A JAVA BASED INTERACTIVE CONTROL DESIGN
AND TUNING PLATFORM

AARON RADKE

Bachelor of Science in Electrical Engineering
Cleveland State University
May, 2002

Bachelor of Arts in Physics
Cleveland State University
May, 2002

submitted in partial fulfillment of the requirements for the degree

MASTERS OF ENGINEERING

at the

CLEVELAND STATE UNIVERSITY

July 2003
This thesis has been approved for the
Department of **ELECTRICAL AND COMPUTER ENGINEERING**
and the College of Graduate Studies by

Thesis Committee Chairperson, Dr. Zhiqiang Gao

___________________________
Department/Date

___________________________
Dr. Yongjian Fu

___________________________
Department/Date

___________________________
Dr. Dan Simon

___________________________
Department/Date
To my parents...
First, I would like to express my deepest appreciation to my advisor, Dr. Zhiqiang Gao, for his supervision and help throughout the course of my study. Most of all, I am grateful for his vision and encouragement to continue in higher education and research. I enjoy learning from his teaching philosophy and problem solving methods. I am also grateful for his flexibility in the area and topics that interest me for research.

Jack Zeller has also been a great source of encouragement as an advisor with sincere enthusiasm over all that I do.

I appreciate Dr. Dan Simon, Dr. Yongjian Fu, who are on my committee, for their time in reading and evaluating this thesis.

I would also like to thank my fellow colleagues and lab members: Rob Miklosovic, Weiwen Wang, Shahid Parvez, Bosheng Sun, Zhan Ping, Arthur Stachowicz, Greg Tollis, Ivan Jurcic, Chunming Yang, Tong Ren, Frank Goeforth, Rachel Lim for their guidance, help and encouragement in the research, writing and logistics. Adrian Fox and Jan Bosch have also been a wonderful help by the great work done behind the scenes, as well as the rest of the electrical engineering department for the endless support given to me while at Cleveland State University.

I would also like to give a special thanks to Tina Miklosovic for her thorough review, corrections, and suggestions.

Finally, I cordially thank my family for their continuous support, encouragement, and heartfelt joy in all that I do.
A JAVA BASED INTERACTIVE CONTROL DESIGN
AND TUNING PLATFORM

AARON RADKE

ABSTRACT

This research focuses on building an interactive graphical tuning application framework for control systems. The objective is to create an environment for controller design that simplifies the repetitive tuning of multiple parameters, allowing rapid insight into the effects of the parameters on control system performance. This is accomplished by creating a simulation package for controller tuning written in Java, offering a cross platform and web-based solution. The tool kit has several advantages over existing conventional simulation packages. Primarily, it provides the ability to interactively vary parameters for optimal performance. It gives control engineers a tool to help explain and explore conventional and advanced control designs and algorithms. The tool kit also offers an alternative to Matlab for practicing engineers from the design phase to practical implementation.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iv</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>x</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>xi</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Project Focus</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Motivation of Research</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Existing Software Tools</td>
<td>2</td>
</tr>
<tr>
<td>1.3.1 Matlab</td>
<td>3</td>
</tr>
<tr>
<td>1.3.2 SysQuake</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Proposed Approach</td>
<td>6</td>
</tr>
<tr>
<td>1.4.1 Problem Formulation</td>
<td>6</td>
</tr>
<tr>
<td>1.4.2 Software Development Strategy</td>
<td>7</td>
</tr>
<tr>
<td>1.5 Software Requirements and Specifications</td>
<td>8</td>
</tr>
<tr>
<td>1.6 Thesis Organization</td>
<td>9</td>
</tr>
<tr>
<td>PART I SOFTWARE STRUCTURE</td>
<td>10</td>
</tr>
<tr>
<td>II. STRUCTURAL IMPLEMENTATION</td>
<td>11</td>
</tr>
<tr>
<td>2.1 Focus on Difference Equations</td>
<td>12</td>
</tr>
<tr>
<td>2.2 Fundamental Building Blocks</td>
<td>12</td>
</tr>
<tr>
<td>2.2.1 Object Orientedness</td>
<td>12</td>
</tr>
<tr>
<td>2.2.2 Class Inheritance Structure</td>
<td>16</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Example hierarchy path of a PID controller</td>
<td>16</td>
</tr>
<tr>
<td>II</td>
<td>Class list and description for the ‘simtk’ package</td>
<td>95</td>
</tr>
<tr>
<td>III</td>
<td>Class list and description for the ‘simtk.applets’ package</td>
<td>100</td>
</tr>
<tr>
<td>IV</td>
<td>Class list and description for the ‘simtk.variable’ package</td>
<td>101</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SysQuake window</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>The class structure for the blocks (without implementation classes)</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>The entire hierarchal class structure of the project</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>Screen shot for the sample simtk applet GUI</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>Simplified example containment of classes for the GeneralControlLoop applet</td>
<td>22</td>
</tr>
<tr>
<td>6</td>
<td>Comparison of ode solvers with 5 iterations</td>
<td>29</td>
</tr>
<tr>
<td>7</td>
<td>Comparison of ode solvers with 10 iterations</td>
<td>29</td>
</tr>
<tr>
<td>8</td>
<td>Comparison of ode solvers with 20 iterations</td>
<td>30</td>
</tr>
<tr>
<td>9</td>
<td>Comparison of ode solvers with 200 iterations</td>
<td>30</td>
</tr>
<tr>
<td>10</td>
<td>Comparison of ode solvers with 3 iterations and Runge-Kutta</td>
<td>31</td>
</tr>
<tr>
<td>11</td>
<td>2nd order approximate derivative with tau=1,0.5,0.25 on a step function</td>
<td>44</td>
</tr>
<tr>
<td>12</td>
<td>ADRC block diagram controller depicting the separation into two transfer</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>functions</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Trapezoidal source for profiles</td>
<td>52</td>
</tr>
<tr>
<td>14</td>
<td>Profile generated from the integration of a trapezoidal input</td>
<td>52</td>
</tr>
<tr>
<td>15</td>
<td>Triangular source for S-curve profiles</td>
<td>53</td>
</tr>
<tr>
<td>16</td>
<td>Profile generated from a step and 2nd order input</td>
<td>55</td>
</tr>
<tr>
<td>17</td>
<td>Profile generated from the simple polynomial with $T_r = 5$</td>
<td>56</td>
</tr>
<tr>
<td>18</td>
<td>1st order low-pass Butterworth filter filtering noise</td>
<td>58</td>
</tr>
</tbody>
</table>
1.1 Project Focus

This research focuses on building a real time, cross-platform, graphical tuning application framework for control systems. The objective is to create an interactive environment for controller design and tuning that reduces the level of difficulty in repetitive tuning with multiple parameters, and allows rapid insight into the effects of the parameters on control system performance.

This chapter describes the motive for this research by looking into the history of the search for a software package for simple controller tuning. It shows that there are a few major simulation packages used by engineers, but they each have shortcomings in light of the simple tuning goal. It further describes a new direction taken to provide functionality for interactively varying parameters, allow for cross-platform use, as well as provide a web accessible solution.
1.2 Motivation of Research

The control theory problem is an age-old discipline confronting issues in mechanical, electrical, chemical and many other fields of engineering. Over the years, many simple and advanced control algorithms have also been fabricated. Despite the wide range of controllers, most are implemented with a simple PID control algorithm [7]. The main reason for this is the practicality in tuning the controller. Other designs may be more advanced methods than PID, but they are significantly more difficult to tune, which explains why it is almost the only controller used in practice.

A primary focus of this research was to devise a method to assist in tuning advanced controllers to make them practical for implementation. Recently, a number of algorithms, explained in [10] and [11], and tuning methods, in [8], have been developed along these lines. The goal of this research is to use these algorithms and tuning methods along with the unique design around dynamically adjustable variables for rapid controller tuning.

1.3 Existing Software Tools

There are many software packages to assist engineering designs, but there are only a few major packages specifically focused toward simulating and designing control systems. The most popular package is Matlab\textsuperscript{1} because its major focus is on control theory. In a few cases, general mathematics packages such as Mathematica\textsuperscript{2} or Maple\textsuperscript{3} are used. Another interesting package that is focused around control and the dynamic tuning focus of this project is SysQuake\textsuperscript{4}. However, none of the these packages fit all the goals of this research.

\textsuperscript{1}Matlab is a trademark of MathWorks, Inc.
\textsuperscript{2}Mathematica is a trademark of Wolfram Research Inc.
\textsuperscript{3}Maple is a trademark of Waterloo Maple Inc.
\textsuperscript{4}SysQuake is a registered trademark of Calerga.
1.3.1 Matlab

Matlab is the leading graphical simulation package for controls for a number of reasons. It is more or less an industry standard. In almost every undergraduate controls class Matlab is used from the beginning. In industry, it is also the most used package for designing control systems, including electrical, chemical and mechanical systems.

Matlab also gained widespread use with the inclusion of Simulink to offer a graphical environment to design systems. Matlab had traditionally been interfaced by code in a programming environment. Simulink is the graphical connection of Simulink blocks which allows easy modeling of nonlinear systems. In the past the entire system would have to be programmed, compiled, and run. Simulink hides it all behind a Graphical User Interface (GUI).

Despite the widespread use of Matlab and the simplicity of Simulink they have a few characteristics that do not match with the goals of this project.

Working with Simulink on a controls problem often proves to be tedious because of the multiple-step process that is required to view the results of adjusting a single parameters. First, the parameter in an m-file or dialog box needs to be physically adjusted, then the changes should be saved, the m-file executed and then finally the Simulink design needs to be re-compiled to get the results. Reducing the work in this process was the initial goal of the research.

Matlab has the provision for real time drag-able sliders while the results are shown in a plot, eliminating the problem of tedious tuning. However, the GUI programming environment for Matlab must be learned. Any subsequent control designs must then be built within this environment.

The cost to continue using the software, even the continuing license fees for education, is increasing. The license issue also prevents those without education or
corporate copies to use the tuning software.

The recent version updates in Matlab contribute to the final issue. Some s-functions\(^6\) will not work in the newer versions. Because of this, a control application that was designed using one version of Matlab may not work with a different version.

