a_{k-1}} \delta_k \tag{3.6}$$

Subject To:

$$\sum_{k=1}^{K} (a_k - a_{k-1}) \delta_k = z \tag{3.7}$$

$$\delta_k \leq a_k - a_{k-1} \qquad \forall k \in K$$

$$\delta_k \geq 0 \qquad \forall k \in K$$

Figure 3.5: The δ Formulation

The objective function equation (3.6) is slope between each set of breakpoints. Each δ_k variable has an upper bound equal to the difference between it's breakpoints (3.7).

Unfortunately, the δ formulation in table 3.5 can not be used to model any piecewise linear function. A piecewise linear function can be approximated and used in an linear program when certain conditions hold for both the objective function and the constraints [2]. Namely, all terms must be convex (concave) when minimizing (maximizing) the objective function. All piecewise linear terms for the constraints must follow:

Table 3.3: Necessary Conditions for δ Formulation

	$\leq$	$\geq$
Left Hand Side	convex	concave
Right Hand Side	concave	convex

If the conditions of table 3.3 are met, linear variables can be used to completely model the piecewise function. The formulation in figure 3.5 assumes $f(z)$ to be convex. If $f(z)$ were concave then figure 3.5 would require a $\max$ objective. When the conditions in table 3.3 are not met, the δ formulation in table 3.5 cannot be used and the λ (3.4) formulation should be selected.

Chapter 4

Methodology

4.1 Overview

The goal of this research is to design a method to predict input values for data flow simulations that yield outputs as close as possible to a user specified goal. The sections below describe both a scope and a methodology for this work.

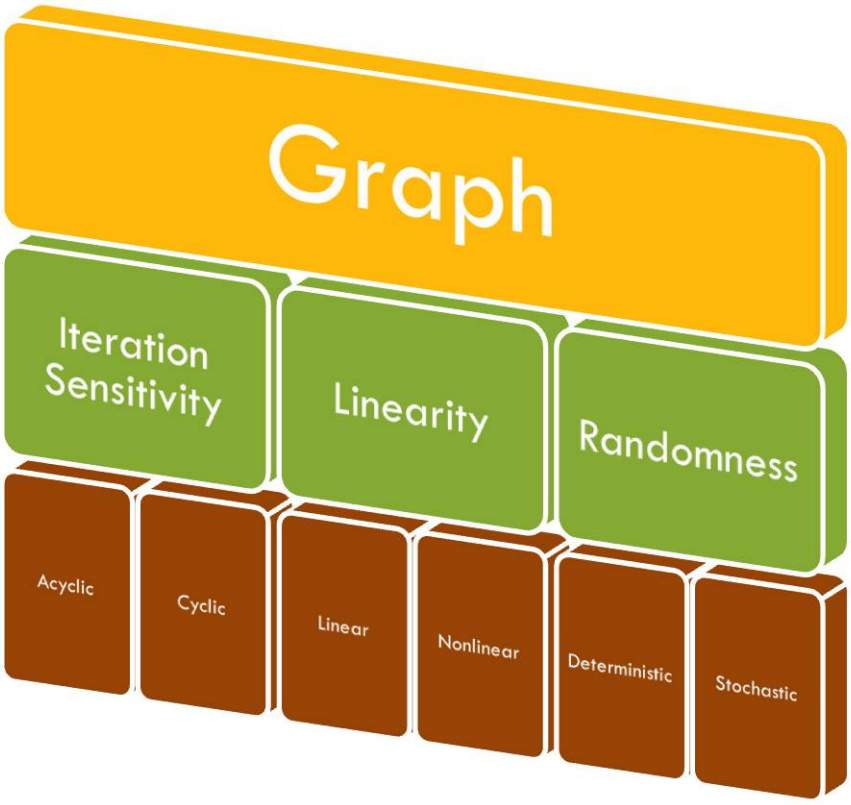

Figure 4.1: Possible Cases of Graph Based Meta-Models Elicited from SDF Models

Figure 4.1 outlines a categorization of cases of the graph based meta-models elicited from an synchronous data flow (SDF) model; these cases are used to frame the scope of this work. In particular, any graph based meta-model can be described by a combination of three parameters: Iterations Sensitivity, Linearity, and Randomness. Iteration sensitivity is a parameter that differentiates between SDF models whose responses are dependent on previous iterations. Linearity refers to the functions inside of each actor. If even one function inside an actor is not linear, then the model is classified nonlinear. Randomness refers to the presence of certain actors in the model. If the model contains one or more actors that produce output values randomly, then model is classified as stochastic. In this work, only deterministic models have been considered resulting in the following test cases: acyclic linear deterministic, cyclic linear deterministic, acyclic nonlinear deterministic, and cyclic nonlinear deterministic.

The following subsections describe each of the three parameters that define all graph meta-model scenarios in detail.

4.1.1 Iteration Sensitivity

Any connected graph can be either acyclic or cyclic. If a graph has one or more cycles, that graph is considered cyclic. All Ptolemy II SDF models considered in this work are transformed into connected graphs. It is necessary to acknowledge cycles in graphs elicited from the SDF model, because a cycle implies that the response functions of some actors will certainly change from one iteration to the next. Many SDF models treat iterations as discrete time periods.

In a Ptolemy II SDF model, one of the most common actors used to create cycles is the Sample Delay. The Sample Delay actor simply passes its input directly out, lagged by one iteration. Another popular actor that is iteration sensitive is the accumulator. As its name implies, accumulator actors accumulate their inputs. So each iteration, the values input to the accumulator are added to the current sum of all previous inputs. Essentially,

the accumulator actor is a self contained cycle.

The sample delay and accumulator actors are the only iteration sensitive actors that have been considered in this work. Additional actors can easily be accommodated in the future, by taking necessary precautions during the experimental sampling and optimization steps.

4.1.2 Linearity

There are many actors included in the standard build of Ptolemy II. Each of these different actors provides certain different functionality. There are actors that convert inputs between various data types, such as double to integer conversions. Additional, logical actors exist that can be used to compare inputs. There are sets of actors for model inputs (sources), and sets specific to outputs (sinks). The point is, there are many different types of actors one can use when creating SDF models.

Some of the most common actors are math actors. Many mathematical functions are represented by specific actors. There are individual actors for taking averages, reporting minimum or maximum values, and rounding. Other math expressions can be included by using the expression actor.

SDF models will be classified as linear or nonlinear depending on the types of functions inside of actors. The majority of actors listed above will produce nonlinear outputs, and as such if an SDF model contained one of those actors that model would be classified nonlinear. In the special case when only linear functions are present in the model, then the model is classified as linear.

The example Ptolemy model shown in figure 4.2 shows a model that is classified as linear. This is because all functions inside of the expression actors (labeled A, B, and C) are linear. The actor labeled Output is a display actor; it has no output in the graph based meta-model. This simple linear Ptolemy model will be used to demonstrate the methodology throughout the rest of this chapter.

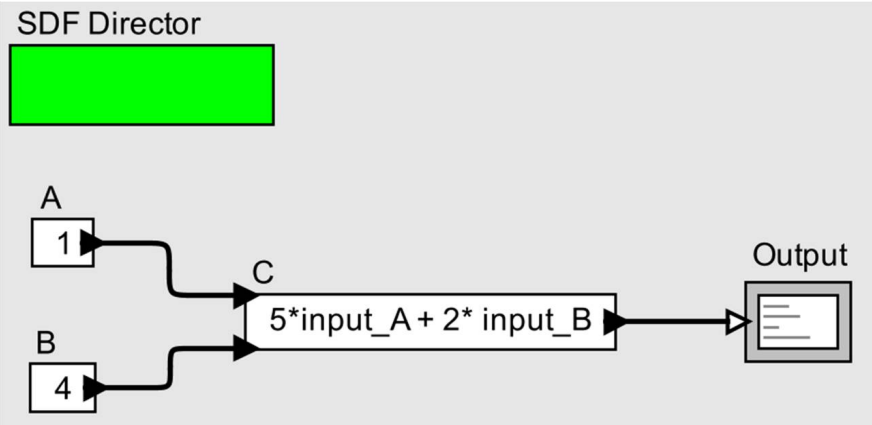

Figure 4.2: Simple Linear Ptolemy Model

4.1.3 Randomness

SDF models that contain actors that produce stochastic outputs have not been considered in this work, however there are suggestions in the future work chapter that outline how one may account for the random variables with minor modifications to the existing mathematical programs.

As mentioned in the linearity subsection, there are a special set of actors that are used to produce random values. There are individual actors corresponding to different probability distributions; there is a Gaussian (normal) actor, a Poisson actor, an Exponential actor, a triangular actor, etc. The presence of one of these types of actors in an SDF model would classify that model as stochastic. Otherwise, the model is classified as deterministic.

4.2 Description of Methodology

Given a deterministic SDF simulation model, the following methodology is developed to find input values that yield an output as close as possible to user specified goals. There are three main categories within the methodology: information abstraction, optimization, and solution evaluation. A flowchart of the overall methodology is shown in figure 4.3.

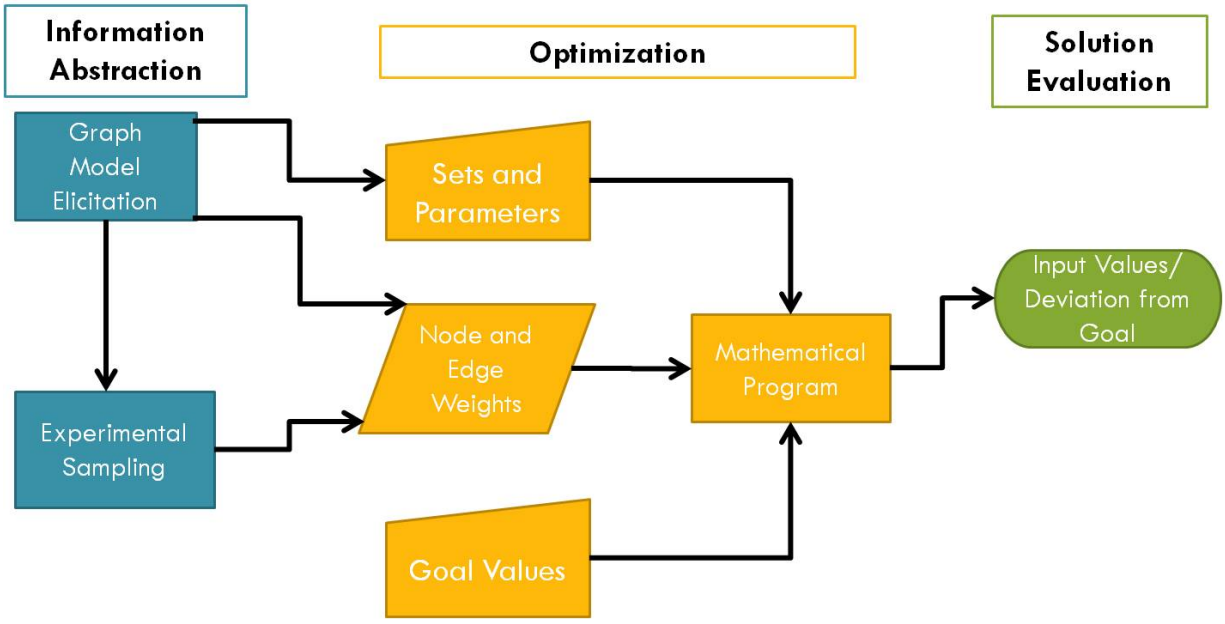

Figure 4.3: Methodology Flowchart

The processes in figure 4.3 follows three components in order: Information abstraction, Optimization, and Solution Evaluation. First, information is abstracted from the SDF model. Information abstraction refers both to methods that reduce SDF models into directed graphs and experimental sampling techniques used to determine the magnitude of an input on an output. Next the gathered information is analyzed and modified for use in an optimization model; these methods are referred to as experimental sampling. Finally, the results from the optimization are evaluated by executing the simulation model with the optimal input values.

The following subsections detail each of the three components of the methodology: Information Abstraction, Optimization, and Solution Evaluation.

4.2.1 Information Abstraction

The first step required in this methodology is the elicitation of the underlying graph structure for the current synchronous data flow (SDF) model. The graph based meta model of

the simple linear Ptolemy model in figure 4.2 is shown in figure 4.4. The graph contains one vertex for each actor in the SDF model, and one edge for each relation.

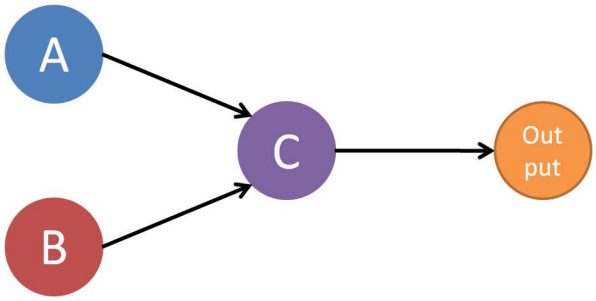

Figure 4.4: Graph Based Meta-Model of Simple Linear Ptolemy Model

Elicitation software has been written for previous work [20], and will be used for this implementation. The Java software to transform a Ptolemy SDF model is a custom adaptation of the Java Universal Network / Graph Framework (JUNG). Actors and relations from the simulation respectively correspond to nodes and edges on the graph. In Ptolemy models, parameters frequently provide input values actors. Parameters typically become source nodes on the elicited graph for this reason.

Once the graph of the SDF model has been elicited, the experimental sampling begins. The graph must be generated first, as we experiment on each node in the elicited graph. Experimental sampling can provide information about a simulation model over a given range. The information collected during experimental sampling is crucial for the optimization section of the methodology.

The sampling experiment used on SDF models in this thesis is similar to the designed experiment from [20]. Tauer ran a 2^k designed experiment on each actor in the model. The experiment is simple, load the the current actor into a new SDF model with only an SDF director and the current actor. Pass the high and low values into the actor, and record the outputs. Then, Tauer computes the corresponding main effects and uses them to determine the relative importance of nodes on the graph.

In this thesis, the sampling experiment is very similar. Individual actors are experimented on, and output values are recorded. The main difference is the number of samples taken, instead of only looking at high and low values a vector of control values are generated within the range of high and low values for each input. The output of the actor is then recorded as the response vector. If the model is non-linear, then the piecewise fitting algorithm, figure algorithm in algorithm 4.1, is used to approximate the response function of the actor.

The procedure for a sampling experiment is:

1. Get the current node from the elicited graph,
2. Find the corresponding actor in the SDF model,
3. Generate control vectors for each input to the current actor,
4. Execute one iteration of the current actor with the current input and record the output response,
5. Repeat step two until all control values have been tested.

Figures 4.5 and 4.6 illustrate the sampling procedure for the ongoing simple linear Ptolemy model example. Note that the current node being experimented on is node C. Figure 4.5 shows that an input range of one through ten is being considered for nodes A and B in the graph based meta-model. The generated control vectors are for nodes A and B and the corresponding response vector for node C is shown in figure 4.6. Since this model is acyclic, the sampling of node C only has to occur once. An assessment of the response information will occur during the optimization step of the methodology.

An important issue when sampling actors one at a time is ensuring range data exists for the current actor. A user is required to provide ranges for the input node on the graph, and initial values for sample delay actors. With this information, an algorithm can be run to determine the actor sampling order.