1.3.2 SysQuake

SysQuake from Calerga is a relatively new package focusing on real time tuning. It came with a sample program showing some of the capabilities, including a real time, graphical tuning control of a PID controller.

Sysquake has a number of nice factors. First of all, the light edition, which performs the same major functions provided by Matlab, is free. SysQuake is also similar to Matlab’s m-file syntax. It uses an engine called LyME, standing for Lightweight Math Engine, which contains most of the same functions as Matlab, including some of the control libraries for step, Bode, and Nyquist response plots.

The most interesting and powerful part of SysQuake is that it was entirely designed around the idea of watching the outputs change instantaneously as the tuning parameters are varied. The icon for the program even has a finger manipulating a graph. The provided sample programs also demonstrate the powerful aspect of being able to tune control designs in real time and reduce the complexity of tuning.

Because of the real time nature of SysQuake, it was initially chosen as the development environment for this research. The project began by turning the SysQuake PID example into a Nonlinear PID\(^6\) (NPID) program to help tune the nonlinear controller in real-time. The problem was that the PID example was linear and the NPID was nonlinear. A new method of representing and solving the differential equations

\(^5\)an s-function is Matlab’s definition of a block that can be programmed and used in a Simulink diagram

\(^6\)see section Section 5 for more information on the description of the algorithm
needed to devised. So the entire program was re-written using an ode45 solver rather than using purely linear solvable transfer functions.

In the NPID Sysquake application the nonlinear regions could be dragged around with the mouse in any direction, and seen in real-time how it affects the output response. The screen shot of this first application is shown in Figure 1. This was a very expandable package which could be manipulated in many ways. Each figure can be selected for different options and different tools to vary parameters such as controller parameters, disturbance input, and all the nonlinear regions.

![Figure 1: SysQuake window](image)

Although SysQuake was very useful and proved to be an interesting effective solution, there are still some characteristics that held it back from full-fledged use for the tuning and simulation goals. The main problem with using SysQuake occurs in modeling the system. It works great for a set system plant or control setup, but if a new system or plant is desired, an entirely new differential equation must be derived.
for the ode45 solver. This makes it difficult to set up and modify. When a plant is given as a transfer function, it must be converted to a differential equation in the time domain. This process, required by even the end user, increases the complexity of creating new plants or controllers to test within SysQuake.

1.4 Proposed Approach

The struggles existing with the underlying method of solving the differential equations with either predefined transfer functions or differential equations led to the decision to start from scratch and choose a programming language.

1.4.1 Problem Formulation

To start from the ground up, a method was required that was as easy to set up as the s domain systems but also had the functionality of nonlinear functions. The solution was to create the entire system with the differential equation solvers in discrete time with difference equations. Solvers can now be created, straight from the linear s domain transfer function as well as any other nonlinear function.

Another advantage of the discrete solvers is that the simulation can be designed to match the actual implementation that is built in hardware. This is helpful for more predictable results of a controller’s performance in practice. This form of implementation assures the closest synthesis of a system in hardware, which is beneficial for two reasons. First, many of the problems that arise in a hardware implementation could be caught in the simulation phase before the expense of hardware implementation is incurred. Secondly, the transition from simulation to implementation is trivial.
1.4.2 Software Development Strategy

Development Language

Since the entire project would be completely rewritten at the difference equation level, the software package or environment is unimportant. The same work would inevitably need to be programmed directly in some general high-level language instead of the specific high-level languages of Matlab or SysQuake. The Java programming language was chosen as the platform to build the simulation package. There are several major benefits of using Java for the platform.

First of all, Java is available at no cost. There are no license fees and no restrictions. Both the runtime environment and the software development kit are available to anyone and can be downloaded from the Internet on Sun’s web-site http://java.sun.com. Java was also originally built to be run on any machine, whether it be Windows, Unix, or MacOS. Virtual machines run on a computer so the same machine code will run on any machine. Therefore, any computer in the world with a web browser can run this package.

Java has a rich framework of libraries and classes available to help build any program. There are also numerous third-party packages that can easily be incorporated into a design. For example, there is a good plotting package called PtPlot from the Ptolemy project which was used as the main plotting framework in the ‘simtk’ library.

Structure

As previously described, the focus was based around difference equations as the means of the describing and simulating engine. Higher level functionality, such as s domain transfer functions, are achieved by laying interfaces over the lower level difference equations. A structure of hierarchal blocks are used as a means of abstraction.
and a means to build up the library.

1.5 Software Requirements and Specifications

There were several major features that the software library designed in this research was meant to fulfill. The items below are the major areas and should fill in the gaps left unavailable in existing packages which were discussed in the previous sections.

The primary requirement of the software, which is the original goal of the project, is to encompass an interactive environment for controller design and tuning that reduces the difficulty of repetitive tuning with multiple parameters, and allows rapid insight into the effects of the parameters in the control system.

It is also important that the design is an extensible framework which is open for additions, modifications and expansions. A method for building up the library with future new algorithms, control methodologies, and user interaction is required to be in place.

An important means for any controls simulation package would be the ability to simply construct, model and simulate nonlinear systems. Both linear mathematical definitions and nonlinear structures need to be easily connected and simulated seamlessly with respect to the user.

Another goal of the software is to be available to many environments. A cross-platform focus allows the package to be available to a larger scope of users. In the same way, making it available through a web-site will open the doors to many who do not have the resources of traditional simulation packages.

Since the package is available on the Internet a useful purpose of the library framework is made as demonstration of new control algorithms. From the Center for Advanced Control Technologies (CACT), a number of new control algorithms have
been presented over the past few years. Putting the algorithms on the Internet allows anyone in the world to test their plant with our controllers.

As always in an engineering design, the product should be useful and practical for implementation. The simulator itself should be useful but it should also help design practical controllers. The provision to automatically generate the tuned programmable difference equations from the plant or controller would assist in implementation.

These are a number of initial major requirements for the software. Chapter 8 presents a summary of the features of the implemented library as well a list of future functionalities that would enhance the software.

1.6 Thesis Organization

This document is split into two parts. The first focuses on the software, and the second focuses on the system functionality and application.

The first few chapters, from Chapter 2 to Chapter 3, discuss the software implementation. Both the structural implementation and the data flow are described. The second part follows from Chapter 4 through Chapter 7 breaks down the system functionalities and applications and covers such topics as the equation solvers and control algorithms. These two major parts are followed by future research in the final chapter.
PART I

SOFTWARE STRUCTURE
CHAPTER II

STRUCTURAL IMPLEMENTATION

There are several important aspects to the software structure. The core engine of the simulation is the difference equation solver. Fundamental building blocks are constructed to each use this engine. A means of connecting the blocks together is also fabricated in order to build or model a system. Behind the package is a rich, extensible, object-oriented library which includes many common control system blocks, as well as some more advanced control designs. In addition, there is a unique graphical user interface which takes advantage of the underlying library for real time tuning.
2.1 Focus on Difference Equations

With the move to start building the simulation package from scratch, the core
design of the package depended strongly on the differential equation solver. The first
step was to investigate the properties and methods of using and generating the discrete
difference equations. This involved looking into simple fundamental functions such
as integrators and differentiators. Various solver methods, such as Euler and Tustin,
which are discussed further in Chapter 4 were investigated and tested. Perl and
Gnuplot, two open source programs, were used to experiment and generate these
core methods for difference equations. After the core was built, every other function
was built off of this basic core by extension. After the initial framework for the
equation solvers was set up and working, a simple test GUI was built around it for
demonstration purposes.

2.2 Fundamental Building Blocks

Fundamental building blocks were built as “components of simulation.” A
rich library, composed of many basic and advanced control systems blocks, was con-
structed. The simulation components could then be selected from the library and
connected together to form desired systems.

2.2.1 Object Orientedness

The object-oriented (OO) nature of Java forced the problem to be broken down
into types and sections. It enforces structured thought; and interesting solutions to
problems are formed. A primary drive of the OO design is to bring the problem space
into the code space so the problem can be directly represented in code in terms of
the original problem. This was helpful in this project to create a hierarchal library
of simulation components.

**Library of Blocks**

The OO breakdown created a library of hierarchal related blocks with different categories and subsets. For example, one source definition could be used for both step sources and noise sources. A root class which included the most basic fundamental block requirements was defined from which the rest of the blocks would be built.

SimBlocks became the basic building blocks of the simtk project. The full documentation and tree structure of these classes can be found on the simtk website[^1] listed at the end of this document. There are three major types of blocks: SimSource, SimFunction and SimSink. A combination of these blocks can be used to create any system. The flow and method of combining these blocks is described fully in Chapter 3. SimSources are blocks with a single output such as step responses, disturbances and profiles. SimFunctions are blocks which have multiple inputs and a single output[^2] SimSinks are blocks with multiple inputs, and no outputs. Some examples of SimSinks are plots, standard output, and text outputs. Tables listing all of the classes and the descriptions are given in Appendix B.

The OO structure hierarchy of the blocks and classes is easier to understand in a graphical manner. The following diagrams were generated by the java2dot program[^3]. From the graphical diagram shown in Figure 2 it can be seen that every SimBlock is an extension of the three main building blocks. The class structure for the blocks is graphed without the implementation classes. The entire hierarchal class structure is displayed in Figure 3. Additional figures are displayed in Appendix D.

[^1]: http://academic.csuohio.edu
[^2]: In the future multiple outputs could also be implemented to provide for state space representation of systems.
[^3]: see Appendix B.3 for information about the java2dot program
Figure 2: The class structure for the blocks (without implementation classes)
Figure 3: The entire hierarchal class structure of the project
2.2.2 Class Inheritance Structure

An interesting example explanation of the hierarchical path is the following:

\[
\text{SimPid} \rightarrow \text{SimStf} \rightarrow \text{SimZtf} \\
\rightarrow \text{SimDiffEq} \rightarrow \text{SimFunction} \\
\rightarrow \text{SimBlock} \rightarrow \text{SimBase} \tag{2.1}
\]

This means that SimPid is a SimStf, and a SimStf is a SimZtf and so on. Each implementation has all of the features of the inherited classes. The sorted ascending list of increasing levels of abstraction and what feature each inherited block provides is shown in Table I.

<table>
<thead>
<tr>
<th>Class name</th>
<th>provides</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimBase</td>
<td>simple title and description structures</td>
</tr>
<tr>
<td>SimBlock</td>
<td>the ability to interface with the rest of the simulation library</td>
</tr>
<tr>
<td>SimFunction</td>
<td>the functionality of inputs and outputs</td>
</tr>
<tr>
<td>SimDiffEq</td>
<td>the ability to create, display, and edit discrete difference equations</td>
</tr>
<tr>
<td>SimZtf</td>
<td>the ability to create, display, and edit z transfer functions and convert them to discrete difference equations</td>
</tr>
<tr>
<td>SimStf</td>
<td>the ability to create, display, and edit s transfer functions and convert them to z transfer functions</td>
</tr>
<tr>
<td>SimPid</td>
<td>the specific transfer function for proportional, integral and derivative control</td>
</tr>
</tbody>
</table>

2.2.3 Block Interconnection

A key component of the structural implementation was the means of connecting each of the desired blocks together. In order to reach the speed required for real
time dynamic response, practicality and efficiency were considered when designing the interconnection buffers. Considering the fact that the amount of data history required for any block is defined by the order of the difference equation, and that the order is most often less than ten for most systems, the memory used for the history can be kept minimal. The reduced size buffers are efficient because extra memory is not required, and they are practical because the same implementation scheme can be used in an embedded system.

The SimBuffer was created to store the recent history of each block. This buffer was also used as the software implementation of a connection between blocks to create the desired signal. The connections are made with a supervisory class called SimFactory. More information on the implementation of the connection and flow of data of the classes is in Chapter 3.

2.3 GUI Development

Once the functional structure was well-defined, simple GUI elements were added to give a visual demonstration of the library.

The GUI was built with Java’s newer GUI Application Programmers Interface (API) called Swing. Just as the object-oriented nature of the design appropriately constructed the hierarchal nature of the library, it also matched the implementation of the GUI. Each of the SimBlocks that are connected can generate their own adjustment dialogs, as well as other windows like plots. A simple block would display its own information while the detail of a more complex block is automatically generated from the separate building components from which it is extended. An example of the simtk applet which incorporates many levels of the hierarchal GUI can be seen in Figure 4 and Figure 40. The use of this applet is more fully described in Appendix C.
Figure 4: Screen shot for the sample simtk applet GUI
CHAPTER III

DATA FLOW IMPLEMENTATION

The structural inheritance implementation of the relationship between blocks can easily be seen and was described in the previous chapter. However, in order to understand how the classes all work together, the interaction and data flow organization between the simtk library blocks needs to be described. Both developers (who would add on to the library and use it for specific purposes) and users of a constructed simulation package should understand the fundamental flow of the data. Developers should build blocks consistent to the package as well as understand how a system is designed.
3.1 Developer’s Viewpoint

In the same way that a person can drive a car without knowing exactly how a gasoline engine works, it is not necessary for a user to understand the underlying functions of the software library. However, a more in-depth knowledge is important for a designer who would want to add or modify a car’s features and functionality. In the same way, it is important for a software developer, to understand the underworkings of the code to design and build additional features to the system.

3.1.1 Top-level Managing Class

A software developer would primarily be interested in how pieces of the system are connected. The previous chapter described that there were many simulation blocks created. There is a need for a top-level managing class that will control, manage, and act as a base to connect each of the simulation pieces together. This class is called the SimFactory, which is used as a means to store the global-type variables, such as simulation parameters, for the final time and step size that are used for every block. Furthermore, with the simulation information the factory class knows how many iterations to perform on the blocks to reach the desired simulation period. It performs the iterations of the blocks and interconnection buffers at the appropriate times and steps through the system. Presently the order that the blocks are added to the factory class is the same order that the blocks are iterated. There is no algorithm for simulation order discovery\footnote{Simulation order discovery is a future area of research (see Section 8) to add to the simulation package which would be necessary for a graphical environment similar to Simulink.} Therefore, some knowledge is needed to correctly order and connect the blocks prior to simulation. This is especially true for some controllers that need the plant information before the controller is calculated. Presently, the top-level class is the sand-box where the library components are placed.
and held together.

The overseeing class is also useful when built in a hierarchical manner. Designing it to be included within itself allows an abstraction allowing complex simulation blocks to appear as any other simple functions, only with more internal functionality. The SimFunctionFactory class is an example of a SimBlock that externally looks like an ordinary SimFunction. However, internally the complexity can be built by combining other blocks. For the hierarchy abstraction, the SimFactory is wrapped, for example, by an applet that implements the HasFactory interface. One such applet is the GeneralControlLoop.java applet, listed in Appendix E on page 103 which contains a SimFactory and shows the basic requirements to set, configure, and make use of the library.

It is also helpful to see a containment graph of classes to get a feel of the relation and use between various classes. The following diagram was generated by the containment.pl program\textsuperscript{2} It simply illustrates how the SimFactory makes use of the other blocks and also requires a few of the global parameters which it passes to the rest of the simulation. A simplified example, of the containment of classes for the GeneralControlLoop applet, with some classes removed, is illustrated in Figure 5.

\textsuperscript{2}see Appendix A.3 on page 89 for information about the containment.pl program which uses the graphviz dot language format to generate directed graphs
Figure 5: Simplified example containment of classes for the GeneralControlLoop applet
3.1.2 Setting Up a Simulation

There are only a few major steps to setup a desired simulation. The three major sections are loading, connecting and simulating. The first step in setting up a system is to define what is needed. Classes from the simtk library can be instantiated statically by code or dynamically by using one of the SimSelectors to load a desired function at runtime. After the blocks are loaded into memory, they need to be added to the simulation factory to be simulated. While adding the blocks to the factory the blocks can also be connected. The SimFactory.add() function also takes the block that should be connected to the input of the block currently being added. Finally, when all the required blocks are connected, the SimFactory.doSimulation() can be called. See Listing E.1 on page 117 for a single example of using and creating a control system with the library.

3.1.3 Building Blocks

There are also three major steps in creating or building more functions to the library: extend a block, add the adjustable variables, and override the iteration definition.

First of all, the choice of a block needs to be made that already has the functions that implements similar operations and the new functionality can simply be added. Most extended blocks are built from SimSource, SimSink, SimFunction or a derivative of these such as or SimZtf, SimStf.

Once the parent class is selected, the new functionality is added by selecting the types of variables that are needed from the simtk.variables package. These are

---

3 As an example the add function could be: add([the block to be added],[title of the block], [output of the block to connect to this one])

4 Within these variables the GUI is developed, each block knows its own variables and each variable knows how to display its own GUI a few of the variables are strings(TextFields), doubles(Sliders and textFields), double arrays(comma separated text lists), and boolean (checkbox).
then added to the VariableContainer where the other variables are also stored.

The final step to define a block and where the actual functionality is stored is within the iteration definition. The function doIteration() is called whenever a parameter is updated. For example, if a variable is changed with the slider this function will make the appropriate changes. This is where the code to calculate the new outputs or parameters from the current inputs or variables is calculated. For some classes such as SimStf, the doIteration() is constant but the parameters are redefined. In this case simply overriding the reDefine() function will load the desired transfer function at a parameter update. See Listing E.2 on page 119 for a single example of extending an S transfer function into a general second order plant.

The developer interested in modifying the code and adding to the library should first understand the bases and underlying theory of the package from the information in these and the following chapters. For reference of the code structure and how to use the library, the example code for the applets can be followed. The main source and documentation of the API is contained in the javadoc-generated documentation for the project.

3.2 User’s Viewpoint

While the knowledge of the underlying structure is not required for the user it is still necessary to understand how the system is held together. There are a few main ideas that the user should grasp.

First of all, it is imperative that the nature of the system is understood. The results of the simulation are meaningless unless the user understands what they are simulating. Although this may sound trivial, it is important to achieve appropriate results. A user must enter meaningful inputs to achieve meaningful outputs.

A second, corresponding, comprehension required from the user is that the
simulation accuracy is dependent upon the specified step size. To make the real time effect transparent the simulation size is kept small. However, the step size for the simulation may need to be reduced by the user to achieve more accurate results.

Finally, along with the step size, there are a few global parameters that make adjustments across the whole simulation. A few of the current global parameters are the final simulation time and the step size. The step size is an important criteria for some discrete controllers\(^5\) which require a knowledge of the step size for calculation of the control signal.

While these fundamental ideas hold to the general nature of the simtk library, additional information should be given by the developer to the user for specific features. As an example, a small user’s guide of the present demonstration applet is provided in Appendix C on page 103.

\(^5\)One example is the Discrete Time Optimal Control. See Chapter 5 for more information.
PART II

SYSTEM FUNCTIONALITIES
AND APPLICATION
CHAPTER IV

DIFFERENTIAL EQUATION SOLVER

This chapter and the following chapters will go further into the depths of the core mathematics for the simulation package. The most important mathematical function for a simulation package is how the system is expressed, modeled and solved. Matlab contains a broad selection of differential equation solvers, but without the help of Simulink, constructing nonlinear systems is difficult. SysQuake provides an ode45\(^1\) solver, but expressing a system in this form restricts the model to time domain differential equations. Although this may be convenient for the solver, it is not for the user. Often it is helpful to express systems in terms of other variables such as frequency or even more abstract references to help model nonlinear systems. For the speed requirements of the real time simulation and tuning, basic difference equations for discrete time solutions have been chosen as the basic simulation functions while further functionality is overlaid.

A comparison is made, in Section 4.1\(^2\), between several differential equation methods and the reasons that the Tustin method was chosen over the Simple Euler

\(^{1}\)4th Runge Kutta algorithm
method is given. Using the Tustin method, algorithms are developed in Section 4.2 to convert various system expressions into a difference equations.

4.1 Comparison of Ode Solvers

There are several types of methods for computerized differential equation solutions. Simple Euler and Tustin are the two major methods which receive the most attention when being compared for this project. The Simple Euler is one of the simplest and most often used in implementation of embedded systems while Tustin is slightly more involved.