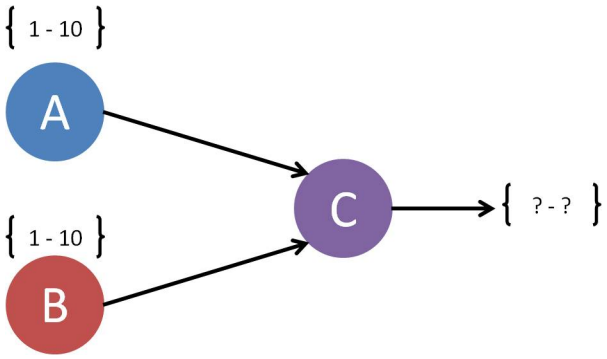

Figure 4.5: Input Ranges for Experimental Sampling of Simple Linear Ptolemy Model

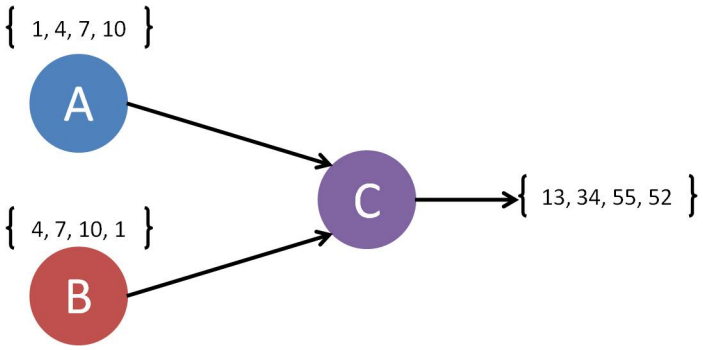

Figure 4.6: Example Experimental Sampling of Simple Linear Ptolemy Model

A feasible ordering can be determined simply by examining the relations in the SDF model, or the edges on the elicited graph. Since sampling ranges are provided for the input actors, any actor that receives tokens from only input actors can be sampled immediately. Then, any actor whose inputs come from the following intermediate actors can be sampled. This process can be repeated until all actors have been ordered.

If the SDF model contains cyclic actors, then the sampling experiment must be run for the number of iterations in interest. The experiment must be run for all iterations to ensure that samples are taken over ranges that will actually occur during normal simulation execution. Without this step, an insufficient amount of data would be collected for the optimization model, and it would most likely produce inaccurate results.

Another important consideration during experimental sampling, especially in nonlinear

models, is that combinations of input values are sampled that are would normally occur when executing the model. When actors have multiple inputs, it becomes even more important to ensure this occurs. If values are sampled that would be impossible in the SDF model, the optimization model will produce inaccurate results.

The only data types that have been considered for actors are integers and doubles. Booleans could be accommodated in the future, however they are not commonly used. Sample delay actors need to be have control vectors generated differently than other actors. During the 0^{th} iteration, the only output a sample delay actor can provide is its' initial value. As such, the sample delay actor should only input to the sample delay should be a repeated vector of the initial value. If the input vector contains values other than the initial value, inaccuracies will occur in the optimization model.

After all nodes on the graph are sampled and node or edge weights are generated, the optimization process can begin.

4.2.2 Optimization

The philosophy behind the optimization models used in this work is simple: a mathematical model that accurately represents the data flow in the SDF model will produce accurate results. The optimization technique used in this work is mathematical programming. Models of many complex systems can be solved efficiently using mathematical programs. However before the optimization models are run two key steps must be completed: an assessment of the magnitude of each node must be generated, and indexing sets and data parameters are required to be generated or input.

4.2.2.1 Magnitude Assessment

At this point, all output response data is collected so an assessment of the magnitude of the output of the each node is conducted. These magnitudes are expressed as either edge weights or mathematical functions of nodes. When the SDF model contains only linear

response functions, edge weights are a logical and convenient representation of the fitted magnitudes. If the linear response includes a constant term (e.g. intercept), that value is represented as a constant data value on the node itself.

Here is a brief example to illustrate the correct edge weights that should be generated for a linear SDF model. In figure 4.7 each expression actor performs some linear operation on all of its input edges. Expression2, for example, takes the sum of 5 times input and -8 times input2. If accurately generated, the edge weights for input and input2 should be 5 and -8, respectively.

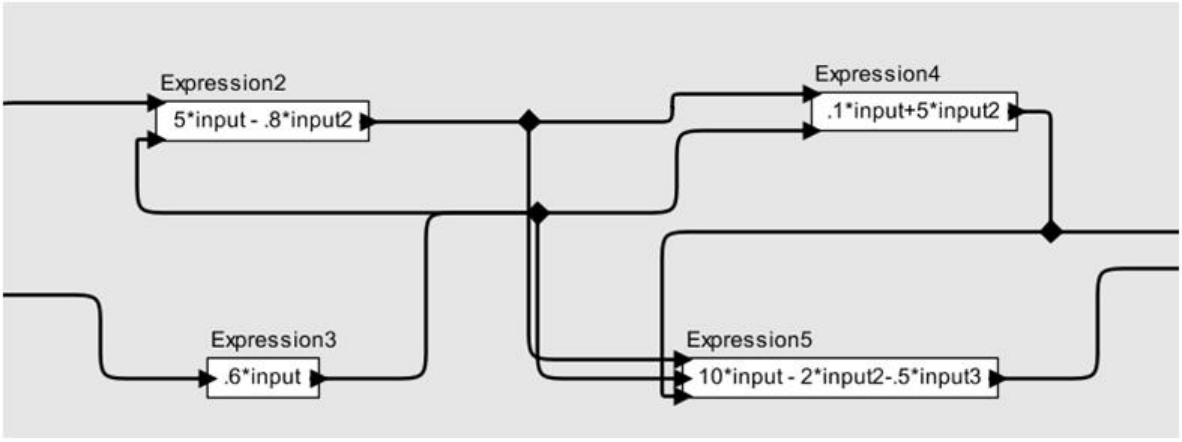

Figure 4.7: Example Ptolemy Model

Continuing the example of the simple linear Ptolemy model in figures 4.2 through 4.6, the correct edge weights should be five and 2. By passing the control and response data shown in figure 4.6 into a linear regression algorithm, the results should be C = 5 $\times$ A + 2 $\times$ B with a $R^2 = 1$. This fitted equation is exactly the same as the function inside expression actor C! The fitted edge weights are shown on the graph based meta-model of the simple linear Ptolemy model in figure 4.8.

Non-linear operations, especially interaction terms, cannot easily be represented as a system of linear edge weights. Due to the difficulties associated with solving the majority of non-linear optimization models, all non-linear functions will be approximated as piecewise

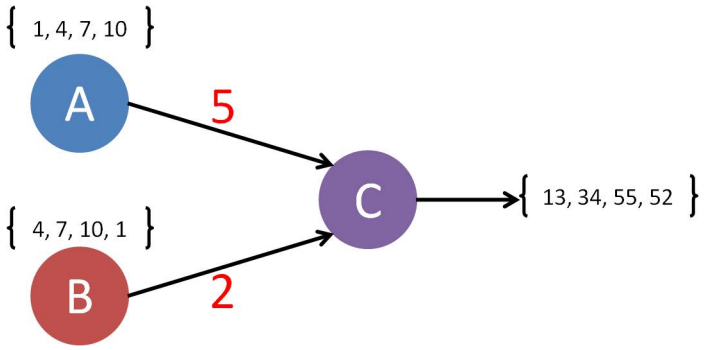

Figure 4.8: Graph Based Meta-Model of Simple Linear Ptolemy Model with Edge Weights

linear functions. The resulting piecewise linear functions are then used in mixed integer programming optimization models. Although MIPs are often difficult to solve, they are (with few exceptions) more efficient to solve than NLPs.

The degree of accuracy required will vary from problem to problem. High accuracy will likely be desired if someone was analyzing a model where changing input parameters in the real system could have life threatening consequences. In practice, obtaining high accuracy can be computationally intensive. As such, the level of expected accuracy should be controllable by a user to allow faster computation for less critical problems. This is accomplished through two different methods: the number of samples taken per node, and the minimum acceptable R^2 value in the piecewise fitting algorithm in algorithm 4.1.

4.2.2.2 Piecewise Linear Fitting Algorithm

Algorithm 4.1 details how piecewise linear approximations are fit to nonlinear outputs from actors. A visual guide to this algorithm is shown in figure 4.9. As the algorithm progresses through each iteration, more line segments are used to fit the observed data.

Simply, the piecewise fitting algorithm 4.1 takes a data set and fits linear regression models until a satisfactory R^2 value is achieved. The number of samples taken during the experiment is restricted to a power of two, to ensure data ranges can always be split evenly. It is possible to modify the piecewise fitting algorithm to avoid this constraint, however the

Algorithm 4.1 Piecewise Linear Function Fitting Pseudo-Code

Require: Number of samples = 2^m
  $i = 0$;
  while $i \leq m$ **do**
    $q = 2^i$
    $currentMax = 1$
    for $p = 1; p \leq q; p++$ **do**
      $currentMin = currentMax$
      $currentMax = p * \frac{2^m}{q}$
      $R^2 = 0$;
      Solve regression model with data subset $currentMin \rightarrow currentMax$
      Update R^2 $\{t$ is the minimum acceptable R^2 value$\}$
      if $R^2 \geq t$ **then**
        Store fitted data range and coefficients
        Do not include the fitted range in any future functions
      end if
    end for
    $i = i + 1$;
  end while

restriction has not been an issue in this work. The while loop determines the number of pieces being fit in the current iteration.

During the 0^{th} iteration, $q = 2^0 = 1$, meaning all data points are being used in the regression fitting. This means one line will be fit to all data points. If the R^2 value of the fitted regression model is greater than or equal to a minimum acceptable R^2 (input by the user), then that fitted range is stored and that data range will not be used in any future fitting. However, if R^2 is less than the minimum acceptable value i is increased by one and the next iteration takes occurs.

In the 1^{st} iteration, $q = 2^1 = 2$, now two lines will be used to fit the data set. The first piece in the data subset is determined using the variables $currentMin$ and $currentMax$; they correspond to the minimum and maximum indices of the sampled data. So with $i = 1$ and $p = 1$ we have:

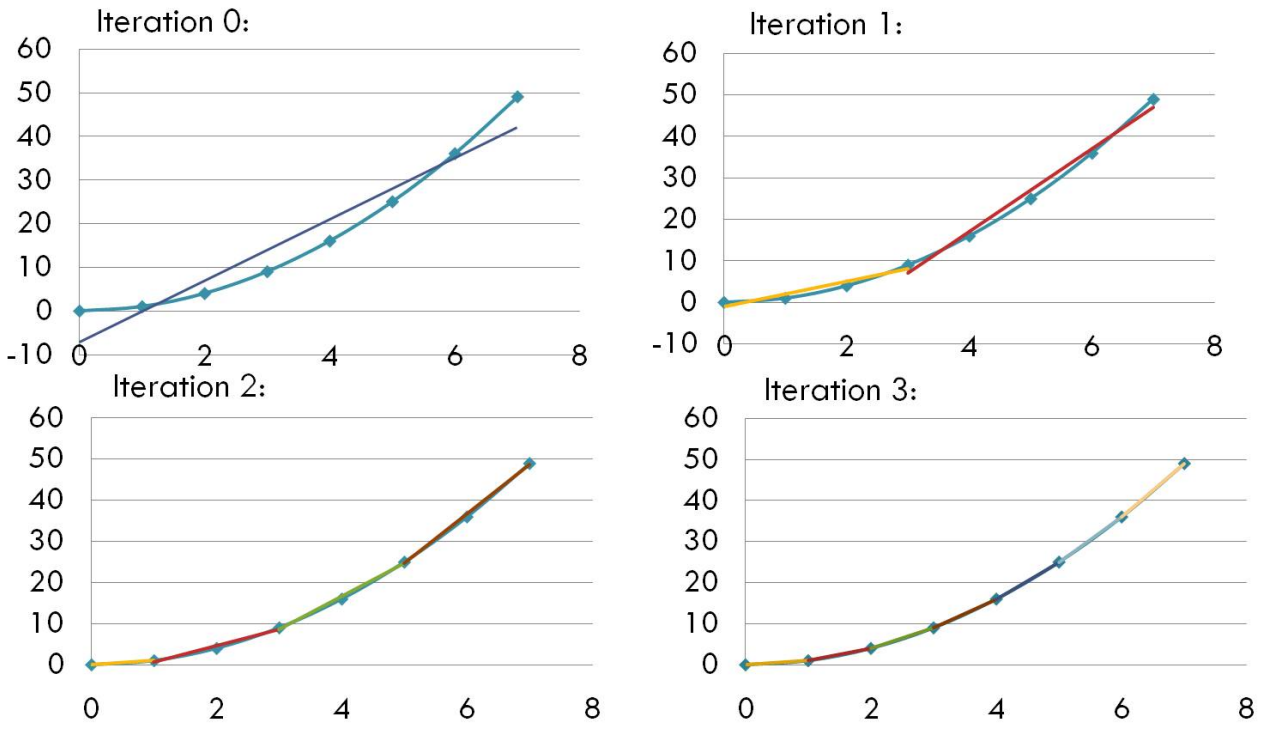

Figure 4.9: Piecewise Linear Function Fitting Example

$$currentMin = 1 \tag{4.1}$$

and

$$\begin{aligned} currentMax &= p * \frac{2^m}{q} \\ &= 1 * \frac{2^m}{2} \\ &= 2^{m-1} \end{aligned} \tag{4.2}$$

$currentMin$ is the first entry in the data set, as shown in equation (4.1). Equation (4.2) shows $currentMax$ equal to the middle index of the data set. One line has now been fit to the first half of the data set. Next we fit the 2^{nd} half of the data by letting $p = 2$. Now:

$$\begin{aligned} currentMin &= currentMax \\ &= 2^{m-1} \end{aligned} \tag{4.3}$$

and

$$\begin{aligned} currentMax &= p * \frac{2^m}{q} \\ &= 2 * \frac{2^m}{2} \\ &= 2^m \end{aligned} \tag{4.4}$$

Equation (4.3) shows $currentMin$ now equal to the previous value of $currentMax$. The previous maximum index becomes the current minimum index to ensure continuous fitting. $currentMax$ has now taken the final index position in the observed data, by equation (4.4). $currentMin$ and $currentMax$ now bound the 2^{nd} half of the data set. Thus, two lines are fit to the data set in order to achieve a higher accuracy than a simple linear approximation.

The piecewise fitting algorithm will continue in splitting ranges in half and fitting lines until $i = m$. With $i = m$, $2^m - 1$ lines will be fit the data set. When this happens, one line is fit between each set of points in the data set (by sampled order). When only using two points to fit a regression model, the R^2 value is guaranteed to be one.

4.2.3 Optimization Overview

Following the edge / node fitting, the other necessary data parameters and indexing sets are generated or input. First, the indexing sets are built based on information from the graph meta-model.

A set is formed that contains all nodes on the graph. Multiple subsets are derived from the main set of all vertices. Two important subsets are the input and goal subsets. The input subset consists of all input nodes on the graph. Another subset is the goal subset, consisting of all goal nodes. The subset of goal nodes will most likely contain the output

nodes; however intermediate nodes may also exist. A user specified goal parameter value is required for each node in the goal subset.

In the linear programming model, one of the primary constraints are based on a principle from linear signal flow graphs. In linear signal flow graphs, the addition rule states that the sum of all inputs to a node multiplied by their respective edge weights equal the output of that node [8]. One of the main constraints (4.8) in the linear programming model in figure 4.10 is based on structure of linear signal flow graphs.

Looking at the addition rule in another way, the scalar product of the inputs of a node and their respective magnitudes equal the output of that node. One can accommodate non-linear terms and functions by generalizing the addition rule; the output of a node is equal to some function of that nodes inputs. Constraints represent this structure for all nodes on the graph except input nodes. There are two reasons why input nodes need unique constraints. By definition, an input node has no inputs; the standard constraints would force all input nodes to zero. Additionally input nodes are assumed controllable in the real life system being modeled. Input values are simply bounded by lower and upper bounds.

The objective function minimizes the sum deviations from the user specified goals. Objectives of this type are implemented with a goal programming approach. By minimizing the sum of deviations about the specified goals, the model balances the trade-offs associated with getting as close as possible to all user specified goals. An added bonus to this type of objective function is that the optimization models should never be infeasible! If a goal can not be achieved, then the objective value at optimality will be larger than zero.

A linear program and mixed integer program are used to find the optimal input combinations. These math program formulations are detailed in this chapter.

4.2.3.1 Sets, Parameters and Variables

All sets, parameters, and variables used in the mathematical programs below are detailed in this subsection. The sets are detailed in table 4.1. Constant data values, also known as

parameters, are listed in table 4.2. All variables that appear in the math models are shown in table 4.3.

Table 4.1: Sets

V	Set of all vertices on the graph.
I	Subset of V, those vertices which are input actors.
G	Subset of V, those vertices which are goal actors.
C	Subset of V, those actors which allow cycles.
CSD	Subset of C, all sample delay actors.
$CACC$	Subset of C, all accumulator actors.
T	Set of all time periods,$\mid$ T $\mid$ should equal the number of iterations.
M_v	Set of indexes related to the number of breakpoints fit for each piecewise linear function.

Table 4.2: Parameters

β	Fitted weight for an edge on the graph.
ω	Target value for each goal vertex.
l	Lower bound for the input variables.
u	Upper bound for the input variables.
k	Initial value of a sample delay actor in the 0th iteration.
a	The “x“ coordinate of fitted piecewise functions.
b	The “y“ coordinate of fitted piecewise functions.

Table 4.3: Variables

x_{vt}	The value of each vertex in the graph.
θ_g	Value corresponding to the percent achieved of each goal, ideally its value is 1.
Δ_g^+	Variable corresponding to goal nodes with actual values greater than their desired value, ideally this variable should be zero.
Δ_g^-	Variable corresponding to goal nodes with actual values less than their desired value, ideally this variable should be zero.
λ_{vtm}	Linear variable used to take the weighted average of two fitted breakpoints in a piecewise function.
δ_{vtm}	Binary variable used to switch between ranges within piecewise functions.

4.2.3.2 Linear Program

When an SDF model contains actors that only have linear functions, a linear program can be used to accurately and efficiently optimize it. Since all output functions within nodes are linear, a weight can be applied to each input edge of each actor on the elicited graph. Any constant (e.g. intercept) term in the linear function can be represented by a data parameter.

The linear programming formulation in figure 4.10 that can optimize the linear deterministic cases of table 4.1 is detailed below. Note, if the SDF's elicited graph is acyclic then the set T should only contain one element. Elicited graphs that are cyclic should have an element in T for each iteration experiments have been conducted on. One can think of the LP as treating acyclic graphs as a special case of cyclic graphs.

$$\text{Minimize } z = \sum_{g \in G} \Delta_g^+ + \Delta_g^- \tag{4.5}$$

Subject To:

$$\theta_g = \frac{x_{gt}}{\omega_g} \qquad \forall g \in G, t = \max(T) \tag{4.6}$$

$$\theta_g - \Delta_g^+ + \Delta_g^- = 1 \qquad \forall g \in G \tag{4.7}$$

$$\beta_{0jt} + \sum_{i \in V} \beta_{ijt} x_{it} = x_{jt} \qquad \forall j \in V \setminus (I \text{ and } C), t \in T \tag{4.8}$$

$$\beta_{0jt} + \sum_{i \in V} \beta_{ijt} x_{it} = x_{jt+1} \qquad \forall j \in CSD, \min(T) < t < \max(T) \tag{4.9}$$

$$k_j = x_{jt} \qquad \forall j \in CSD, t = \min(T) \tag{4.10}$$

$$x_{jt-1} + \beta_{0jt} + \sum_{i \in V} \beta_{ijt} x_{it} = x_{jt} \qquad \forall j \in CACC, t \in T | t > \min(T) \tag{4.11}$$

$$\beta_{0jt} + \sum_{i \in V} \beta_{ijt} x_{it} = x_{jt} \qquad \forall j \in CACC, t = \min(T) \tag{4.12}$$

$$x_{it} = x_{it+1} \qquad \forall i \in I, t \in T | t < \max(T) \tag{4.13}$$

$$l_i \leq x_{it} \leq u_i \qquad \forall i \in I, t \in T \tag{4.14}$$

$$\Delta_g^+, \Delta_g^- \geq 0 \qquad \forall g \in G \tag{4.15}$$

Figure 4.10: Linear SDF Optimization Model

The objective function (4.5) minimizes the sum of positive and negative deviation variables. This allows the model to balance the trade-offs associated with achieving all goals. Constraint (4.6) sets the percent of goal achieved, θ_g, for each goal g in the set G. All deviation variables are bound by constraint (4.7). If every θ_g equals one, then the deviation variables Δ_g^+ and Δ_g^- will have values of zero. If θ_g is larger than one, then Δ_g^+ will have a value greater than zero to meet this constraint. Similarly, Δ_g^- will need to be non-zero if θ_g is less than one.

Constraint (4.8) is the implementation of the addition rule from linear signal flow graphs.

The scalar product of weights β_{ijt} and node values x_{it} for each node i going into node j equals the value of node j, for all vertices excluding those in the I subset. The β_{0jt} term represents the fitted intercept of node j.

Sample delay actors are modeled in (4.9), where the scalar product of all inputs into the sample delay are equal to the output of that actor in the next time period (x_{jt+1}). The initial value of the sample delay actor is fixed to k (4.10).

Accumulator actors are constrained in (4.12) and (4.11). Constraint (4.12) requires the output of an accumulator node to be equal to the sum of its inputs, plus the value of that node in the previous time period (x_{jt-1}). The variable x_{jt-1} is omitted from (4.11), since that term will not exist in the 0^{th} time period.

Finally, input node values x_{it} are bounded by l_i and u_i in constraint (4.14). Since input values will not change once the SDF model executes, constraint (4.13) is necessary. The Δ variables are restricted to be non-negative in constraint (4.15).

Applying the LP formulation in figure 4.10 to the simple linear Ptolemy example in figures 4.2 through 4.8, the resulting optimization model is shown in figure 4.11. Here the goal value is set to 25.0 for node C.

Minimize $z = 1.0 * \Delta_0^+ + 1.0 * \Delta_0^-$

Subject To:

$$2.0 * B_0 + 5.0 * A_0 = 1.0 * C_0$$

$$1.0 * \theta_0 = \frac{C_0}{25.0}$$

$$1.0 * \theta_0 - 1.0 * \Delta_0^+ + 1.0 * \Delta_0^- = 1$$

$$1.0 \leq A_0, B_0 \leq 10.0$$

Figure 4.11: Linear Optimization Formulation for Simple Linear Ptolemy Model

4.2.3.3 Mixed Integer Program

The mixed integer programming model in figure 4.12 is used to optimize the non-linear deterministic cases in table 4.1. Just as in the linear programming formulation, if the model

is acyclic then the set T should only have one element. Special constraints for accumulators are not necessary, as their variation over time is captured with the sampling experiment. However, using constraints (4.12) and (4.11) would reduce the number of binary variables in the mixed integer program.

Minimize $z = \sum_{g \in G} \Delta_g^+ + \Delta_g^-$

Subject To:

$$\theta_g = \frac{x_{gt}}{\omega_g} \qquad \forall g \in G, t = \max(T)$$

$$\theta_g - \Delta_g^+ + \Delta_g^- = 1 \qquad \forall g \in G$$

$$\sum_{m \in M_i} a_{itm} \lambda_{jtm} = x_{it} \qquad \forall j \in V \setminus I, i \in V | i \to j, t \in T \tag{4.16}$$

$$\sum_{m \in M_j} b_{jtm} \lambda_{jtm} = x_{jt} \qquad \forall j \in V \setminus (I \text{ and } CSD), t \in T \tag{4.17}$$

$$\sum_{m \in M_j} b_{jtm} \lambda_{jtm} = x_{jt+1} \qquad \forall j \in CSD, t \in T \tag{4.18}$$

$$\sum_{m \in M_i} \lambda_{itm} = 1 \qquad \forall i \in V, t \in T \tag{4.19}$$

$$\sum_{m \in M_i} \delta_{itm} = 1 \qquad \forall i \in V, t \in T \tag{4.20}$$

$$\lambda_{itm} \leq \delta_{itm-1} + \delta_{itm} \qquad \forall i \in V, m \in M_i, t \in T \tag{4.21}$$

$$k_j = x_{jt} \qquad \forall j \in CSD, t = \min(T)$$

$$x_{it} = x_{it+1} \qquad \forall i \in I, t \in T | t < \max(T)$$

$$l_i \leq x_{it} \leq u_i \qquad \forall i \in I, t \in T$$

$$\Delta_g^+, \Delta_g^- \geq 0 \qquad \forall g \in G$$

$$0 \leq \lambda_{itm} \leq 1 \qquad \forall i \in V, m \in M_i, t \in T \tag{4.22}$$

$$\delta_{itm} \in \mathbf{B}^+ \qquad \forall i \in V, m \in M_i, t \in T \tag{4.23}$$

Figure 4.12: Non-Linear SDF Optimization Model

The MIP in figure 4.12 is very similar to the LP in figure 4.10; the main differences are the addition of constraints that are needed to model piecewise linear functions. The MIP implements the λ form of piecewise linear functions, since the outputs of actors can not be guaranteed to be convex. The constraints derived from linear signal flow graphs, (4.8) and (4.9), have been replaced by constraints (4.16), (4.17), and (4.18).

Essentially, constraints (4.16), (4.17) are a split up version of (4.8), where constraint (4.16) restricts the inputs of a node and constraint (4.17) limits the outputs. These two constraints ensure that λ_{jtm} values are equal for both the input and output of each node on the graph. This relationship helps ensure accuracy in the model, since the fitted ranges directly correspond to the control and response data. Allowing the λ_{jtm} variables to take different values would result in combinations of a parameters that were not sampled together being selected, thus producing inaccurate results. Constraint (4.18) is the output constraint for sample delay actors; output values are lagged by one time iteration (x_{jt+1}).

Constraints (4.19), (4.20), (4.21), (4.22), and (4.23) are all taken directly from the λ formulation outlined in the literature review (see figure 4.1).

4.2.4 Solution Evaluation

Following the execution of the optimization solver, the resulting optimal solution is provided to the user in a meaningful way. Simply dumping the output from an optimization solver is unacceptable! Variables from the optimization should be named such that they are easily identifiable. If possible, the names should be the same as the Ptolemy actors the elicited graph structure represents.

In this work, a text file is generated after the solver is executed. This text file contains the entire math program, along with the value of every variable at optimality. All the variables names are taken directly from the elicited graph meta-model. Having results displayed in this manor all the user to quickly find the optimal input values that should yield their desired goal. As listed in the problem statement, there are three metrics primarily used to assess quality in this work: run time, precision, and accuracy. The output text file contains the run time information in addition to the math model and optimal variable values.

To test quality of the method, one simply has to run the simulation model with the reported optimal input values. If the output of the simulation model is the same as or very close to the predicted output from the optimization model, one can say the optimization was

accurate. If the input values reported by the optimization solver produce an output close to the user specified goals, then one can say the optimization was precise. However it is important to note that the precision measurement only makes sense when the user chooses goal values that are obtainable within the provided input ranges. The most desirable results are highly accurate, highly precise, with a low run time.

```
The expanded model:
IloModel  {
IloMinimize  : 1.0*PositiveDeviation[0] + 1.0*NegativeDeviation[0]
IloRange  : 0.0 <= 2.0*.example.B.output[0] + 5.0*.example.A.output[0]
- 1.0*.example.C.output[0] <= 0.0
IloRange  : 0.0 <= -0.04*.example.C.output[0] + 1.0*theta[0] <= 0.0
IloRange  : 1.0 <= -1.0*PositiveDeviation[0] + 1.0*NegativeDeviation[0]
+ 1.0*theta[0] <= 1.0

}

Objective Value: 0.000

The solve time = 0

All theta Values:
theta[0] = 1.000000
All Positive Deviation Values:
posDelta[0] = 0.000000
All Negative Deviation Values:
negDelta[0] = 0.000000
All x Values:
.example.A.output[0] = 1.000000
.example.B.output[0] = 10.000000
.example.C.output[0] = 25.000000

Total time = 1984
Solve time = 0
Model prep time = 125
Experiment time = 1859
```

Figure 4.13: Example Solver Output for Simple Linear Ptolemy Model

A demonstration of the solution evaluation method begins with figure 4.13, a sample output from the Java implementation of the methodology. CPLEX 12.1 was used to solve the simple linear Ptolemy model with a goal value of 25.0 for expression C. The optimal solution reported by the solver is A = 1.0 and B = 10.0 This is certainly feasible, as the

input ranges were 1.0 through 10.0 for both nodes A and B. What about the accuracy and precision, and solve time?

Executing the Ptolemy model with the optimal input value yields and output of 25.0! These results can be seen in figure 4.14. The name of the metrics used to quantify accuracy and precision are respectively: math program vs simulation deviation and math program vs goal deviation. These metrics will appear again in chapter 5 and Appendix A. The math program vs simulation deviation (accuracy) is given by:

$$\begin{aligned}\text{MP vs Sim Deviation} &= |1.0 - \frac{\text{Simulation Output}}{\text{Math Program Output}}| \\ &= |1.0 - \frac{25.0}{25.0}| \\ &= |1.0 - 1.0| \\ &= 0.0\end{aligned}$$

The math program vs goal deviation (precision) is given by:

$$\begin{aligned}\text{MP vs Goal Deviation} &= |(\text{Math Program Output}) - (\text{Simulation Output})| \\ &= |25.0 - 25.0| \\ &= 0.0\end{aligned}$$

Thus, it can be said that the simple linear Ptolemy model was solved accurately, precisely, and quickly. The accuracy and precision is high, because there is no deviation seen between the optimization model and the actual SDF model. The solve times are shown at the end of figure 4.13; a total run time of under 2000 ms is low, and desirable! This methodology would have high practical significance if large complex SDF models could be solved in a

similar amount of time.