The ordinary differential equation solver algorithms can become confusing since there are many names for the same algorithm. Different fields have different names. The primary solver that is used in the simtk library is called Tustin by the control theorists, Trapezoidal by Mathematicians, and Bilinear by computers scientists. Since this research centers around control theory, Tustin will be used to denote this type of solver.

4.1.1 Simple Example Problem for Comparisons

A simple definition of a differential equation of a system is defined in (4.1). Various solvers are then applied to the same function and compared to each other and the exact solution.

\[
y'(t) = f(t, y(t)) \quad (4.1)
\]
\[
y(t_0) = y_0 \quad (4.2)
\]

The following comparisons plots were generated from a Java applet at the University of British Columbia to compare various differential equation solvers for the same
Figure 6 and Figure 7 demonstrate that the Tustin algorithm has a more accurate solution to the actual solution than the Simple Euler method, however with only these graphs it is difficult to get a picture of how much better it behaves.

The following two graphs give an understanding of the benefits of using the slightly more advanced algorithm.\textsuperscript{3}

\textsuperscript{2}This Applet can be found at: \url{http://www.math.ubc.ca/~feldman/demos/demo2.html}

\textsuperscript{3}The level of sophistication of the Tustin and Euler algorithms are derived in Chapter 4.2
By increasing the number of iterations to 20, as shown in Figure 8, the Tustin algorithm is shown to reach quite close, at least at this scale, to the exact solution. In order to reach this same level of accuracy with the Simple Euler method the number of iterations is increased to 200 in Figure 9. Almost 10 times the samples needed for the Tustin solver which is only slightly more involved. This reduction in the need of sampling time is directly related to the real world problem of sampling time and the cost of faster analog to digital converters.

The final comparison in Figure 10 shows the advantage of the Runge-Kutta algorithm.
Although it is significantly better than the Simple Euler and even the Tustin, especially for smaller sampling times, for now\footnote{In the future it would be an interesting area of research to test algorithms at extremely low sampling rates and incorporate the Runge-Kutta solver} the complexity of the implementation of the algorithm is not worth the accuracy gained over the simplicity of the Tustin algorithm at higher sampling rates. The Runge-Kutta algorithm is significantly more involved and not suited for the application of real time control especially for exportation of algorithms to embedded systems.

The Tustin algorithm is significantly more accurate than the Simple Euler method with only a slightly more involved implementation. The primary benefit is an almost ten-fold decrease in the sample rate or step-size for the same level of accuracy, which is the reason it was chosen as the primary solution for the engine of this project.

\section{4.2 Differential Equation Solver Engine}

The core of the simulation package is the differential equation solver engine. It is important to understand its purpose and specifically describe the implementation. At first, many specific discrete equation solutions were derived from commonly used
continuous functions. For an extensible library, multiple levels of the solver needed to be designed separately in a modular form. Instead of a static specific transformation, a fundamental general $s$ to $z$ transform algorithm was then designed around the Tustin method. This was converted directly to the discrete difference equation with a general conversion from $z$ transform functions. These various levels of conversions are overlaid transparently and executed in succession to produce the illusion of various types of model descriptions in a single simulation environment.

4.2.1 Specific Conversions from Continuous Form to Discrete

The following sections derive the discrete solutions as $z$ transform functions, for specific blocks such as integrators, second and first order responses, and PID. The derivation is shown for the Simple Euler and the Tustin method to show the difference in implementation complexity. All of the difference equations were ultimately solved using Tustin’s method, which is simple but still has a significant performance enhancement over the simple Euler method.

Simple Euler

The Simple Euler method used for the conversions of the $s$ to $z$ domain is defined by substituting $s$ with the following equality. Where $s$ is the Laplace transform of a pure differentiator,

$$ s = \frac{-1 + z}{T} \quad (4.3) $$

The simple integrator becomes

$$ \frac{1}{s} \rightarrow \frac{T}{-1 + z} \quad (4.4) $$

the performance enhancement can be shown with the Java applet comparing the Simple Euler, trapezoidal and Runge-Kutta methods in Section 4.1.
A double integrator after some reduction then becomes

\[
\frac{1}{s^2} \rightarrow \frac{T^2}{1 - 2z + z^2} \tag{4.5}
\]

These simple equations demonstrate that the Simple Euler method is quite simple to implement, however it does not provide a very efficient or accurate conversion when compared with some other methods as described in Section [4.1].

**Tustin**

The Tustin method proves to be far more accurate and efficient than the Simple Euler but only slightly more involved when doing the conversions of the \( s \) to \( z \) domain [2]. This method is defined by substituting \( s \) in a Laplace transform with the following equality.

\[
s = \frac{2 \left(-1 + z\right)}{T \left(1 + z\right)} \tag{4.6}
\]

The simple integrator then becomes

\[
\frac{1}{s} \rightarrow \frac{T + Tz}{-2 + 2z} \tag{4.7}
\]

A double integrator after some reduction then becomes

\[
\frac{1}{s^2} \rightarrow \frac{T^2 + 2T^2 z + T^2 z^2}{4 - 8z + 4z^2} \tag{4.8}
\]

A first order plant, where \( a \) is the pole location becomes

\[
\frac{1}{s + a} \rightarrow \frac{T + Tz}{-2 + aT + 2z + aTz} \tag{4.9}
\]

A second order plant, where \( \omega \) is the frequency and \( \zeta \) is the damping ratio is given as
\[ \frac{\omega^2}{s^2 + 2\zeta\omega + \omega^2} \rightarrow \frac{T^2 \omega^2 + 2T^2 z \omega^2 + T^2 z^2 \omega^2}{4 - 4T \zeta \omega + T^2 \omega^2 + 2z (-4 + T^2 \omega^2) + z^2 (4 + 4T \zeta \omega + T^2 \omega^2)} \] (4.10)

A second order plant where \( \omega = \zeta = 1 \) is shown to be slightly simpler

\[ \frac{1}{s^2 + 2 + \omega^2} \rightarrow \frac{T^2 + 2T^2 z + T^2 z^2}{4 - 4T + T^2 - 8z + 2T^2 z + 4z^2 + 4Tz^2 + T^2 z^2} \] (4.11)

A PID controller where the proportional, integral, and derivative gain are defined as \( k_p, k_i, \) and \( k_d \) respectively is derived as

\[ \frac{k_ps + k_ds^2 + k_i}{s} \rightarrow \frac{4k_d - 2k_p T + k_i T^2 - 8k_d z + 2k_i T^2 z + 4k_d z^2 + 2k_p T z^2 + k_i T^2 z^2}{-2T + 2T z^2} \] (4.12)

A pure differentiator can not be implemented since it results in an improper transfer function, but approximate results can be obtained. The second order approximate differentiator has been proven to give excellent differentiation, in [12], with relatively simple implementation. The derived \( z \) transform function is given as

\[ \frac{s}{(\tau s + 1)^2} \rightarrow \frac{-2T + 2T z^2}{T^2 + 2T^2 z + T^2 z^2 - 4T \tau + 4T z^2 \tau + 4 \tau^2 - 8z \tau^2 + 4z^2 \tau^2} \] (4.13)

**General Transfer Function Derivations**

The specific solutions show relatively simple difference equations to implement such functions as second-order plants and PID controllers. A rather flat solution for a simulation package could simply have a large library of these solutions for every
desired model. However, to create an extensible, powerful, dynamic package rather than a limited set of static predefined blocks, general describing solutions are solved. The first five orders of a Laplace polynomial are shown below.

The general 0th order polynomial

\[ Gp = n_0 \]  \hspace{1cm} (4.14)

transforms to

\[ n_0. \]  \hspace{1cm} (4.15)

The general 1st order

\[ Gp = n_1 s + n_0 \]  \hspace{1cm} (4.16)

transforms to

\[-2n_1 + n_0 T + 2n_1 z + n_0 T z.\]  \hspace{1cm} (4.17)

The general 2nd order

\[ Gp = n_2 s^2 + n_1 s + n_0 \]  \hspace{1cm} (4.18)

transforms to

\[ 4n_2 - 2n_1 T + n_0 T^2 - 8n_2 z \]
\[ +2n_0 T^2 z + 4n_2 z^2 + 2n_1 T z^2 + n_0 T^2 z^2. \]  \hspace{1cm} (4.19)

The general 3rd order

\[ Gp = n_3 s^3 + n_2 s^2 + n_1 s + n_0 \]  \hspace{1cm} (4.20)
transforms to

\[-8 n_3 + 4 n_2 T - 2 n_1 T^2 + n_0 T^3\]
\[+24 n_3 z - 4 n_2 T z - 2 n_1 T^2 z + 3 n_0 T^3 z\]
\[-24 n_3 z^2 - 4 n_2 T z^2 + 2 n_1 T^2 z^2 + 3 n_0 T^3 z^2 + 8 n_3 z^3\]
\[+4 n_2 T z^3 + 2 n_1 T^2 z^3 + n_0 T^3 z^3.\]  \hspace{1cm} (4.21)

The general 4th order

\[Gp = n_4 s^4 + n_3 s^3 + n_2 s^2 + n_1 s + n_0\]  \hspace{1cm} (4.22)

transforms to

\[16 n_4 - 8 n_3 T + 4 n_2 T^2 - 2 n_1 T^3 + n_0 T^4 - 64 n_4 z + 16 n_3 T z\]
\[-4 n_1 T^3 z + 4 n_0 T^4 z + 96 n_4 z^2 - 8 n_2 T^2 z^2 + 6 n_0 T^4 z^2 - 64 n_4 z^3\]
\[-16 n_3 T z^3 + 4 n_1 T^3 z^3 + 4 n_0 T^4 z^3 + 16 n_4 z^4\]
\[+8 n_3 T z^4 + 4 n_2 T^2 z^4 + 2 n_1 T^3 z^4 + n_0 T^4 z^4.\]  \hspace{1cm} (4.23)

### 4.2.2 General $s$ to $z$ Transform Matrices

The previous results can be displayed in a more concise manner in the form of matrices. This is very important for the simplicity of the implementation in software for the solver engine. A general $s$ to $z$ ($s2z$) transform matrix can be derived as follows:

\[Z_n = M_n \cdot S_n \cdot T_n\]  \hspace{1cm} (4.24)
where $n$ is the order of the system and, $s_n$ is the coefficient of $s$ with the exponent of $n$.

$$S_n = \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ \vdots \\ s_n \end{pmatrix}$$ (4.25)

and

$$T_n = \begin{pmatrix} T^0 \\ T^1 \\ T^2 \\ \vdots \\ T^n \end{pmatrix}$$ (4.26)

With the above definitions of $M_n$, the general-ordered Laplace polynomials are summarized in the following matrices.

$$M_0 = \begin{pmatrix} 1 \end{pmatrix}$$ (4.27)

$$M_1 = \begin{pmatrix} -2 & 1 \\ 2 & 1 \end{pmatrix}$$ (4.28)

$$M_2 = \begin{pmatrix} -4 & -2 & 1 \\ -8 & 0 & 2 \\ 4 & 2 & 1 \end{pmatrix}$$ (4.29)
The matrix form is very powerful and efficient for numerical computation. The resulting Java code used to implement conversion printed in Listing 4.1 was directly implemented from (4.24). This shows a very simple level of code to convert a general system.