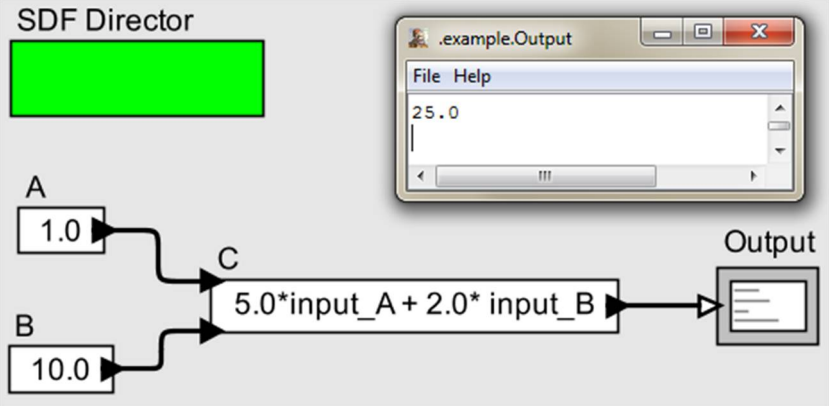

Figure 4.14: Simple Linear Ptolemy Model Execution with Optimal Input Values

4.3 Discussion of Methodology

The three categories of the methodology outlined in this chapter are information abstraction, optimization, and solution evaluation. Here is a brief summary of all steps in the methodology:

1. Load an SDF model into memory,
2. Elicit graph based meta-model from the current SDF,
3. Perform a sampling experiment for each node on the graph (one at a time),
4. Assess the magnitude of the input(s) on the output of each node on the graph (one at a time) by fitting functions (either linear or piecewise linear),
5. Read in user specified goals,
6. Generate indexing sets and constant data parameters using the graph based meta-model, fitted functions, and user specified goals,
7. Pass all data into the optimization model,

8. Solve the optimization model and write out a text file with run time information and optimal values for all variables,

9. Execute the SDF model with the optimal input values as reported by the optimization model,

10. Compare the actual results of the SDF model with the expected results from the optimization model.

The structure of the methodology should allow flexibility for many different scenarios. Total run time should be significantly effected by the number of samples taken during the experimental sampling step, and the minimum acceptable R^2 value. Setting an R^2 value less than one should allow the algorithm to complete in fewer than m iterations (where the number of samples taken = 2^m).

One important aspect of the information abstraction step that has not been discussed in depth is the reason for experimenting on actors one at a time. Most simulation optimization methods take an approach similar to the one in this work; that is sampling parts of the model with no a priori knowledge of the type of function inside. There are many reasons to do this, one of the most prevalent being that the majority of simulation software allow the user to execute their won custom code. If a method was developed that could not support this, it would most likely be limited in functionality.

The optimization models, in figures 4.10 and 4.12, should solve efficiently relative to size of the SDF model. If the SDF is linear, then one should have minimal concerns about the size of the model as model LP solvers can deal with hundreds of millions of variables. In the nonlinear cases, the modeler should be more concerned as mixed integer program is typically computationally intensive. At least the types of models produced in this methodology are at worst mixed binary integer programs, which are generally less computationally intensive than a mixed integer program or a nonlinear program.

An added aspect of this methodology is that it should be applicable to any system that has a graph structure in which the output of a node is some function of its inputs. One of the intriguing benefits of the method is that as long as the output functions are continuous, the piecewise linear fitting algorithm, in algorithm 4.1, will fit a function usable in the mixed integer programming model (figure 4.12). The large variety of systems this method may be applicable for is an attractive feature.

Chapter 5

Experimental Performance Evaluation

This chapter describes the example SDF models used to test the information abstraction and optimization methods outlined in the methodology.

5.1 Experimental Models

Test SDF models were created In order to verify the effectiveness of the methodology. Four models were built, one for each possible graph characteristic combination. The name of each model and size of each elicited graph is listed in table 5.1.

5.1.1 Acyclic Linear Deterministic Model

The acyclic linear deterministic SDF model is shown in figures 5.1 and 5.2. The Ptolemy model is shown in figure 5.1 and the elicited graph is shown in figure 5.2. This model consists of many expression actors that multiply their inputs by linear terms. All mathematical operations in this model are linear, thus the linear optimization formulation (figure 4.10) can be used to efficiently find the input values that yield the optimal goal.

This model has a tree structure; with many input nodes and one output node. The large

Table 5.1: List of Example Models

Name of Model	Number of Vertices	Number of Edges
Acyclic Linear Deterministic	54	53
Cyclic Linear Deterministic	12	17
Acyclic Nonlinear Deterministic	13	12
Cyclic Nonlinear Deterministic	13	12

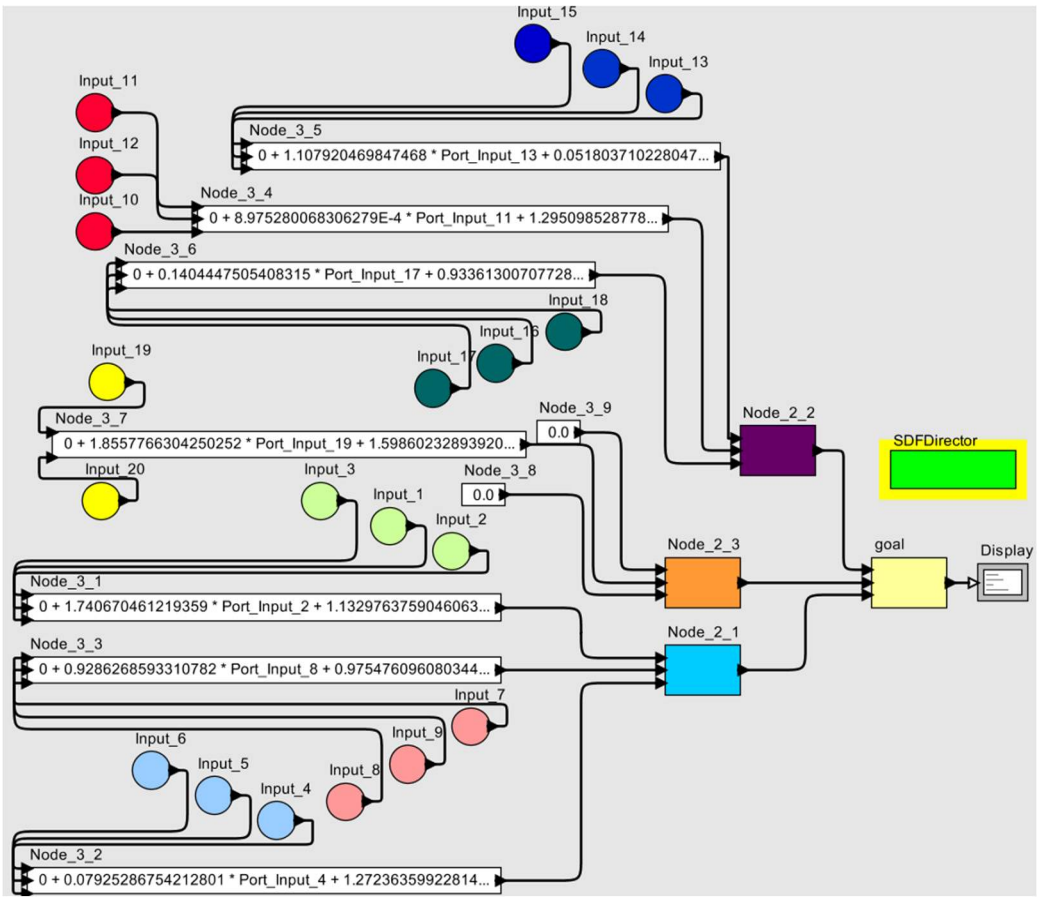

Figure 5.1: Acyclic Linear Deterministic SDF Model: Ptolemy II

number of inputs and small number of outputs provide many feasible solutions to obtain a user specified goal.

The node labeled goal is the user specified goal node. Technically the goal node is an intermediate node. However the output node will always have the same value as the goal node, since it is elicited from a display actor. The display actor shows all input tokens in a small window on screen to user, when executing the SDF simulation. A display actors output can be seen in figure 4.14, the results from the simple linear Ptolemy model example in chapter 4.

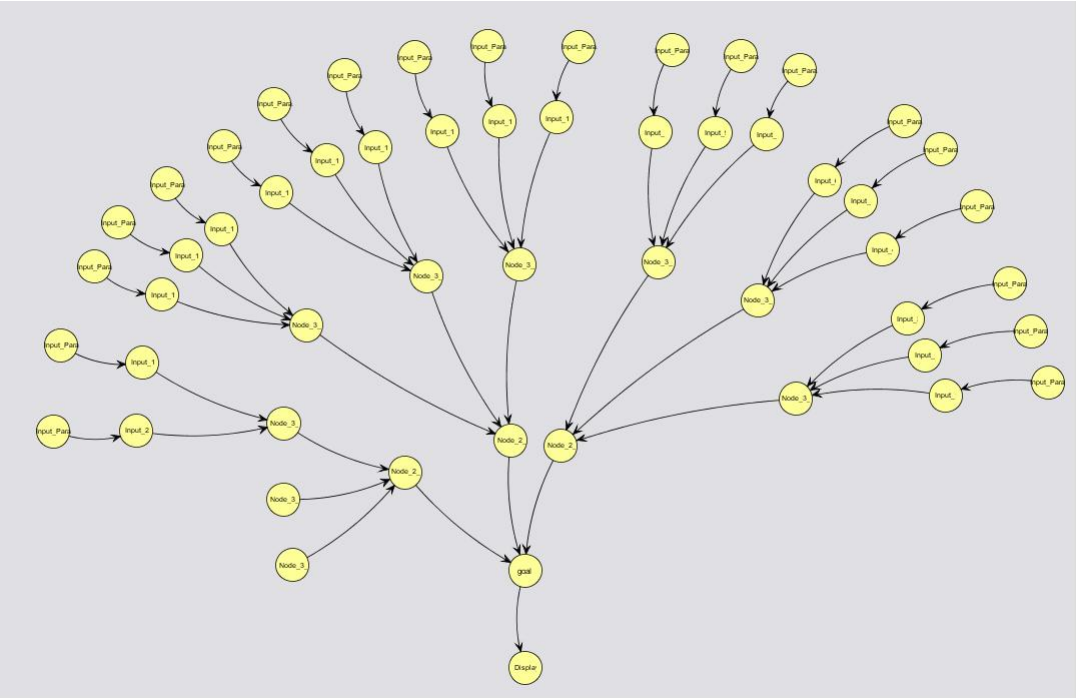

Figure 5.2: Acyclic Linear Deterministic SDF Model: Graph Structure

5.1.2 Cyclic Linear Deterministic Model

The second linear test case is the cyclic linear deterministic model. The Ptolemy SDF model is shown in figure 5.3 and the elicited graph structure is shown in figure 5.4. This model uses a sample delay actor to create a cycle. All functions within the other expression actors are linear, so the LP (figure 4.10) can be used to solve this model.

This model has three input nodes and one output node. Just as in the acyclic linear deterministic case, the output node is elicited from a display actor. Here, many expression actors perform linear functions on their inputs. The expression actors all effect one and other; input values are magnified rapidly due to the cycle.

5.1.3 Acyclic Nonlinear Deterministic Model

Figures 5.5 and 5.6 show the acyclic nonlinear deterministic Ptolemy model and graph structure. Many nonlinear functions are inside the actors in this model. In particular, the

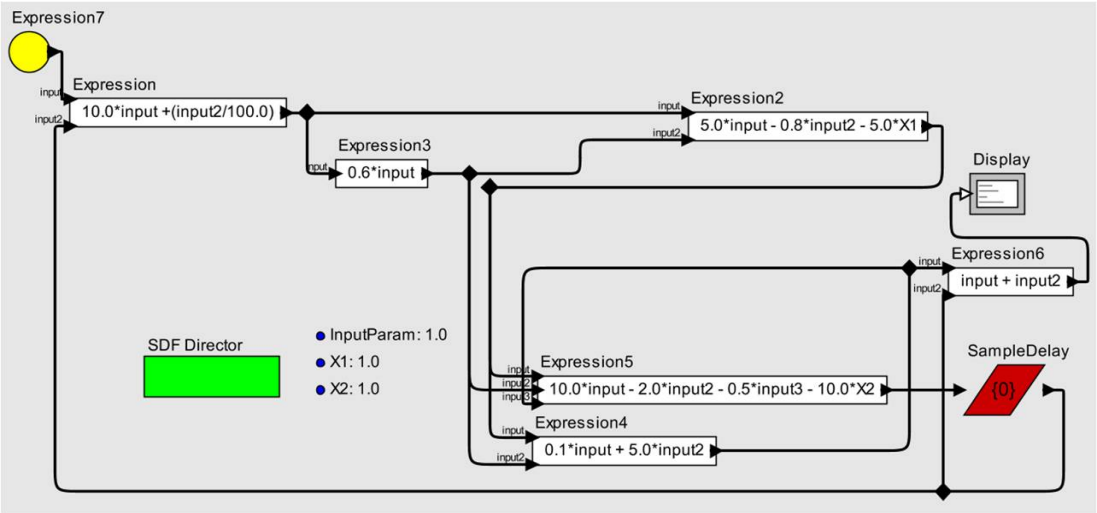

Figure 5.3: Cyclic Linear Deterministic: Ptolemy II Model

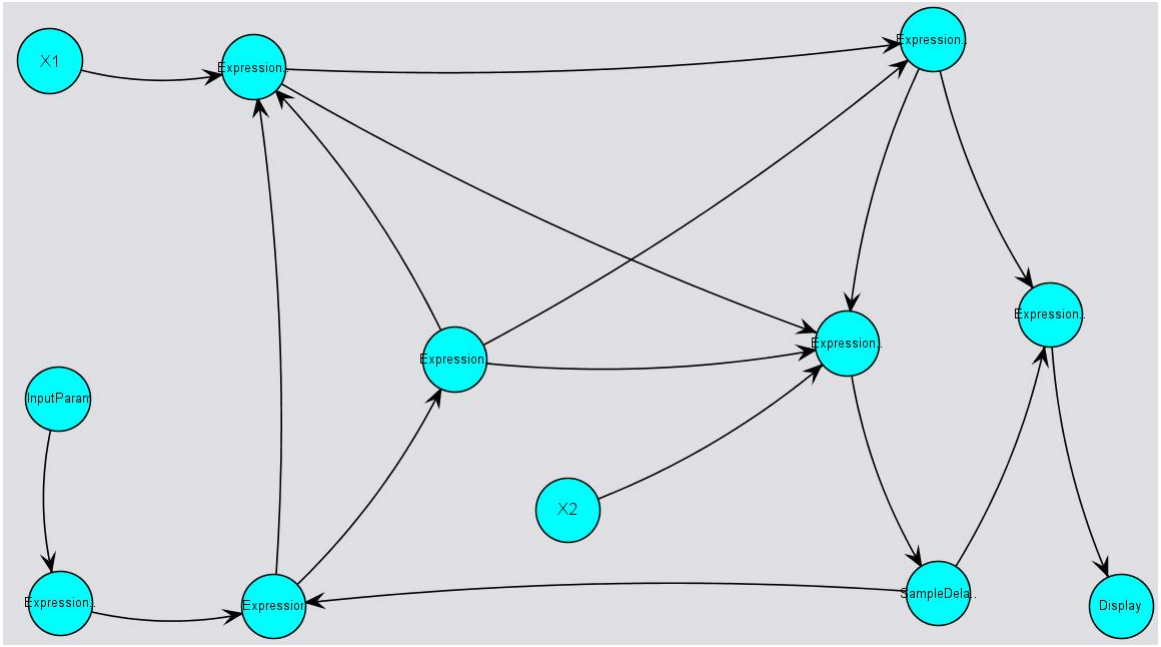

Figure 5.4: Cyclic Linear Deterministic: Graph Structure

multiply divide actor produces a difficult function to approximate: $\frac{\cos(x_3)}{x_4^5}$.

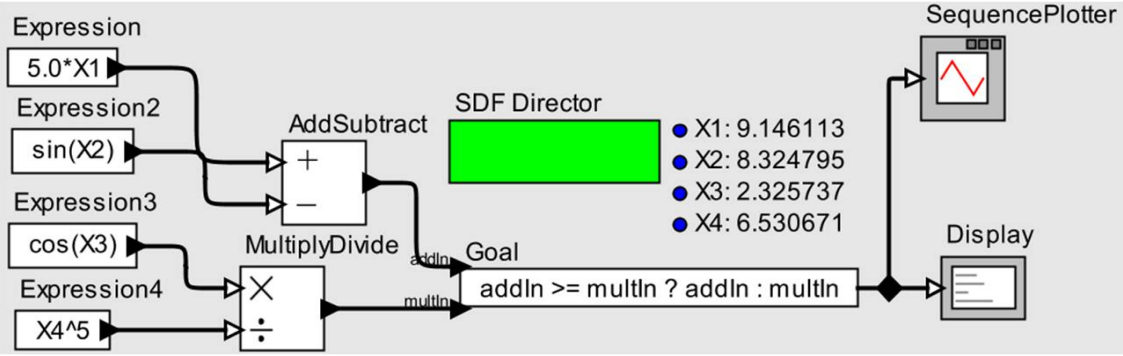

Figure 5.5: Acyclic Nonlinear Deterministic: Ptolemy II Model

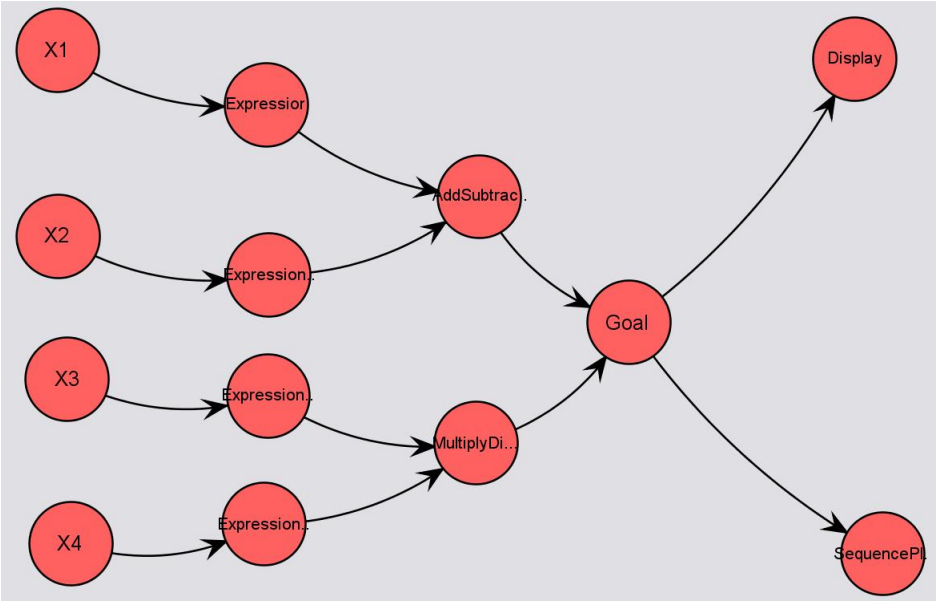

Figure 5.6: Acyclic Nonlinear Deterministic: Graph Structure

This test model has four input nodes and two output nodes. The goal node is once again labeled goal, and it elicited from an expression actor that contains a ternary expression. The ternary operation selects the largest input, from the output of the add subtract actor and multiply divide actor, and then outputs that value to the sequence plotter and display actors.

5.1.4 Cyclic Nonlinear Deterministic Model

The final example model is the cyclic nonlinear deterministic model, shown in figures 5.7 and 5.8. This model is simply a modified version of the acyclic nonlinear deterministic example. The ternary expression has been replaced with an accumulator actor. Note, that the cycle is not shown in figure 5.8. The current version of graph elicitation software does not properly represent accumulator actors as creating a cycle, however the mixed integer optimization model (figure 4.12); does correctly account for the cycle.

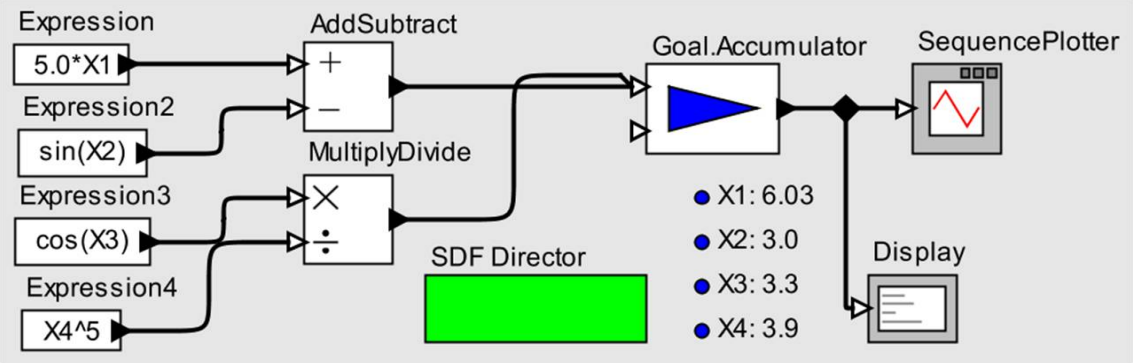

Figure 5.7: Cyclic Nonlinear Deterministic: Ptolemy II Model

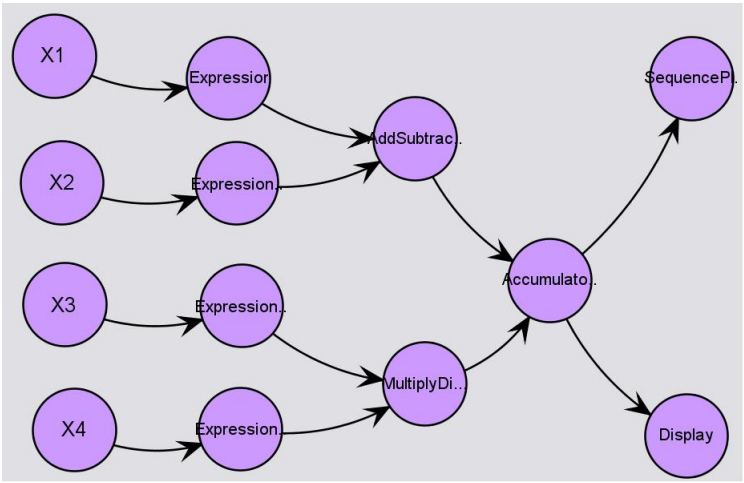

Figure 5.8: Cyclic Nonlinear Deterministic: Graph II Structure

5.2 Experimental Setup

To test the performance of the models, ten runs of each model were conducted with different goal values each run. Eight samples were taken for each of the linear models. In theory only two samples should be required to fit a line to a linear function, however the R statistical software provided inconsistent results when only two points were used. Much more reliable fits were observed when eight samples were taken, hence their use. Since the cyclic models are sensitive to iterations, the experiment was set to run for 25 iterations. The outputs of acyclic models will not change regardless of the number of iterations, so they were only experimented on for one iteration.

A variety of data is shown for each model; the math program vs simulation deviation (MP vs Sim Deviation), math program vs goal deviation (MP vs Goal Deviation), time required for sampling and functions fitting (Experiment Time) and the time required to solve the math program (Solve Time). MP vs Sim deviation describes the accuracy of the optimization method, while MP vs Goal Deviation describes the precision of the method. In the best cases these values will be zero, implying that the optimization correctly predicted the output of the simulation and the output was very close (or exactly equal) to the user specified goal. Experiment and solve times show how long the method took to run, the hope is that run times remain low while accuracy and precision are high. Thus, the best attainable results are a combination that minimize the values of MP vs Sim Deviation, MP vs Goal Deviation, Experiment Time, and Solve Time.

All experiments were completed on a computer with a 2.66 GHz Intel core 2 duo processor and 4 GB of ram. The machine runs a 32 bit version of Windows XP, which limits the amount of ram available to only 3.37 GB. All sampling experiments and function fitting algorithms were written in or called from Java. The Java JDK build 1.6.0_20 was used for all experiments, and all optimization models were solved using CPLEX 12.1 via concert for Java. Concert is a CPLEX API that provides methods to build optimization models.

The optimization models were solved at CPLEX's default settings.

5.3 Results

A complete list of data from all experiments run is listed in the Data Table appendix (Appendix A). The tables in this section are collections of the averages of the results from all experiments. All four of the performance evaluations are the average of 10 runs; each run with varied goal values. An example of the detailed results shown in Appendix A can be seen in table 5.2. Note, the acyclic linear deterministic entry in table 5.3 contains data from the **Averages** row of table 5.2. Additionally, note that table 5.2 is identical to table A.2.

Table 5.2: Detailed Acyclic Linear Deterministic Results

Goal Value	**Math Program Output**	**Simulation Output**	**MP vs Sim Deviation**	**MP vs Goal Deviation**	**Total Time (ms)**	**Solve Time (ms)**	**Model Prep Time (ms)**	**Experiment Time (ms)**
500.0	500.0	500.0	0.000	0.0	2421.0	16.0	109.0	2296.0
550.0	550.0	550.0	0.000	0.0	2374.0	0.0	109.0	2265.0
600.0	600.0	600.0	0.000	0.0	2407.0	0.0	110.0	2297.0
650.0	650.0	650.0	0.000	0.0	2406.0	0.0	109.0	2297.0
700.0	700.0	700.0	0.000	0.0	2391.0	0.0	110.0	2281.0
750.0	750.0	750.0	0.000	0.0	2390.0	0.0	109.0	2281.0
800.0	800.0	800.0	0.000	0.0	2390.0	0.0	110.0	2280.0
850.0	850.0	850.0	0.000	0.0	2376.0	0.0	110.0	2266.0
900.0	900.0	900.0	0.000	0.0	2422.0	0.0	109.0	2313.0
950.0	950.0	950.0	0.000	0.0	2407.0	0.0	110.0	2297.0
Averages:			**0.000**	**0.0**	**2398.4**	**1.6**	**109.5**	**2287.3**

Table 5.3: Results From Linear Models

Name of Model	MP vs Sim Deviation	MP vs Goal Deviation	Experiment Time (ms)	Solve Time (ms)
Acyclic Linear Deterministic	0.0	0.0	2287.3	1.6
Cyclic Linear Deterministic	0.0	0.0	4662.0	4.8

Table 5.4: Acyclic Nonlinear Deterministic Results

	Number of Samples	MP vs Sim Deviation	MP vs Goal Deviation	Experiment Time (ms)	Solve Time (ms)
$R^2 = 1$	4	0.014	0.2	2498.1	6.3
	8	0.003	0.2	2659.6	10.9
	16	0.001	0.2	2914.4	31.2
	32	0.000	0.2	3536.4	48.5
$R^2 = 0.9$	4	0.011	0.2	2409.6	4.5
	8	0.005	0.2	2480.2	6.3
	16	0.002	0.5	3391.2	4.8
	32	0.002	0.7	2770.6	4.7
$R^2 = 0.75$	4	0.014	0.2	2450.5	6.2
	8	0.005	0.2	2601.6	6.4
	16	0.005	5.0	3508.2	9.3
	32	0.008	0.7	2740.6	1.6

Table 5.5: Cyclic Nonlinear Deterministic Results

	Number of Samples	MP vs Sim Deviation	MP vs Goal Deviation	Experiment Time (ms)	Solve Time (ms)
$R^2 = 1$	4	0.016	0.0	6199.6	100.0
	8	0.027	0.0	9603.4	707.7
	16	0.027	0.0	17066.2	12259.1
	32	0.028	0.0	30724.5	44949.5
$R^2 = 0.9$	4	0.016	0.0	5810.0	78.2
	8	0.026	0.0	7682.2	329.6
	16	0.033	0.0	8675.1	114.0
	32	0.032	0.0	9490.0	64.3
$R^2 = 0.75$	4	0.016	0.0	5649.5	65.7
	8	0.025	0.0	6588.6	79.8
	16	0.015	80.3	6963.5	48.4
	32	0.032	0.0	8112.5	35.7

5.4 Discussion of Results

The results from linear models, shown in table 5.3, are positive. Both models have values of zero for both deviation metrics! This means that the optimization output was always exactly the same as the actual simulation output. These models were essentially solved instantaneously, as noted by the solve times. The experiment time grows as the number of iterations increase; this makes sense because the experiment must be run for each iteration.

The nonlinear results, shown in tables 5.4 and 5.5 are also positive. Both models show relatively low deviations; less than 1.4 % for the acyclic case and less than 3.3 % for the cyclic case. For the acyclic model, the deviation score decreases (becomes better) as the number of samples taken increases. The cyclic case shows the opposite effect, with the deviation scores increasing (becoming worse) as more samples are taken.

A likely explanation for this behavior must be that small errors seen from approximating non-linear functions are compounded over each iteration. So, negligible improvements are seen from the extra data points provided from additional sampling. A second notable cause could be the increased complexity of solving larger mixed integer programs. It is also possible that multiple degenerate solutions exist, and the branch and bound tree terminated when the MIP gap was below the threshold.

Another important column in the results summary tables is the precision column. Precision is a measurement of how close the math program was able to get to the user specified goal. All goals specified to all test cases are achievable within the set input bounds, so the math program should have always been able to find input values that yielded the requested output. Precision values less than one in tables 5.4 and 5.5 are the effects of the sampling algorithm, and varying R^2 values. For example, the high accuracy in the cyclic non linear SDF model with 16 samples and $R^2 = 0.75$ occurred because of low precision. The exact results can be seen in table A.26; note that the reduced R^2 value eliminated feasible solutions less than 600.7. Thus a high accuracy is observed because the optimization

model was frequently accurate, the provided input would always yield 617.2 regardless of the requested goal.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The work presented in this thesis show that it is possible to use mathematical programming to optimize graphs that have a special structure. All scenarios listed in table 4.1 can be handled by the linear and integer programs (figures 4.10 and 4.12). Systems that operate similarly to SDF models should be able to solved using the methods outlined in this thesis. A system that operates similarly would be any graph whose vertices output values based on a function of that nodes inputs. However please note that only graphs elicited from SDF models have been tested, so this idea is purely speculation.

Unfortunately, the methods implemented in this work are not without fault. The results of the optimization models are highly dependent on the data provided to them. In many cases, having even one poorly fit actor can result in totally inaccurate output from the optimization model.

There are many common situations in SDF models that can easily produce inaccurate fits when using the current experimental sampling algorithm. Noncontinuous functions will not be correctly handled using the current sampling algorithm and mathematical program. Even linear functions with multiple variables will not be fit accurately unless a special sampling method is used! Many models pass an initial value into sample delay actors, such as an initial population or an initial inventory. A special sampling experiment is required if one wishes to optimize these initial values. The information abstraction methods should

be targeted first for improvement.

Although the methods presented in this thesis are not without fault, the results are overwhelmingly positive! Many scenarios can be optimized using the outlined optimization models, and the sampling experiment applies to a wide variety of situations. The examples and testing were completed in Java using Ptolemy as the simulation software, but the methods outlined in this work could easily be applied to other domains.