Listing 4.1: S transform to z transform algorithm

```java
for(int i = 0; i<=order; i++) {
    // loop through z^i terms
    for(int j = 0; j<=order; j++) {
        // continually add up terms for the coef of z^i
        // storing them into zpoly in the form of z^3+z^2+z^1
        zpoly[order - i] += c2dmatrix[i][j]*Tx[j];
    }
}
```

This conversion algorithm is also useful for future implementations of differing solver engines. For example, to implement a different solver simply swap in a new matrix.

4.2.3 General z Transfer Function to Difference Equations

After the z transfer function is produced it can simply be converted directly into a time domain, code-executing solution. The following derivation is a simple
algorithm for automatically converting a general $z$ transfer function into its specific difference equation for simulation or code generation.

Using the general 4th order transfer function a general $z$ transfer function to discrete (z2d) form equation can be derived with input $r$ and output $y$ defined as:

$$\frac{y}{r} = \frac{n_0 z^4 + n_1 z^3 + n_2 z^2 + n_3 z + n_4}{d_0 z^4 + d_1 z^3 + d_2 z^2 + d_3 z^1 + d_4 z^0}$$  \hspace{1cm} (4.32)

The form of this equation can be converted to negative powers of $z$ by multiplying by:

$$z^{-4}$$  \hspace{1cm} (4.33)

Then we can solve for the output, $y$

$$y = \frac{1}{d_0} \left( (n_0 r + n_1 r z^{-1} + n_2 r z^{-2} + n_3 r z^{-3} + n_4 r z^{-4}) ight)$$

$$- (d_1 r z^{-1} + d_2 r z^{-2} + d_3 r z^{-3} + d_4 r z^{-4}))$$  \hspace{1cm} (4.34)

By inspection we can see that we could further reduce this formula to an even more general form of the output for a $q$ ordered system.

$$y = \frac{\sum_{i=1}^{q} (n_i r z^{-i} - d_i r z^{-i})}{d_0} + n_0 r$$  \hspace{1cm} (4.35)

By the inverse $z$ transform we know that

$$z^{-i} \rightarrow y(k - i)$$  \hspace{1cm} (4.36)

Once in this form, we can easily read off the difference equation. The entire process can now easily be implemented in code with an iterative loop for any ordered system.
This derived formula can be implemented in Java code, which is listed in Listing 4.2

Listing 4.2: z transform to difference equation algorithm

```java
double tempsumterms = 0;
for (int i=1;i<=zorder;i++) {
    tempsumterms += num[i]*in.val[i] - den[i]*out.val[i];
}
out.val[0] = (tempsumterms + num[0]*in.val[0])/den[0];
```

\(^7\) this code is found in the SimZtf class within the doIteration function
CHAPTER V

CONTROLLER DESIGN AND IMPLEMENTATION

With the core of the solver implemented in Section 4.2, control modules can now be built. There are a number of important control theory algorithms and methods such as parameterization in [8], Nonlinear-proportional-integral-derivative (NPID) in [13], Active Disturbance Rejection Control (ADRC) in [11], and Discrete Time Optimal Control (DTOC) [10], that are essential to this project’s purpose of simplistic, practical tuning which are implemented in this project and discussed within this section. Many control systems are simple filters and Section 5.6 is devoted to them as well as the important use of pre-filters in control designs.
5.1 Parameterization

Parameterization is a control methodology introduced, in [8], for vastly improving the tuning of a system. It allows for the fundamental understanding of a system and relates many defining parameters into one super-parameter which has a direct correspondence to the system, usually in terms of the speed or bandwidth of the system.

For example, the parameters of a PD controller can be optimally derived for a double integrator plant, $\frac{1}{s^2}$. The closed loop form is put into a critically damped response with a bandwidth of $\omega_c$. To get this type of response $k_d$ and $k_p$ are solved as:

$$k_d = 2\omega_c$$  \hspace{1cm} (5.1)

$$k_p = \omega_c^2$$  \hspace{1cm} (5.2)

Parameterization is used to vastly simplify the advanced controllers, such as ADRC. This, as well as some more familiar controllers are described in the following sections.

5.2 Approximate PID

The proportional-integral-derivative (PID) controller, in [2], is a prevalent controller due to its effectiveness and simplicity. With the inclusion of parameterization, it becomes an even more efficient controller since it can easily be tuned for optimality for a desired bandwidth, instead of using the tedious trial-and-error tuning. The definition of the PID equation can be written as a function of the input error, $e$, and
the output from the PID, \( u \)

\[
\begin{align*}
u &= k_p e + k_i \int e + k_d \dot{e} \\
&= k_p e + k_i \int e + k_d \dot{e} \quad (5.3)
\end{align*}
\]

Differentiating both sides of this equations yields

\[
\dot{u} = k_p \dot{e} + k_i e + k_d \ddot{e} \quad (5.4)
\]

Using the inverse Laplace transform and solving for \( u/e \) gives the transfer function of the PID formula

\[
\frac{u(s)}{e(s)} = k_p \frac{s}{s} + k_i + k_d \dot{s} \quad (5.5)
\]

Since this differentiator, \( \dot{s} \), is inexpressible in a real system, an approximate differentiator is required. An appropriate, sufficient, approximate differentiator, as explained in [12], equation can be expressed in the s domain as:

\[
\dot{s} = \frac{s}{(\tau s + 1)^2} \quad (5.6)
\]

Figure [11] demonstrates an example 2nd order approximate derivative on a step function to demonstrate the effect of \( \text{tau}=1,0.5,0.25 \).

\[\text{\underline{1\text{ derived on 4/27/02}}}\]
Figure 11: 2nd order approximate derivative with $\tau=1,0.5,0.25$ on a step function

Within the PID equation, $\dot{s}$ can be substituted with the equation above to get the approximate PID transfer function

$$\frac{u(s)}{e(s)} = k_p + \frac{k_i}{s} + k_d \frac{s}{(\tau s + 1)^2}$$

(5.7) can be algebraically solved as a transfer function:

$$\frac{u(s)}{e(s)} = \frac{s^3(k_p \tau^2) + s^2(k_d + 2k_p \tau + k_i \tau^2) + s(k_p + 2k_i \tau) + k_i}{s^3(\tau^2) + s^2(2\tau) + s(1) + s(0)}$$

(5.8)

This simple but powerful and popular controller in this form is ready for realizable implementation.

### 5.3 Nonlinear PID

Nonlinear PID, from [13], is an extension of the standard PID discussed in the previous section, which simply defines a nonlinear gain-scheduled region in terms of error for the P, I, and D parameters. Each term has carefully designed constraints that define the specific gain values for regions of low and high error. For example, the primary function of the nonlinear region is to have a high gain when the error
is small and small gain when the error is large, rather than a continuous gain for all error levels.

The nonlinear PID control law can be described as

$$u = G_{kp}(e) + G_{ki}(e) \int e + G_{kd}(e) \dot{e}$$

(5.9)

where $G_x(e)$ is a specifically designed regional gain for $x$ and is a function of the error.

One simple function that implements these regions of low and high gain for varying errors is called the GFunction, which is a simple implementation using 4 regions of constant gains. It is the basis of the nonlinear PID used within the 'simtk' library. The definition of the G function is described in (5.10).

$$G_x(e, k_{p1}, k_{p2}, k_{n1}, k_{n2}, w_p, w_n) = \begin{cases} 
 e > 0 & \begin{cases} e \cdot k_{p1} \quad |e| < w_p \\
 e \cdot k_{p2} \quad |e| \geq w_p \end{cases} \\
 e \leq 0 & \begin{cases} e \cdot k_{n1} \quad |e| < n_p \\
 e \cdot k_{n2} \quad |e| \geq n_p \end{cases}
\end{cases}$$

(5.10)

where

- $e$ is the error,
- $k_{p1}$ is the positive small error gain constant,
- $k_{p2}$ is the positive large error gain constant,
- $k_{n1}$ is the negative small error gain constant,
- $k_{n2}$ is the negative large error gain constant,
- $w_p$ is the positive width of low error region,
- And $w_n$ is the negative width of low error region.

This can be directly implemented in the following Java code example.
Listing 5.1: GFunction example code in java (extracted from SimGFunc.java)

```java
// positive
if(error > 0)
    // is in the first gain width
    if(error < pw.getValue()){
        output = error * pk1;
    }
    else{
        output = error * pk2;
    }
// negative
else{
    // small region
    if(Math.abs(error) < nw){
        output = error * nk2;
    }
    // large region
    else{
        output = error * nk2;
    }
}
```

5.4 Active Disturbance Rejection Control

ADRC is a great leap toward a solution for plant-independent or near plant-independent control solutions, [11]. The extended states of the plant are observed by the Extended State Observer (ESO). The unknown plant is reduced by estimation to a known double integrator which can be optimally controlled by a static controller such as a PD.