6.2 Future Work

There are many different areas of the methods outlined in this work that could be improved in the future. Some of the potential improvements are listed below.

One possible improvement could be creating information abstraction methods specific to a particular simulation domain or programming language. By limiting the scope of the experiments, code could be written to abstract the necessary information from efficiently and accurately. The obvious downside to this is the high restriction placed on the types of models that can be solved. A possible mitigation could be to create a hybrid approach where many actors have specific code written to abstract information correctly, and experimental sampling is relied upon to weight nodes or edges that are not supported.

The current piecewise linear fitting algorithm, in algorithm 4.1, requires the user to input a minimum acceptable R^2 value. The purpose of this value is to allow the user to adjust the quality of to the functions being experimented on The major downside to this approach is that piecewise functions are fit only to the sampled data, since the underlying functions are unknown. If the functions inside of actors in the simulation model were known, it is likely that better approximations could be generated. This may significantly improve the results of the cyclic nonlinear case.

In this work all SDF input values are treated as real numbers, however in many real life problems certain inputs must be integers. One simple way to force an input to take an

integer value, would be to set the sampling range of these inputs to only test integers and then require the relevant λ variables to be binary instead of real. This method could yield higher efficiency than just declaring the input to be integer, especially if the input range is broad and the modeler is not concerned with many small increments of integers.

Consider instances where the input must be an integer that is a multiple of some quantity. Perhaps the input sets an initial inventory, and orders can only be placed in increments of 10 (e.g. order 10 or 20 or 30...). Situations like this should gain efficiency in the branch and bound if the "binary λ" method was used; certainly when compared to defining that input over the complete integer range of 10 through 30.

Another possible improvement could come from sampling at a "higher level". If a graph elicitation method was written to group actors into aggregate (or composite) nodes, it is possible that noisy functions could have less effect in the optimization model. Unfortunately, it is also likely that a more complicated sampling experiment would be required to effectively implement this idea. In situations where the aggregate node would have many inputs, more complex algorithms may be necessary to ensure meaningful input combinations are sampled.

One important area that has not been considered in any of the test cases in this work are stochastic functions. Most simulations include stochastic functions, as many simulations are used to analyze a system subject to randomness! If a simulation model has many stochastic functions, it is possible that methods from robust optimization could be incorporated into the existing optimization models.

Robust optimization is a technique used to deal with uncertain data in mathematical modeling [4]. By specifying an uncertainty set for the constraints of the problem, one can produce solutions that are much less sensitive to minor perturbations in data. This methodology also avoids the overly conservative results of worst-case estimates of uncertain data.

An extension of robust optimization useful for this work would be the application of

robust optimization methods to remove the noise generated from stochastic variables in the simulation. A key development has shown that robust methods can be used even when the stochastic data is not normally distributed [6] . This should allow the perturbation of uncertainty sets regardless of the distribution of the stochastic variable.

Finally, it could be possible to make the optimization models solve more efficiently by taking advantage of the underlying graph structure. In general, graph based linear programs can be solved much more quickly than standard LP's since special algorithms can be used. The optimization models in this work do not exploit the graph structure, so it is possible that additional efficiencies can be realized by modifying some of the constraints. The graph structure could be used to remove constraints from the model that include nodes which are irrelevant to the goals. Getting rid of unnecessary constraints and variables from the optimization model will result in solutions being found more quickly. Determining important nodes could be done strictly or loosely, either by relying 100 % on the graph structure or by using factor screening techniques [20].

Bibliography

[1] F. Azadivar. A tutorial on simulation optimization. In *Proceedings of the 24th conference on Winter simulation*, pages 198–204. ACM, 1992.

[2] E.M.L. Beale. Advanced algorithmic features for general mathematical programming systems. *Integer and Nonlinear Programming, American Elsevier Publishing Company, New York*, pages 119–137, 1970.

[3] E.M.L. Beale and J.A. Tomlin. Special facilities in a general mathematical programming system for non-convex problems using ordered sets of variables. *OR*, 69:447–454, 1970.

[4] A. Ben-Tal and A. Nemirovski. Robust optimization–methodology and applications. *Mathematical Programming*, 92(3):453–480, 2002.

[5] C. Brooks, E.A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous concurrent modeling and design in Java (Volume 1: Introduction to Ptolemy II). *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-28, Apr*, 2008.

[6] X. Chen, M. Sim, and P. Sun. A robust optimization perspective on stochastic programming. *Operations Research*, 55(6):1058, 2007.

[7] G. B. Dantzig. On the significance of solving linear programs with some integer variables. *Econometrica*, 28:30–44, 1960.

[8] J. J. DiStefano, A. R. Stubberud, and I. J. Williams. *Schaums outline series of theory and problems of feedback and control systems*. McGraw-Hill, 1990.

[9] R. Fourer and D.M. Gay. Expressing special structures in an algebraic modeling language for mathematical programming. *ORSA Journal on Computing 7*, pages 166–190, 1995.

[10] M.C. Fu. Optimization for simulation: Theory vs. practice. *INFORMS Journal on Computing*, 14(3):192–215, 2002.

[11] JA Joines, RR Barton, K. Kang, PA Fishwick, J.R. Swisher, and S.H. Jacobson. A Survey Of Simulation Optimization Techniques And Procedures. 2000.

[12] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[13] J. T. Linderoth and M. W. P. Savelsbergh. A computational study of search strategies for mixed integer programming. *INFORMS J. Comput.*, 11(2):173–187, 1999.

[14] A. Matta. Simulation optimization with mathematical programming representation of discrete event systems. In *Proceedings of the 40th Conference on Winter Simulation*, pages 1393–1400. Winter Simulation Conference, 2008.

[15] D.C. Montgomery, E.A. Peck, G.G. Vining, and J. Vining. *Introduction to linear regression analysis*. Wiley New York, 2006.

[16] S. Olafsson and J. Kim. Simulation optimization. In *Proceedings of the 34th conference on Winter simulation*, pages 79–84. Winter Simulation Conference, 2002.

[17] R.L. Rardin. *Optimization in operations research*. Prentice Hall, 2000.

[18] L.W. Schruben. Mathematical programming models of discrete event system dynamics. In *Proceedings of the 32nd conference on Winter simulation*, pages 381–385. Society for Computer Simulation International, 2000.

[19] E. Stinstra and D. Den Hertog. Robust optimization using computer experiments. *European Journal of Operational Research*, 191(3):816–837, 2008.

[20] G. Tauer. A graph-based factor screening method for synchronous data flow simulation models. 2009.

[21] W. Venables and D. M. Smith. An Introduction to R. 2009.

[22] L.A. Wolsey. *Integer programming*. Wiley New York, 1998.

Appendix A

Data Tables

This appendix lists all data collected during the solution evaluation phase of the methodology, for each test case. A description of the column headings in the data tables is provided in table A.1.

Table A.1: Description of Data Table Columns

Goal Value	Desired output value.
Math Program Output	Expected simulation output reported by the optimization model.
Simulation Output	Actual output from the simulation model with the predicted inputs.
MP vs Sim Deviation	$\lvert 1 - \frac{\text{Simulation Output}}{\text{Math Program Output}} \rvert$
MP vs Goal Deviation	$\lvert (\text{Math Program Output}) - (\text{Simulation Output}) \rvert$
Total Time	Sum of Solve Time, Model Prep Time, and Experiment Time.
Solve Time	Time (ms) required to solve the optimization model.
Model Prep Time	Time (ms) required to prepare sets and data parameters for the optimization model.
Experiment Time	Time (ms) required to elicit graph from SDF model and perform all experimental sampling and fitting (including the piecewise linear fitting algorithm).

A.1 Acyclic Linear Deterministic Model

These are the results from the acyclic linear deterministic SDF model.

Simple linear models should all exhibit the same behavior as the results shown in table A.2; low deviations and low run time. The average run time of 2398.4 ms is very fast, the 1.6 ms average solve time is essentially instantaneous.

Table A.2: Acyclic Linear Deterministic Results: R^2 = 1, Samples = 8, Iterations = 1

Goal Value	Math Program Output	Simulation Output	MP vs Sim Deviation	MP vs Goal Deviation	Total Time (ms)	Solve Time (ms)	Model Prep Time (ms)	Experiment Time (ms)
500.0	500.0	500.0	0.000	0.0	2421.0	16.0	109.0	2296.0
550.0	550.0	550.0	0.000	0.0	2374.0	0.0	109.0	2265.0
600.0	600.0	600.0	0.000	0.0	2407.0	0.0	110.0	2297.0
650.0	650.0	650.0	0.000	0.0	2406.0	0.0	109.0	2297.0
700.0	700.0	700.0	0.000	0.0	2391.0	0.0	110.0	2281.0
750.0	750.0	750.0	0.000	0.0	2390.0	0.0	109.0	2281.0
800.0	800.0	800.0	0.000	0.0	2390.0	0.0	110.0	2280.0
850.0	850.0	850.0	0.000	0.0	2376.0	0.0	110.0	2266.0
900.0	900.0	900.0	0.000	0.0	2422.0	0.0	109.0	2313.0
950.0	950.0	950.0	0.000	0.0	2407.0	0.0	110.0	2297.0
Averages:			**0.000**	**0.0**	**2398.4**	**1.6**	**109.5**	**2287.3**

A.2 Cyclic Linear Deterministic Model

These are the results from the cyclic linear deterministic SDF model.

Table A.3: Cyclic Linear Deterministic Results: R^2 = 1, Samples = 8, Iterations = 25

Goal Value	Math Program Output	Simulation Output	MP vs Sim Deviation	MP vs Goal Deviation	Total Time (ms)	Solve Time (ms)	Model Prep Time (ms)	Experiment Time (ms)
500.0	500.0	500.0	0.000	0.0	4485.0	0.0	125.0	4360.0
550.0	550.0	550.0	0.000	0.0	5500.0	0.0	156.0	5344.0
600.0	600.0	600.0	0.000	0.0	5422.0	0.0	140.0	5282.0
650.0	650.0	650.0	0.000	0.0	5329.0	16.0	125.0	5188.0
700.0	700.0	700.0	0.000	0.0	4531.0	16.0	109.0	4406.0
750.0	750.0	750.0	0.000	0.0	4584.0	0.0	125.0	4459.0
800.0	800.0	800.0	0.000	0.0	4538.0	0.0	125.0	4413.0
850.0	850.0	850.0	0.000	0.0	4522.0	0.0	125.0	4397.0
900.0	900.0	900.0	0.000	0.0	4503.0	16.0	109.0	4378.0
950.0	950.0	950.0	0.000	0.0	4518.0	0.0	125.0	4393.0
Averages:			**0.000**	**0.0**	**4793.2**	**4.8**	**126.4**	**4662.0**

The results of the seconds linear test case are just as positive as the first. Here, the deviation scores are zero and the run time is low. Even though this model has been run for 25 iterations, there is a minimal run time penalty.

A.3 Acyclic Nonlinear Deterministic Model

These are the results from the acyclic nonlinear deterministic SDF model.

Table A.4: Acyclic Nonlinear Deterministic Results: R^2 = 1, Samples = 4, Iterations = 1

Goal Value	Math Program Output	Simulation Output	MP vs Sim Deviation	MP vs Goal Deviation	Total Time (ms)	Solve Time (ms)	Model Prep Time (ms)	Experiment Time (ms)
5.0	5.8	5.8	0.000	0.8	2968.0	0.0	140.0	2828.0
10.0	10.0	9.9	0.007	0.0	2564.0	0.0	109.0	2455.0
15.0	15.0	14.4	0.043	0.0	2642.0	16.0	156.0	2470.0
20.0	20.0	19.6	0.018	0.0	2564.0	16.0	109.0	2439.0
25.0	25.0	24.5	0.020	0.0	2548.0	0.0	109.0	2439.0
30.0	30.0	29.4	0.020	0.0	2579.0	15.0	110.0	2454.0
35.0	35.0	34.4	0.017	0.0	2642.0	0.0	157.0	2485.0
40.0	40.0	39.5	0.012	0.0	2596.0	16.0	110.0	2470.0
45.0	45.0	44.7	0.006	0.0	2611.0	0.0	109.0	2502.0
50.0	49.2	49.2	0.000	0.8	2595.0	0.0	156.0	2439.0
Averages:			**0.014**	**0.2**	**2630.9**	**6.3**	**126.5**	**2498.1**

Table A.5: Acyclic Nonlinear Deterministic Results: R^2 = 1, Samples = 8, Iterations = 1

Goal Value	Math Program Output	Simulation Output	MP vs Sim Deviation	MP vs Goal Deviation	Total Time (ms)	Solve Time (ms)	Model Prep Time (ms)	Experiment Time (ms)
5.0	6.0	6.0	0.000	1.0	4034.0	0.0	1329.0	2705.0
10.0	10.0	10.1	0.010	0.0	2799.0	16.0	109.0	2674.0
15.0	15.0	15.1	0.009	0.0	2799.0	16.0	110.0	2673.0
20.0	20.0	20.1	0.005	0.0	2798.0	0.0	125.0	2673.0
25.0	25.0	25.0	0.000	0.0	2736.0	15.0	110.0	2611.0
30.0	30.0	30.0	0.001	0.0	2783.0	16.0	109.0	2658.0
35.0	35.0	34.9	0.002	0.0	2752.0	15.0	110.0	2627.0
40.0	40.0	39.9	0.002	0.0	2815.0	16.0	109.0	2690.0
45.0	45.0	44.8	0.004	0.0	2815.0	0.0	125.0	2690.0
50.0	49.1	49.1	0.000	0.9	2720.0	15.0	110.0	2595.0
Averages:			**0.003**	**0.2**	**2905.1**	**10.9**	**234.6**	**2659.6**

With R^2 = 1 (figures A.4 through A.7), the results are predictable for the acyclic nonlinear deterministic case. As the number of samples increase, the deviations decrease. Unfortunately, the run times of the models with more samples are much higher than those with fewer. The solve time is significantly higher in these cases as well; this is most attributable to the increase in binary variables in the math program.

Table A.6: Acyclic Nonlinear Deterministic Results: R^2 = 1, Samples = 16, Iterations = 1

Goal Value	Math Program Output	Simulation Output	MP vs Sim Deviation	MP vs Goal Deviation	Total Time (ms)	Solve Time (ms)	Model Prep Time (ms)	Experiment Time (ms)
5.0	6.0	6.0	0.000	1.0	3033.0	16.0	109.0	2908.0
10.0	10.0	10.0	0.002	0.0	3064.0	31.0	109.0	2924.0
15.0	15.0	15.0	0.002	0.0	3080.0	31.0	125.0	2924.0
20.0	20.0	20.0	0.000	0.0	3049.0	16.0	109.0	2924.0
25.0	25.0	25.0	0.000	0.0	3064.0	31.0	110.0	2923.0
30.0	30.0	30.0	0.000	0.0	3112.0	47.0	110.0	2955.0
35.0	35.0	35.0	0.000	0.0	3033.0	31.0	125.0	2877.0
40.0	40.0	40.0	0.000	0.0	3080.0	47.0	156.0	2877.0
45.0	45.0	45.0	0.001	0.0	3049.0	47.0	109.0	2893.0
50.0	49.0	49.0	0.000	1.0	3064.0	15.0	110.0	2939.0
Averages:			**0.001**	**0.2**	**3062.8**	**31.2**	**117.2**	**2914.4**

Table A.7: Acyclic Nonlinear Deterministic Results: R^2 = 1, Samples = 32, Iterations = 1

Goal Value	Math Program Output	Simulation Output	MP vs Sim Deviation	MP vs Goal Deviation	Total Time (ms)	Solve Time (ms)	Model Prep Time (ms)	Experiment Time (ms)
5.0	6.0	6.0	0.000	1.0	3643.0	16.0	109.0	3518.0
10.0	10.0	10.0	0.000	0.0	3690.0	47.0	110.0	3533.0
15.0	15.0	15.0	0.000	0.0	3690.0	63.0	109.0	3518.0
20.0	20.0	20.0	0.000	0.0	3690.0	47.0	125.0	3518.0
25.0	25.0	25.0	0.000	0.0	3705.0	62.0	110.0	3533.0
30.0	30.0	30.0	0.000	0.0	3705.0	62.0	126.0	3517.0
35.0	35.0	35.0	0.000	0.0	3721.0	63.0	125.0	3533.0
40.0	40.0	40.0	0.000	0.0	3674.0	47.0	125.0	3502.0
45.0	45.0	45.0	0.000	0.0	3721.0	63.0	109.0	3549.0
50.0	49.0	49.0	0.000	1.0	3768.0	15.0	110.0	3643.0
Averages:			**0.000**	**0.2**	**3700.7**	**48.5**	**115.8**	**3536.4**

Reducing the R^2 value from 1 to 0.9 (tables A.8 through A.11) has an effect on the solution quality. Run time is reduced at the expense of deviation! Since fewer lines are fit to the output of nodes, certain sampled points are excluded from the regression line. These excluded points increase the MP vs goal deviation values.

Further reducing the R^2 value from 0.9 to 0.75 (tables A.12 through A.15) continues the observed trends. Deviations are increased while lower run times are observed. This is because the piecewise fitting algorithm completes in fewer iterations by fitting less accurate lines to the non linear output. Having less breakpoints in the piecewise functions require

Table A.8: Acyclic Nonlinear Deterministic Results: R^2 = 0.9, Samples = 4, Iterations = 1

Goal Value	Math Program Output	Simulation Output	MP vs Sim Deviation	MP vs Goal Deviation	Total Time (ms)	Solve Time (ms)	Model Prep Time (ms)	Experiment Time (ms)
5.0	5.8	5.8	0.000	0.8	2562.0	0.0	109.0	2453.0
10.0	10.0	9.9	0.007	0.0	2563.0	0.0	110.0	2453.0
15.0	15.0	14.8	0.013	0.0	2532.0	0.0	156.0	2376.0
20.0	20.0	19.6	0.018	0.0	2531.0	0.0	156.0	2375.0
25.0	25.0	24.5	0.020	0.0	2578.0	15.0	157.0	2406.0
30.0	30.0	29.4	0.020	0.0	2531.0	0.0	156.0	2375.0
35.0	35.0	34.4	0.017	0.0	2563.0	0.0	156.0	2407.0
40.0	40.0	39.5	0.012	0.0	2531.0	15.0	110.0	2406.0
45.0	45.0	44.7	0.006	0.0	2547.0	15.0	141.0	2391.0
50.0	49.2	49.2	0.000	0.8	2563.0	0.0	109.0	2454.0
Averages:			**0.011**	**0.2**	**2550.1**	**4.5**	**136.0**	**2409.6**

Table A.9: Acyclic Nonlinear Deterministic Results: R^2 = 0.9, Samples = 8, Iterations = 1

Goal Value	Math Program Output	Simulation Output	MP vs Sim Deviation	MP vs Goal Deviation	Total Time (ms)	Solve Time (ms)	Model Prep Time (ms)	Experiment Time (ms)
5.0	6.0	6.0	0.000	1.0	2657.0	16.0	141.0	2500.0
10.0	10.0	10.2	0.016	0.0	2656.0	15.0	188.0	2453.0
15.0	15.0	15.2	0.012	0.0	2610.0	0.0	156.0	2454.0
20.0	20.0	20.0	0.002	0.0	2626.0	16.0	140.0	2470.0
25.0	25.0	25.1	0.005	0.0	2641.0	0.0	156.0	2485.0
30.0	30.0	29.6	0.014	0.0	2610.0	16.0	109.0	2485.0
35.0	35.0	35.0	0.000	0.0	2641.0	0.0	156.0	2485.0
40.0	40.0	39.9	0.001	0.0	2610.0	0.0	156.0	2454.0
45.0	45.0	44.9	0.001	0.0	2626.0	0.0	110.0	2516.0
50.0	49.1	49.1	0.000	0.9	2609.0	0.0	109.0	2500.0
Averages:			**0.005**	**0.2**	**2628.6**	**6.3**	**142.1**	**2480.2**

fewer binary variables in the math programs, yielding faster solve times.

Table A.10: Acyclic Nonlinear Deterministic Results: R^2 = 0.9, Samples = 16, Iterations = 1

Goal Value	**Math Program Output**	**Simulation Output**	**MP vs Sim Deviation**	**MP vs Goal Deviation**	**Total Time (ms)**	**Solve Time (ms)**	**Model Prep Time (ms)**	**Experiment Time (ms)**
5.0	8.4	8.4	0.000	3.4	11300.0	16.0	328.0	10956.0
10.0	10.0	10.0	0.003	0.0	2672.0	0.0	109.0	2563.0
15.0	15.0	15.1	0.004	0.0	2813.0	0.0	125.0	2688.0
20.0	20.0	20.0	0.002	0.0	2657.0	0.0	110.0	2547.0
25.0	25.0	25.0	0.001	0.0	2626.0	0.0	110.0	2516.0
30.0	30.0	29.9	0.003	0.0	2641.0	16.0	109.0	2516.0
35.0	35.0	34.9	0.004	0.0	2626.0	0.0	110.0	2516.0
40.0	40.0	39.9	0.004	0.0	2461.0	0.0	110.0	2531.0
45.0	45.0	44.9	0.001	0.0	2657.0	16.0	109.0	2532.0
50.0	48.3	48.3	0.000	1.7	2656.0	0.0	109.0	2547.0
Averages:			**0.002**	**0.5**	**3510.9**	**4.8**	**132.9**	**3391.2**

Table A.11: Acyclic Nonlinear Deterministic Results: R^2 = 0.9, Samples = 32, Iterations = 1

Goal Value	**Math Program Output**	**Simulation Output**	**MP vs Sim Deviation**	**MP vs Goal Deviation**	**Total Time (ms)**	**Solve Time (ms)**	**Model Prep Time (ms)**	**Experiment Time (ms)**
5.0	10.2	10.2	0.000	5.2	2828.0	15.0	110.0	2703.0
10.0	10.2	10.2	0.000	0.2	2719.0	0.0	109.0	2610.0
15.0	15.0	15.0	0.001	0.0	2735.0	0.0	110.0	2625.0
20.0	20.0	20.0	0.001	0.0	2719.0	0.0	110.0	2609.0
25.0	25.0	24.9	0.003	0.0	2751.0	16.0	109.0	2626.0
30.0	30.0	29.9	0.005	0.0	2735.0	0.0	110.0	2625.0
35.0	35.0	34.8	0.005	0.0	2735.0	16.0	109.0	2610.0
40.0	40.0	39.8	0.005	0.0	3329.0	0.0	141.0	3188.0
45.0	45.0	44.9	0.002	0.0	3594.0	0.0	125.0	3469.0
50.0	48.7	48.7	0.000	1.3	2451.0	0.0	110.0	2641.0
Averages:			**0.002**	**0.7**	**2859.6**	**4.7**	**114.3**	**2770.6**

Table A.12: Acyclic Nonlinear Deterministic Results: R^2 = 0.75, Samples = 4, Iterations = 1

Goal Value	**Math Program Output**	**Simulation Output**	**MP vs Sim Deviation**	**MP vs Goal Deviation**	**Total Time (ms)**	**Solve Time (ms)**	**Model Prep Time (ms)**	**Experiment Time (ms)**
5.0	5.8	5.8	0.000	0.8	2532.0	0.0	125.0	2407.0
10.0	10.0	9.9	0.007	0.0	2563.0	16.0	109.0	2438.0
15.0	15.0	15.6	0.039	0.0	2563.0	15.0	157.0	2391.0
20.0	20.0	20.7	0.034	0.0	2844.0	15.0	110.0	2719.0
25.0	25.0	25.7	0.026	0.0	2563.0	0.0	109.0	2454.0
30.0	30.0	30.5	0.017	0.0	2532.0	0.0	110.0	2422.0
35.0	35.0	35.3	0.009	0.0	2563.0	0.0	109.0	2454.0
40.0	40.0	40.1	0.003	0.0	2563.0	0.0	157.0	2406.0
45.0	45.0	44.7	0.006	0.0	2579.0	16.0	156.0	2407.0
50.0	49.2	49.2	0.000	0.8	2563.0	0.0	156.0	2407.0
Averages:			**0.014**	**0.2**	**2586.5**	**6.2**	**129.8**	**2450.5**

Table A.13: Acyclic Nonlinear Deterministic Results: R^2 = 0.75, Samples = 8, Iterations = 1

Goal Value	**Math Program Output**	**Simulation Output**	**MP vs Sim Deviation**	**MP vs Goal Deviation**	**Total Time (ms)**	**Solve Time (ms)**	**Model Prep Time (ms)**	**Experiment Time (ms)**
5.0	6.0	6.0	0.000	1.0	3297.0	0.0	140.0	3157.0
10.0	10.0	10.2	0.016	0.0	3204.0	0.0	188.0	3016.0
15.0	15.0	15.2	0.012	0.0	2626.0	16.0	109.0	2501.0
20.0	20.0	20.0	0.002	0.0	2626.0	0.0	125.0	2501.0
25.0	25.0	24.8	0.008	0.0	2626.0	0.0	157.0	2469.0
30.0	30.0	29.6	0.014	0.0	2594.0	16.0	109.0	2469.0
35.0	35.0	35.0	0.000	0.0	2610.0	0.0	157.0	2453.0
40.0	40.0	39.9	0.001	0.0	2625.0	16.0	156.0	2453.0
45.0	45.0	44.9	0.001	0.0	2625.0	0.0	156.0	2496.0
50.0	49.1	49.1	0.000	0.9	2626.0	16.0	109.0	2501.0
Averages:			**0.005**	**0.2**	**2745.9**	**6.4**	**140.6**	**2601.6**

Table A.14: Acyclic Nonlinear Deterministic Results: R^2 = 0.75, Samples = 16, Iterations = 1

Goal Value	Math Program Output	Simulation Output	MP vs Sim Deviation	MP vs Goal Deviation	Total Time (ms)	Solve Time (ms)	Model Prep Time (ms)	Experiment Time (ms)
5.0	24.6	24.7	0.004	19.6	7407.0	0.0	172.0	7235.0
10.0	24.6	24.7	0.004	14.6	2875.0	0.0	109.0	2766.0
15.0	24.6	24.7	0.004	9.6	3656.0	15.0	125.0	3516.0
20.0	24.6	24.7	0.004	4.6	3234.0	0.0	140.0	3094.0
25.0	25.0	25.1	0.003	0.0	3297.0	16.0	125.0	3156.0
30.0	30.0	29.7	0.008	0.0	3266.0	0.0	125.0	3141.0
35.0	35.0	34.5	0.013	0.0	3203.0	15.0	125.0	3063.0
40.0	40.0	39.6	0.010	0.0	3282.0	16.0	140.0	3126.0
45.0	45.0	44.9	0.001	0.0	3172.0	15.0	125.0	3032.0
50.0	48.3	48.3	0.000	1.7	3094.0	16.0	125.0	2953.0
Averages:			**0.005**	**5.0**	**3648.6**	**9.3**	**131.1**	**3508.2**

Table A.15: Acyclic Nonlinear Deterministic Results: R^2 = 0.75, Samples = 32, Iterations = 1

Goal Value	Math Program Output	Simulation Output	MP vs Sim Deviation	MP vs Goal Deviation	Total Time (ms)	Solve Time (ms)	Model Prep Time (ms)	Experiment Time (ms)
5.0	10.2	10.2	0.000	5.2	2721.0	0.0	110.0	2611.0
10.0	10.2	10.2	0.000	0.2	2674.0	0.0	110.0	2564.0
15.0	15.0	15.4	0.028	0.0	2721.0	0.0	109.0	2612.0
20.0	20.0	20.5	0.025	0.0	2752.0	0.0	109.0	2643.0
25.0	25.0	25.3	0.011	0.0	3112.0	0.0	125.0	2987.0
30.0	30.0	29.9	0.003	0.0	2705.0	0.0	110.0	2595.0
35.0	35.0	35.1	0.004	0.0	2658.0	0.0	109.0	2549.0
40.0	40.0	39.6	0.010	0.0	2705.0	0.0	110.0	2595.0
45.0	45.0	44.9	0.001	0.0	3406.0	0.0	109.0	3297.0
50.0	48.7	48.7	0.000	1.3	3078.0	16.0	109.0	2953.0
Averages:			**0.008**	**0.7**	**2853.2**	**1.6**	**111.0**	**2740.6**

A.4 Cyclic Nonlinear Deterministic Model

These are the results from the cyclic nonlinear deterministic SDF model.