The states of the plant are defined in (5.11), where \( h(t) \) is plant disturbance, and \( b_0 \) is a constant plant parameter which is the only required information about the plant.

\[
\begin{align*}
    \dot{x}_1 &= x_2 \\
    \dot{x}_2 &= x_3 + b_0 u \\
    \dot{x}_3 &= h(t)
\end{align*}
\]  

An ESO, with 3 parameters is fabricated to estimate the state of the plant including
the disturbance:

\[
\begin{align*}
\dot{z}_1 &= z_2 + B_1(y - z_1) \\
\dot{z}_2 &= z_3 + B_2(y - z_1) + b_0 u \\
\dot{z}_3 &= B_3(y - z_1)
\end{align*}
\] (5.12)

The ADRC control law is then shown as

\[
u = \frac{u_0 - z_3}{b_0}
\] (5.13)

A controller has now been built that is defined by 3 terms: \(B_0, B_1\) and \(B_3\) plus \(b_0\). By parameterization\(^2\) these terms can be derived in terms of controller bandwidth, \(\omega_c\), and observer bandwidth, \(\omega_o\). These can then be solved\(^3\) for the linear ADRC for any second order plant as two transfer functions shown in (5.14) and (5.15).

The block diagram in Figure 12 shows how the ADRC controller can be broken into two sections for simple implementation.

---

\(^2\)See Section 5.1 for information on this control tuning method.

\(^3\)Rob Miklosovic solved this equation in the dual transfer function form in 04/02.
The feed-forward controller or pre-filter is defined as:

\[
\frac{U(s)}{R(s)} = \frac{\omega_c^2 [1 \quad 3\omega_o \quad 3\omega_o^2 \quad \omega_o^3]}{b_0 [(1) \quad (3\omega_o + 2\omega_c) \quad (3\omega_o 2\omega_c + \omega_c^2 + 3\omega_o^2) \quad (0)]} \cdot S \tag{5.14}
\]

The feedback controller or feedback filter is defined as:

\[
\frac{Y(s)}{R(s)} = \frac{\omega_c^2 [(3\omega_o \omega_c^2 + 3\omega_o^2 2\omega_c + \omega_o^3) \quad (3\omega_o^2 \omega_c^2 + \omega_o^3 2\omega_c) \quad (\omega_o^3 \omega_c^3)]}{b_0 [(1) \quad (3\omega_o + 2\omega_c) \quad (3\omega_o 2\omega_c + \omega_c^2 + 3\omega_o^2) \quad (0)]} \cdot S \tag{5.15}
\]

where \( S \) is a column vector defined as:

\[
S = \begin{pmatrix}
s^3 \\
s^2 \\
s^1 \\
s
\end{pmatrix} \tag{5.16}
\]

A \( b_0 \) guideline is proposed as:

\[
b_0 = \frac{\text{Largest plant numerator coefficient}}{\text{Highest order plant denominator coefficient}} \tag{5.17}
\]

Once this controller is in transfer function form it could easily be implemented in the SimStf or SimZtf simulation blocks\(^4\). The block diagram of this controller setup is shown in Figure 25. The result is a controller that has very good disturbance rejection and allowance for plant variability, and, more importantly, a simple way to tune with a few reasonable parameters.

### 5.5 Discrete Time Optimal Control

The DTOC algorithm is another leap toward plant model independent control design, discussed and derived in [10]. It is a controller formulated in a constructive

\(^4\)see Chapter 4 on page 27
manner from the optimal control law. Its use in engineering, is very promising due to the level of plant independence, the ease of tuning, and the beautiful implementation simplicity of the algorithm.

DTOC is a discrete solution that comes from another controller called Continuous Time Optimal Control(CTOC) which has been around for some time. The control law for this controller is shown in (5.18).

\[ u = -r \text{sign} \left( x_1 + \frac{x_2 |x_2|}{2r} \right) \]  \hspace{1cm} (5.18)

This law is optimal but not practical due to the chattering that occurs near zero error. The DTOC offers a solution and, not only provides a method to create a linear region near zero error, but it reaches zero in optimal time. The DTOC control law, shown in (5.19) along with the internal definitions, has a simple direct mapping to an implementable algorithm.

\[ u = \text{fst}(x_1, x_2, r, h) \] \hspace{1cm} (5.19)
\[ d = rh \]
\[ d_0 = hd \]
\[ y = x_1 + hx_2 \]
\[ a_0 = \sqrt{d^2 + 8r|y|} \]
\[ a = \begin{cases} 
    x_2 + \frac{a_0 - d}{2} \text{sign}(y), & |y| > d_0 \\
    x_2 + \frac{y}{h}, & |y| \leq d_0 
\end{cases} \]
\[ \text{fst} = -\begin{cases} 
    r \text{sign}(a), & |a| > d \\
    r \frac{a}{d}, & |a| \leq d 
\end{cases} \]

where

\[ u \] is the control signal,
$x_1$ is $e$ or the error signal,

$x_2$ is $\dot{e}$ or the derivative of the error signal (which was obtained with a 2nd order approximate derivative),

$r$ is the saturation limit (maximum control output),

$k$ is the step size scaling parameter to be raised in the presence of noise,

$t_s$ is the discrete sampling period,

$h = kt_s$ is the DTOC step size.

The DTOC algorithm was implemented in code with a sample shown in the following listing.

Listing 5.2: DTOC control algorithm implemented in java

```java
/** DTOC, f s t function. *from dr. Gao, 3/7/3 */
double fst2(double x1, double x2, double u0, double r, double h) {
    double d, d0, y, a0, a1;
    d=r*h;
    d0=d+h;
    y=x1-u0+h*x2;
    a0=Math.sqrt(d*d+8*r*Math.abs(y));
    if (y>d0) a1=x2+(a0-d)/2;
    else if (y<-d0) a1=x2-(a0-d)/2;
    else a1=x2+y/h;
    return(-r*usat(a1,d));
}
/** nonlinear unity saturation function. *from dr. Gao, 3/7/3 */
double usat(double x, double delta) {
    if (x>delta) return(1.0);
    else if (x<-delta) return(-1.0);
    else return(x/delta);
}
```

The result is a controller that has very good disturbance rejection and extreme plant variability with simple tuning and implementation.
5.6 Filters

Low-pass and high pass filters are an important building block and tool for creating models of control systems. In the following sections, a few of the basic filters are derived, and the important role of pre-filter use and design is discussed.

5.6.1 Pre-filters and Profiles

The use of pre-filters or profiles for the input control reference is an important role of control system design that is often not emphasized enough. A step input is often used as the only reference for simulation and design. This creates a host of problems for design and real-world implementation. In the real world, an instantaneous jump from a signal level to another, is impossible. A step input creates the largest error possible to the controller at a given instant. If, instead, the input is slowly brought up to the desired value at a rate that the system can follow, it is much easier for the controller since the error stays within a small region, and, therefore, an overall better design. The control can also be made more active since overshoot would be less likely.

In a motion application, it is physically evident why a profile is required. A step input for position yields in an infinite velocity and acceleration which destroys motors. Even a ramp input yields in an infinite acceleration. So, a method is required to slowly bring up the reference and at the same time slowly bring up the first, second, and third derivatives of the reference signal, for velocity, acceleration, and jerk, respectively.

The Trapezoidal Pre-filter

There are several methods to create a profile or pre-filter. One method is to integrate a Trapezoidal input to get a slowly increasing curve, followed by a slowly decreasing curve until the reference point is reached. In this way, a smooth response
for the velocity is assured. The equation defining a trapezoid is shown in (5.20).

\[
y = \begin{cases} 
0 & x < d \\
\frac{3g}{w} x - \frac{3gd}{w} & d < x < d + \frac{w}{3} < x < d \\
g & d + \frac{w}{3} < x < d + \frac{2w}{3} \\
\frac{-3g}{w} x + \frac{3gd}{w} + 3g & d + \frac{2w}{3} < x < d + w \\
0 & x > d + w
\end{cases}
\] (5.20)

The various parameters in this equation are illustrated in Figure 13, and the response is shown in Figure 14.
The S-curve Pre-filter

Extending from the thought of the trapezoidal pre-filter, the S-curve is smooth in not only the velocity but also in the acceleration. This causes it to be even on the plant dynamics especially for application to motors. The S-curve can be generated from an odd triangular form, which is shown in Figure 15, and defined by (5.21) to (5.23).

\[
y = m_s mx + (-mm_s x_{int})
\]  \hspace{1cm} (5.21)

where

\[
m = \frac{6g}{w} \quad \hspace{1cm} (5.22)
\]

Figure 15: Triangular source for S-curve profiles
and

\[\begin{align*}
m_s = 0; & \quad x_{int} = 0 \quad x < d \\
m_s = 1; & \quad x_{int} = d \quad d < x < d + \frac{w}{6} \\
m_s = -1; & \quad x_{int} = d + \frac{w}{3} \quad d + \frac{w}{6} < x < d + \frac{w}{3} \\
m_s = 0; & \quad x_{int} = 0 \quad d + \frac{w}{3} < x < d + \frac{2w}{3} \\
m_s = -1; & \quad x_{int} = d + \frac{2w}{3} \quad d + \frac{2w}{3} < x < d + \frac{5w}{6} \\
m_s = 1; & \quad x_{int} = d + w \quad d + \frac{5w}{6} < x < d + w \\
m_s = 0; & \quad x_{int} = 0 \quad x > d + w
\end{align*}\]  

(5.23)

and the parameters, \(w\) as width, \(d\) as delay, and \(g\) as the gain, are illustrated in Figure 15.

**Critically Damped Functions**

Another method to generate a curve that smoothly brings the value up to the desired reference level is to simply use a general-second order transfer function after a step input. This is a good solution because a transfer function can easily be put into a difference equation, and the defining parameter for the pre-filter can easily be adjusted with one meaningful bandwidth parameter, \(\omega_r\). A general 2nd order profile defined by \(\omega_r\), is shown in (5.24) with the response shown in Figure 16.

\[R(s) = \frac{\omega_r^2}{s^2 + 2\omega_r s + \omega_r^2}\]  

(5.24)
Two pre-filters can be cascaded in a row, as in (5.25), to create an even smoother profile on deeper levels.

\[ R(s) = \frac{\omega_r^2}{s^2 + 2\omega_r s + \omega_r^2} \frac{\omega_r^2}{s^2 + 2\omega_r s + \omega_r^2} \]  

(5.25)

This is a simple solution. However, for precise control it is ineffective since the response never mathematically reaches the set point where the trapezoidal and the triangular S curve are assured to reach the desired point in a desired time.