Table A.16: Cyclic Nonlinear Deterministic Results: R^2 = 1, Samples = 4, Iterations = 25

Goal Value	Math Program Output	Simulation Output	MP vs Sim Deviation	MP vs Goal Deviation	Total Time (ms)	Solve Time (ms)	Model Prep Time (ms)	Experiment Time (ms)
250.0	250.0	266.5	0.066	0.0	6827.0	156.0	141.0	6530.0
350.0	350.0	362.0	0.034	0.0	6358.0	94.0	141.0	6123.0
450.0	450.0	456.9	0.015	0.0	6452.0	141.0	140.0	6171.0
550.0	550.0	552.0	0.004	0.0	6452.0	140.0	203.0	6109.0
650.0	650.0	647.6	0.004	0.0	6452.0	94.0	140.0	6218.0
750.0	750.0	744.2	0.008	0.0	6313.0	94.0	141.0	6078.0
850.0	850.0	865.9	0.019	0.0	6344.0	78.0	141.0	6125.0
950.0	950.0	960.0	0.011	0.0	6328.0	78.0	140.0	6110.0
1050.0	1050.0	1043.1	0.007	0.0	6438.0	63.0	203.0	6172.0
1150.0	1150.0	1146.3	0.003	0.0	6578.0	62.0	156.0	6360.0
Averages:			**0.017**	**0.0**	**6454.2**	**100.0**	**154.6**	**6199.6**

Table A.17: Cyclic Nonlinear Deterministic Results: R^2 = 1, Samples = 8, Iterations = 25

Goal Value	Math Program Output	Simulation Output	MP vs Sim Deviation	MP vs Goal Deviation	Total Time (ms)	Solve Time (ms)	Model Prep Time (ms)	Experiment Time (ms)
250.0	250.0	274.4	0.097	0.0	10078.0	281.0	156.0	9641.0
350.0	350.0	371.8	0.062	0.0	10484.0	703.0	156.0	9625.0
450.0	450.0	469.0	0.042	0.0	10391.0	593.0	157.0	9641.0
550.0	550.0	564.1	0.026	0.0	10126.0	422.0	156.0	9548.0
650.0	650.0	661.8	0.018	0.0	10390.0	609.0	156.0	9625.0
750.0	750.0	759.0	0.012	0.0	10344.0	625.0	156.0	9563.0
850.0	850.0	856.6	0.008	0.0	10469.0	782.0	218.0	9469.0
950.0	950.0	954.5	0.005	0.0	10516.0	782.0	156.0	9578.0
1050.0	1050.0	1051.4	0.001	0.0	11547.0	1562.0	141.0	9844.0
1150.0	1150.0	1147.8	0.002	0.0	10437.0	718.0	129.0	9500.0
Averages:			**0.027**	**0.0**	**10478.2**	**707.7**	**158.1**	**9603.4**

The cyclic nonlinear deterministic (CND) model with R^2 = 1 (tables A.16 through A.19) shows results with the acyclic nonlinear deterministic (AND) example model. Since the CND model was executed for 25 iterations, all the run times are longer than the AND model. The experiment and optimization times become significantly longer as more sample points are taken. Also, the MP vs simulation deviation appears best when the fewest

Table A.18: Cyclic Nonlinear Deterministic Results: R^2 = 1, Samples = 16, Iterations = 25

Goal Value	Math Program Output	Simulation Output	MP vs Sim Deviation	MP vs Goal Deviation	Total Time (ms)	Solve Time (ms)	Model Prep Time (ms)	Experiment Time (ms)
250.0	250.0	272.3	0.089	0.0	19891.0	3297.0	172.0	16423.0
350.0	350.0	370.5	0.059	0.0	32105.0	15514.0	172.0	16419.0
450.0	450.0	467.7	0.039	0.0	21810.0	5249.0	204.0	16357.0
550.0	550.0	565.5	0.028	0.0	23356.0	6733.0	235.0	16388.0
650.0	650.0	663.0	0.020	0.0	22200.0	5546.0	172.0	16482.0
750.0	750.0	760.7	0.014	0.0	21937.0	5485.0	156.0	16296.0
850.0	850.0	858.1	0.010	0.0	20969.0	4375.0	250.0	16344.0
950.0	950.0	956.0	0.006	0.0	38359.0	21953.0	250.0	16156.0
1050.0	1050.0	1053.4	0.003	0.0	50126.0	28860.0	297.0	20969.0
1150.0	1150.0	1151.2	0.001	0.0	44657.0	25579.0	250.0	18828.0
Averages:			**0.027**	**0.0**	**29541.0**	**12259.1**	**215.8**	**17066.2**

Table A.19: Cyclic Nonlinear Deterministic Results: R^2 = 1, Samples = 32, Iterations = 25

Goal Value	Math Program Output	Simulation Output	MP vs Sim Deviation	MP vs Goal Deviation	Total Time (ms)	Solve Time (ms)	Model Prep Time (ms)	Experiment Time (ms)
250.0	250.0	272.3	0.089	0.0	43439.0	12500.0	188.0	30751.0
350.0	350.0	369.9	0.057	0.0	84175.0	53893.0	265.0	30017.0
450.0	450.0	467.7	0.039	0.0	54315.0	23392.0	203.0	30720.0
550.0	550.0	565.4	0.028	0.0	90972.0	60455.0	266.0	30251.0
650.0	650.0	663.1	0.020	0.0	99815.0	64142.0	203.0	35470.0
750.0	750.0	760.8	0.014	0.0	69877.0	40126.0	188.0	29563.0
850.0	850.0	867.6	0.021	0.0	73080.0	42985.0	250.0	29845.0
950.0	950.0	956.1	0.006	0.0	68126.0	37704.0	188.0	30234.0
1050.0	1050.0	1053.8	0.004	0.0	102959.0	71905.0	188.0	30866.0
1150.0	1150.0	1151.6	0.001	0.0	72109.0	42393.0	188.0	29528.0
Averages:			**0.028**	**0.0**	**75886.7**	**44949.5**	**212.7**	**30724.5**

number of samples are taken. This is because the same sample points are not taken with each increment of additional sample points. This example shows why information about the function inside an actor would be useful, as a better fit could be possible (the potential to yield lower deviations).

Just as in the AND model, reducing the R^2 value yields higher deviations while decreasing run times.

Table A.20: Cyclic Nonlinear Deterministic Results: R^2 = 0.9, Samples = 4, Iterations = 25

Goal Value	**Math Program Output**	**Simulation Output**	**MP vs Sim Deviation**	**MP vs Goal Deviation**	**Total Time (ms)**	**Solve Time (ms)**	**Model Prep Time (ms)**	**Experiment Time (ms)**
250.0	250.0	266.5	0.066	0.0	6110.0	78.0	141.0	5891.0
350.0	350.0	362.0	0.034	0.0	6001.0	94.0	141.0	5766.0
450.0	450.0	456.9	0.015	0.0	6017.0	94.0	141.0	5782.0
550.0	550.0	552.0	0.004	0.0	6048.0	109.0	157.0	5782.0
650.0	650.0	647.6	0.004	0.0	6017.0	110.0	141.0	5766.0
750.0	750.0	744.2	0.008	0.0	6032.0	62.0	141.0	5829.0
850.0	850.0	842.1	0.009	0.0	6095.0	63.0	156.0	5876.0
950.0	950.0	941.7	0.009	0.0	6001.0	63.0	140.0	5798.0
1050.0	1050.0	1043.1	0.007	0.0	6032.0	62.0	141.0	5829.0
1150.0	1150.0	1146.3	0.003	0.0	5985.0	47.0	157.0	5781.0
Averages:			**0.016**	**0.0**	**6033.8**	**78.2**	**145.6**	**5810.0**

Table A.21: Cyclic Nonlinear Deterministic Results: R^2 = 0.9, Samples = 8, Iterations = 25

Goal Value	**Math Program Output**	**Simulation Output**	**MP vs Sim Deviation**	**MP vs Goal Deviation**	**Total Time (ms)**	**Solve Time (ms)**	**Model Prep Time (ms)**	**Experiment Time (ms)**
250.0	250.0	274.6	0.098	0.0	8001.0	187.0	156.0	7658.0
350.0	350.0	372.7	0.065	0.0	8079.0	297.0	219.0	7563.0
450.0	450.0	468.1	0.040	0.0	8079.0	250.0	141.0	7688.0
550.0	550.0	561.9	0.022	0.0	8172.0	406.0	219.0	7547.0
650.0	650.0	655.1	0.008	0.0	8173.0	344.0	203.0	7626.0
750.0	750.0	748.9	0.001	0.0	8173.0	375.0	203.0	7595.0
850.0	850.0	844.5	0.007	0.0	8782.0	343.0	157.0	8282.0
950.0	950.0	942.6	0.008	0.0	8251.0	437.0	219.0	7595.0
1050.0	1050.0	1043.8	0.006	0.0	8173.0	422.0	140.0	7611.0
1150.0	1150.0	1148.4	0.001	0.0	8033.0	235.0	141.0	7657.0
Averages:			**0.026**	**0.0**	**8191.6**	**329.6**	**179.8**	**7682.2**

Table A.22: Cyclic Nonlinear Deterministic Results: R^2 = 0.9, Samples = 16, Iterations = 25

Goal Value	**Math Program Output**	**Simulation Output**	**MP vs Sim Deviation**	**MP vs Goal Deviation**	**Total Time (ms)**	**Solve Time (ms)**	**Model Prep Time (ms)**	**Experiment Time (ms)**
250.0	250.0	277.4	0.110	0.0	8751.0	156.0	156.0	8439.0
350.0	350.0	379.5	0.084	0.0	8855.0	78.0	219.0	8558.0
450.0	450.0	476.9	0.060	0.0	8714.0	124.0	157.0	8433.0
550.0	550.0	570.6	0.038	0.0	8901.0	156.0	156.0	8589.0
650.0	650.0	662.3	0.019	0.0	10839.0	110.0	187.0	10542.0
750.0	750.0	753.8	0.005	0.0	8698.0	109.0	140.0	8449.0
850.0	850.0	847.4	0.003	0.0	8667.0	78.0	203.0	8386.0
950.0	950.0	944.6	0.006	0.0	8731.0	125.0	218.0	8388.0
1050.0	1050.0	1046.5	0.003	0.0	8749.0	94.0	140.0	8515.0
1150.0	1150.0	1150.2	0.000	0.0	8702.0	110.0	140.0	8452.0
Averages:			**0.033**	**0.0**	**8960.7**	**114.0**	**171.6**	**8675.1**

Table A.23: Cyclic Nonlinear Deterministic Results: R^2 = 0.9, Samples = 32, Iterations = 25

Goal Value	**Math Program Output**	**Simulation Output**	**MP vs Sim Deviation**	**MP vs Goal Deviation**	**Total Time (ms)**	**Solve Time (ms)**	**Model Prep Time (ms)**	**Experiment Time (ms)**
250.0	250.0	273.9	0.096	0.0	9610.0	47.0	141.0	9422.0
350.0	350.0	380.0	0.086	0.0	9625.0	63.0	156.0	9406.0
450.0	450.0	480.4	0.068	0.0	9704.0	63.0	156.0	9485.0
550.0	550.0	575.6	0.046	0.0	9641.0	63.0	140.0	9438.0
650.0	650.0	657.4	0.011	0.0	9672.0	78.0	141.0	9453.0
750.0	750.0	753.1	0.004	0.0	9686.0	63.0	156.0	9467.0
850.0	850.0	849.3	0.001	0.0	9717.0	78.0	156.0	9483.0
950.0	950.0	945.1	0.005	0.0	9655.0	47.0	156.0	9452.0
1050.0	1050.0	1046.3	0.004	0.0	9655.0	47.0	156.0	9452.0
1150.0	1150.0	1149.7	0.000	0.0	10077.0	94.0	141.0	9842.0
Averages:			**0.032**	**0.0**	**9704.2**	**64.3**	**149.9**	**9490.0**

Table A.24: Cyclic Nonlinear Deterministic Results: R^2 = 0.75, Samples = 4, Iterations = 25

Goal Value	Math Program Output	Simulation Output	MP vs Sim Deviation	MP vs Goal Deviation	Total Time (ms)	Solve Time (ms)	Model Prep Time (ms)	Experiment Time (ms)
250.0	250.0	266.5	0.066	0.0	6454.0	62.0	157.0	6235.0
350.0	350.0	361.9	0.034	0.0	5783.0	63.0	140.0	5580.0
450.0	450.0	455.8	0.013	0.0	5782.0	78.0	140.0	5564.0
550.0	550.0	552.0	0.004	0.0	5783.0	63.0	140.0	5580.0
650.0	650.0	647.6	0.004	0.0	5797.0	78.0	140.0	5579.0
750.0	750.0	744.2	0.008	0.0	5907.0	78.0	203.0	5626.0
850.0	850.0	842.1	0.009	0.0	5844.0	62.0	219.0	5563.0
950.0	950.0	941.7	0.009	0.0	6954.0	62.0	1282.0	5610.0
1050.0	1050.0	1043.1	0.007	0.0	5861.0	79.0	203.0	5579.0
1150.0	1150.0	1146.3	0.003	0.0	5767.0	32.0	156.0	5579.0
Averages:			**0.016**	**0.0**	**5993.2**	**65.7**	**278.0**	**5649.5**

Table A.25: Cyclic Nonlinear Deterministic Results: R^2 = 0.75, Samples = 8, Iterations = 25

Goal Value	Math Program Output	Simulation Output	MP vs Sim Deviation	MP vs Goal Deviation	Total Time (ms)	Solve Time (ms)	Model Prep Time (ms)	Experiment Time (ms)
250.0	250.0	274.6	0.098	0.0	6830.0	63.0	219.0	6548.0
350.0	350.0	372.6	0.065	0.0	6876.0	78.0	141.0	6657.0
450.0	450.0	468.1	0.040	0.0	6798.0	63.0	156.0	6579.0
550.0	550.0	561.9	0.022	0.0	6813.0	62.0	203.0	6548.0
650.0	650.0	654.0	0.006	0.0	6891.0	62.0	219.0	6610.0
750.0	750.0	749.2	0.001	0.0	6845.0	63.0	230.0	6579.0
850.0	850.0	844.5	0.007	0.0	6876.0	78.0	203.0	6595.0
950.0	950.0	942.6	0.008	0.0	6845.0	94.0	172.0	6579.0
1050.0	1050.0	1043.8	0.006	0.0	6908.0	78.0	219.0	6611.0
1150.0	1150.0	1148.4	0.001	0.0	6877.0	157.0	140.0	6580.0
Averages:			**0.025**	**0.0**	**6855.9**	**79.8**	**190.2**	**6588.6**

Table A.26: Cyclic Nonlinear Deterministic Results: R^2 = 0.75, Samples = 16, Iterations = 25

Goal Value	**Math Program Output**	**Simulation Output**	**MP vs Sim Deviation**	**MP vs Goal Deviation**	**Total Time (ms)**	**Solve Time (ms)**	**Model Prep Time (ms)**	**Experiment Time (ms)**
250.0	600.7	617.2	0.028	350.7	7860.0	47.0	156.0	7657.0
350.0	600.7	617.2	0.028	250.7	7048.0	47.0	156.0	6845.0
450.0	600.7	617.2	0.028	150.7	7111.0	47.0	141.0	6923.0
550.0	600.7	617.2	0.028	50.7	7079.0	47.0	141.0	6891.0
650.0	650.0	662.3	0.019	0.0	7141.0	62.0	141.0	6938.0
750.0	750.0	753.8	0.005	0.0	7095.0	62.0	141.0	6892.0
850.0	850.0	847.4	0.003	0.0	7017.0	32.0	140.0	6845.0
950.0	950.0	944.6	0.006	0.0	7048.0	31.0	141.0	6876.0
1050.0	1050.0	1046.5	0.003	0.0	7111.0	47.0	141.0	6923.0
1150.0	1150.0	1148.4	0.001	0.0	7048.0	62.0	141.0	6845.0
Averages:			**0.015**	**80.3**	**7155.8**	**48.4**	**143.9**	**6963.5**

Table A.27: Cyclic Nonlinear Deterministic Results: R^2 = 0.75, Samples = 32, Iterations = 25

Goal Value	**Math Program Output**	**Simulation Output**	**MP vs Sim Deviation**	**MP vs Goal Deviation**	**Total Time (ms)**	**Solve Time (ms)**	**Model Prep Time (ms)**	**Experiment Time (ms)**
250.0	250.0	273.9	0.096	0.0	8235.0	31.0	140.0	8064.0
350.0	350.0	380.0	0.086	0.0	8251.0	31.0	157.0	8063.0
450.0	450.0	480.4	0.068	0.0	8392.0	47.0	141.0	8240.0
550.0	550.0	575.6	0.046	0.0	8267.0	31.0	157.0	8079.0
650.0	650.0	667.1	0.026	0.0	8344.0	46.0	141.0	8157.0
750.0	750.0	757.4	0.010	0.0	8298.0	31.0	140.0	8127.0
850.0	850.0	849.3	0.001	0.0	8219.0	31.0	141.0	8047.0
950.0	950.0	945.1	0.005	0.0	8376.0	31.0	156.0	8189.0
1050.0	1050.0	1046.3	0.004	0.0	8205.0	32.0	156.0	8017.0
1150.0	1150.0	1149.7	0.000	0.0	8329.0	46.0	141.0	8142.0
Averages:			**0.034**	**0.0**	**8291.6**	**35.7**	**147.0**	**8112.5**