**Polynomial Pre-filter**

A final method to generate a pre-filter is via a derived polynomial which is specifically designed so the first, second, third, and fourth derivatives are all still continuous. This allows even higher orders of derivative continuity. Even the S-curve presents some trouble at the triple derivative. The polynomial\(^5\) in (5.26), is simple to implement and has a simple equation which produces great results. \(T_r\) is the desired

\(^5\)This was extracted from a Matlab simulation that Rob Miklosovic had tested in 2/3.
rise time of the profile, and \( g \) is the gain or the final value of the profile.

\[
g(x) = g \begin{cases} 
\frac{10x^3}{T_r^3} - \frac{15x^4}{T_r^4} + \frac{6x^5}{T_r^5} & x < T_r \\
1 & x \geq T_r
\end{cases}
\]  

(5.26)

The plot where \( T_r \) is equal to five is shown in Figure 17.

\[\text{polynomial profile}\]

Figure 17: Profile generated from the simple polynomial with \( T_r = 5 \)

### 5.6.2 General Normalized Butterworth Filters

Butterworth filters are the most basic filters for variety of needs such as low-pass, high-pass and band-pass filters. Various orders of the Butterworth filters are derived as follows.

The first order normalized butterworth filter\(^6\) is given as:

\[
H(s) = \frac{1}{s + 1}
\]  

(5.27)

The second order normalized butterworth filter is given as:

\[
H(s) = \frac{1}{s^2 + \sqrt{2}s + 1}
\]  

(5.28)

\(\text{\(^6\)The following normalized butterworth filters were found at: http://www.apicsllc.com/apics/Sr_3/Sr_3.htm}\)
The third order normalized butterworth filter is given as:

\[ H(s) = \frac{1}{s^3 + 2s^2 + 2s + 1} \]  

(5.29)

The fourth order normalized butterworth filter is given as:

\[ H(s) = \frac{1}{\left(s^2 + \frac{3}{25}\sqrt{26}s + 1\right)\left(s^2 + \frac{181}{506}\sqrt{26}s + 1\right)} \]  

(5.30)

**Low-pass, High-pass, Bandpass, and Bandstop Extension**

The following equations demonstrate how to generate different filters from the normalized butterworth filters. A helpful relationship for the conversion of radians to Hertz is

\[ \omega = 2\pi f \]  

(5.31)

For the low-pass case:

\[ s \rightarrow \frac{s}{\omega} \]  

(5.32)

For the high-pass case:

\[ s \rightarrow \frac{\omega}{s} \]  

(5.33)

For the bandpass case:

\[ s \rightarrow \frac{s^2 + \omega^2}{Bs} \]  

(5.34)

For the bandstop case:

\[ s \rightarrow \frac{Bs}{s^2 + \omega^2} \]  

(5.35)
Lowpass

For the first order case, using (5.32), (5.27) is converted to the familiar transfer function form:

\[ H(s) = \frac{1}{\frac{1}{\omega} s + 1} \quad (5.36) \]

An example of the first order low-pass Butterworth filter filtering noise is shown in Figure 18.

![Figure 18: 1st order low-pass Butterworth filter filtering noise](image)

For the second order case, using (5.32), (5.28) is converted to the familiar transfer function form:

\[ H(s) = \frac{1}{\frac{1}{\omega^2} s^2 + \frac{\sqrt{2}}{\omega} s + 1} \quad (5.37) \]

For the third order case, using (5.32), (5.29) is converted to the familiar transfer function form:

\[ H(s) = \frac{1}{\frac{1}{\omega^3} s^3 + \frac{2}{\omega^2} s^2 + \frac{2}{\omega} s + 1} \quad (5.38) \]
Highpass

For the first order case, using (5.33), (5.27) is converted to:

\[ H(s) = \frac{1}{\frac{\omega}{s} + 1} \quad (5.39) \]

After some algebra the equation can be reduced to the familiar transfer function form:

\[ H(s) = \frac{s}{s + \omega} \quad (5.40) \]

For the second order case, using (5.33), (5.28) is converted to:

\[ H(s) = \frac{1}{\frac{1}{s\omega^2} + \frac{\sqrt{2}}{s}\omega + 1} \quad (5.41) \]

After some algebra the equation can be reduced to the familiar transfer function form:

\[ H(s) = \frac{s^2}{s^2 + \sqrt{2}\omega s + \omega^2} \quad (5.42) \]

For the third order case, using (5.33), (5.29) is converted to:

\[ H(s) = \frac{1}{\frac{1}{s\omega^3} + \frac{2}{s^2}\omega^2 + \frac{2}{s}\omega + 1} \quad (5.43) \]

After some algebra the equation can be reduced to the familiar transfer function form:

\[ H(s) = \frac{s^3}{s^3 + 2\omega s^2 + 2\omega^2 s + \omega^3} \quad (5.44) \]

From a few simple equations, a powerful library of control design tools has been built. This included simple controllers to advanced controllers as well as the inclusion of many types of filters.
Chapter VI

A GENERAL MODEL DIAGRAM

Chapter 5 described a number of control algorithms. In order to make a simple tool to compare different controllers, a method for dynamically selecting and comparing a wide range of controllers is required.

The control system diagram is classically drawn and modeled after the difference in the error, while the pre-filters, feedback filter, and observers are modeled separately. However, these and almost any controller can be combined for a more general view and model of a controller which has two inputs and one output. Putting any controller in this format allows the rest of the system being tested to remain the same while various types of controllers in this general model structure can simply be plugged in and fairly compared.

---

1 for a single input single output system
6.1 Classic System

The control system traditionally has a structure with a loop formed around a difference junction. This can be seen in Figure 19 and a more complex version including the noise and disturbance modeling is shown in Figure 20.

From these basic feedback structures, the system is expanded by slowly adding more components to the appropriate positions. This is often required for a better representation of the plant, but often the controller also requires additional information and blocks in various places in the diagram. Once the complexity has begun to build, it becomes increasingly difficult to fairly compare differing controllers.
6.2 Combined General System

Although the classical representation is a good model of many systems, an even more general representation can be formed in terms of the controller within which any control can be composed. For the sake of generality and pluggable architecture, a more general feedback system could be written by block diagram reduction. The summing junction for the error and reference can be internal to the controller so any pre-filters or feedback filters can be realized. The new representation is shown in Figure 21. This can be compared to the model in Figure 20 to see that the summing junction is now just internal to the controller.

![Figure 21: New General controller model setup](image)

This allows the same plant, sensor, disturbance and input structure to be compared with any controller for a fair comparison and no redesign.

6.3 General Blocks

The general controller block itself with two inputs and two outputs as a general form is illustrated in Figure 22.
An improved general controller block to include a pre-filter and feedback-filter is shown in Figure 23.

The entire system is the control design, and if a general controller is desired the entire sum is a better description. By this method, controllers that do not use error as an input can also be used. In general control diagram structure, the most basic information is provided to the controller, which is the reference and the output. If an error control is desired it can be internally calculated by a subtraction. Using this description, more information than just the error is sent to the box creating a more general controller block that any control design can be implemented within.

An example of the general controller block implementing a PID controller is
shown in Figure 24. If this general form PID block from Figure 24 is substituted into the general controller system in Figure 21, the same classical control system that is illustrated in Figure 20 is achieved by block reduction.

Figure 24: New General controller block example using PID

An example of a general controller block using a ADRC is also shown in Figure 25.

Figure 25: ADRC Controller (which is a special case of the improved general controller where error controller is unity)

A general control block is an important aspect when looking towards a simple system for users to model systems and compare various controllers. It would also
play an important role in computer-automated control design for testing of various controllers.
CHAPTER VII

ACTIVE OPTIMIZATION CONTROL

Parameterization, as discussed in Chapter 5, allows the many variables of a system to be reduced to one conclusive parameter that has a physical meaning to the system, such as the bandwidth. This idea of parameterization offers a whole new approach to active tuning. Instead of auto-tuning in terms of extracting plant information for new control values, the single parameter can be actively optimized to a threshold or limit inherent to the system. The design can then be automated by a computer as long as the requirements are appropriately defined and the method to quantify these requirements is established. Although this scheme of optimization is not directly implemented in the simtk library, the functionality and functions have been built in place for its future development and discussed in Chapter 8.

7.1 Threshold Limit Requirement

Understandably, different systems require different constraints. Some are more dependent on the level of the control signal noise that occurs before the system
goes unstable, which is ultimately the constraint of any system. Often systems are constrained by the rate of change in the control signal. For example, in the case of motor control, changing the speed of the motor can very quickly generate jerk that is harmful to the life of the motor or system.

The optimization threshold has to be carefully chosen for the specific system. This selection is the “cost function” by which the system will be optimized. Defining this limit requires information about the system to know what limit and even what definition to use.

Using these thresholds and optimization tuning, the system solution can be directly optimized in terms of the engineering requirements. This establishes a comfortable connection between control theory and practice.

A simple active optimization scheme has been implemented on a digital DC/DC power converter described in [14]. In this controller, the value of $\omega_c$ was linked to a maximum value that was roughly related to the noise. Even the results for this simple implementation proved effective.

Noise, rate of change in the control signal, and overshoot are a few of the important constraints that occur in practice.

7.1.1 Noise Level Limit

Quantifying the noise level is very important for auto-tuning a system by the method of parameterization. Auto-tuning can be linked to noise by the method of increasing the bandwidth until the noise is beyond a given threshold. By this method, the single parameter of bandwidth would be extended into another single parameter, usually the noise. This allows the controller to automatically adjust the bandwidth in the presence of a quantified noise.

---

$^1$Bosheng Sun implemented this in DSpace in an incremental optimization in 12/2002.
For example, if the controller was reduced to a bandwidth, $\omega_c$, and the system was limited to noise, then $\omega_c$ could automatically be increased until a threshold of allowable noise was reached. This allows the system to be easily tuned for an optimum response by raising the bandwidth to the appropriate level in the presence of the noise. If the noise of a given signal can be quantified, then the bandwidth setting can be automatically updated to match the given noise level. This would be an entirely new way of tuning for automated active tuning of active disturbance rejection.

In each threshold definition, it is equally important to devise a means to mathematically quantify the particular threshold so that it can be calculated on the fly during implementation. Some possible methods of detecting the noise (a few of which have been implemented in the simtk library) are discussed in Section 7.3.

### 7.1.2 Rate of Change Limit

Another level of quantification that could be helpful for tuning $\omega_c$ is the rate of change in the control signal. A system that may not be affected much by noise in the control signal may still be limited by the rate of change$^3$. In this case, the bandwidth would simply be increased until the control signal rate is beyond the constraints of the system.

A good differentiator can be used on the control signal to determine the rate of change. In $^12$, Jing Liu has researched a number of different differentiators that would be useful for detecting the rate of change in the signal. As previously discussed in Chapter $^5$, the 2nd-order approximate differentiator produces good results and has been implemented in the simtk library$^3$.

---

$^3$There may also be many other control signal thresholds that limit, including the saturation limit.

$^3$The 2nd-order approximate differentiator is implemented in the SimDiff class.
7.1.3 Overshoot Limit

Another important criterion for tuning a system is in terms of overshoot. Often a system is required to either have no overshoot or an overshoot within a certain percentage. The bandwidth can be increased until overshoot begins. The overshoot can easily be monitored by testing if the signal goes above the set point after a disturbance.

7.1.4 The Most Significant Bandwidth Threshold

Most systems' requirements have many constraints, including noise, rate of change, and overshoot. For this case, the bandwidth can be increased until the first threshold is hit, giving the best performance within the given restraints. The first threshold can be called Most Significant Bandwidth Threshold (MSBT) which defines the highest bandwidth that can be reached. This is significant because it shows the "bottleneck" point of the system. Many times the level of noise determines how fast the system can be controlled, but in other cases a situation with less noise may be physically constrained. In either case the MSBT would reach the optimal tuned bandwidth for the specified requirements.

7.2 Optimization and Threshold Assumptions

A few assumptions about the active optimization control have been made. The two major assumptions are that the highest bandwidth possible is the best and that the threshold is easily defined.

An assumption that is made throughout this paper for “optimal” control is that the most optimal design has the highest bandwidth. This is a fair assumption because an infinite bandwidth would produce a bang-bang controller, defined in the
mathematics of Time Optimal Control, as the fastest response.

Since all of the parameters are dependent on the threshold limits, they must be easily defined, understood, and measurable. If the thresholds are not available or do not appropriately define the system, the controller will be tuned incorrectly.

### 7.3 Active Noise Quantification Methods

This optimization scheme is a new method of active tuning and optimization that has many interesting branches that should be researched. Many questions still need to be answered. A few of the major topics to be researched are the types and levels of thresholds and limits, and how to programatically quantify them.

In active tuning and optimization of a control system, an important criteria to tune by would be the level of the noise. In this case, a good method is needed to determine the level of noise. The human eye can get a rough idea of the level of noise by looking at a signal plot and watching the width of the variation on the plot. However, a computer algorithm to produce a number that represents the level of noise (quantifying noise) is a bit more involved.

![Figure 26: The human eye can simply ’get a feel’ for the level of noise](image)

The following methods do not contain the complete answer to the problem or
even show the best solutions. They simply show a few of the issues involved with the
problem. They also present a few of the mathematics incorporated in the statistical
block in the simtk library, which could be used for future active optimization tuning.

7.3.1 Max and Min

A maximum sample and the minimum sample can be found over a range, and
this difference could give a measure of the level of noise. (7.1) describes this method,
where $\nu_{\text{maxmin}}$ is the quantification of the noise level from the samples of $y_i$

$$\nu_{\text{maxmin}} = y_{\text{max}} - y_{\text{min}}$$

(7.1)

This is a simple scheme but there are several problems with this method. First of
all, any spikes or outliers in the signal would quickly corrupt the result. Also, this is
only useful for a straight line or a steady-state mode in controls since a rise would be
considered noise.

This scheme is illustrated in Figure 27 with the maximum value, drawn with
a buffer size of 15.

Figure 27: Maximum value, with a buffer size of 15

4SimStat is the simtk class function block, which has a wide range of real-time statistical
functionality.
In the same way the minimum value is depicted in Figure 28 with a buffer size of 15.

![Figure 28: Minimum value, with a buffer size of 15](image)

These two figures, Figure 27 and Figure 28 would then be compared to determine the level of the noise.

### 7.3.2 Moving Average Window

The most common computational algorithm for filtering, due to its cogent simplicity, is the moving average window. It is simple to compute, but the frequency spectrum is shown to be a Comb filter which is not the best filter for white noise. The equation for the mean values is

$$\bar{y} = \frac{1}{n} \sum_{i=1}^{n} y_i$$  \hspace{1cm} (7.2)

An example of this averaging filter with a buffer size of 20 on random noise is shown in Figure 29.
7.3.3 Variance and Standard Deviation

The maximum and minimum method evaluates only two extreme cases or peaks. Another value that could be used which takes into account all of the samples is the variance, $s^2$, and standard deviation, $s$.

\[
s^2 = \frac{1}{n-1} \sum_{i=1}^{n} (y_i - \bar{y})^2 \quad (7.3)
\]
\[
= \sum y^2 - n\bar{y}^2 \quad (7.4)
\]

where $\bar{x}$ is the sample mean defined in (7.2).

A sample plot of the real-time evaluation of the variance on a random noise signal is shown in Figure 30 to hover around a value of 4. If the amount of noise was greater the variance would shift up in magnitude.
Standard deviation is closely related to variance and is defined as

\[ s = \sqrt{s^2} \]  

(7.5)

The standard deviation level on a random noise signal is depicted in Figure 31 to fluctuate around a value of 1.5.

This term can be normalized to the mean value, which is called the Coefficient of Variation (COV) defined in (7.6), producing a general description of the variance.
compared to the signal.

\[ \text{COV} = \frac{s}{\bar{y}} \quad (7.6) \]

This method gives a better range of the variance over all of the samples. However, it is still only useful for systems in steady-state.

### 7.3.4 Linear Regression

Specifying the amount of noise in a signal that would take into account ascending and descending trends requires information about the signal. If the change in the signal over the samples is considered linear, a best fit for the data to a linear equation defined in (7.7) is called the linear regression:

\[ y_i = b_0 + b_1 x_i \quad (7.7) \]

The constants are defined by a least sum of squares of the error from the mean which is called the Least Mean Square (LMS). The solution for the parameters of (7.7) are

\[ b_1 = \frac{(\sum xy) - n \bar{x} \bar{y}}{(\sum x^2) - n \bar{x}^2} \quad (7.8) \]
\[ b_0 = \bar{y} - b_1 \bar{x} \quad (7.9) \]

A feel for how well this matches the data is defined by the correlation coefficient or \( R \) value.

\[ R^2 = \frac{SST - SSE}{SST} \quad (7.10) \]
\[ R = \sqrt{R^2} \quad (7.11) \]

where
\[
SST = \left( \sum y^2 \right) - n\bar{y}^2 \quad (7.12)
\]

\[
SSE = \left( \sum y^2 \right) - \left( b_0 \sum y \right) - \left( b_1 \sum xy \right) \quad (7.13)
\]

This provides a good measure of the noise for all of the data points and for changes as long as they are approximately linear.

**Relationship Between Bandwidth and Linear Model**

The slope of the linear approximation model is limited by the bandwidth of the system\(^5\). For example, the bandwidth states how fast a signal can jump to another level. This is the same definition of the slope or how fast a change occurs\(^6\). The sample size is also related to the bandwidth because we only want to see the effect of changes in the past approximate linear region.

\[
\alpha n \sim \beta b_1 \sim \omega_c \quad (7.14)
\]

There are still quite a few issues that must be resolved for this system to work properly. The bandwidth level is still needed to set the parameters.

**7.3.5 High-Pass Filter With Variance**

Using a high-pass filter to pass only the noise would be a better method than the previous methods to quantify the noise over transients and disturbances. After the high pass filter, the signal should be left with the noise. Then the variance could be taken to show the level of the noise. However, the bandwidth of the system is still needed in order to define the filter to quantify the noise.

\(^5\)This whole section is just speculation, no simulation or testing has been done to verify this idea of bandwidth to linear slope relationship

\(^6\)this is also the same definition of a differentiator, so is this just another implementation of a type of differentiator?
7.3.6 Wavelet for a New Variance Definition

Statistical variance is defined by a difference from the average or mean shown in (7.3). The frequency response of an averaging filter is not a good filter for noise because of the comb nature. The use of a wavelet has proved to be very powerful in extracting the signal from the noise.

It is possible to create a new variance in terms of the variation from the wavelet signal instead of the variation from the average. This idea is shown in (7.15). The wavelet variance can be defined as \( s' \) and \( \bar{y}' \) as the average of the wavelet signal.

\[
\begin{align*}
s' &= \frac{1}{n-1} \sum_{i=1}^{n} (y_i - \bar{y}')^2 \\
&= \sum y_i^2 - n\bar{y}'^2
\end{align*}
\]

This term can be normalized to the mean extracted value, which is called the coefficient of wavelet variation, to produce a general description of the variance compared to the wavelet extracted signal.

\[
\text{COWV} = \frac{s'}{\bar{y}'}
\]

This could be an interesting determination of the noise of the system.\[7\]

Turning the visual notion of noise into a quantified number is difficult and requires some type of statistical analysis on the past data. There are several major issues involved, including steady state with disturbances and the transient cases. To implement active optimization and tuning in the future, much is left to be explored in both the types of thresholds and the methods to quantify them.

---

\[7\]This is simply an idea and has not been tested or even thoroughly analyzed.
CHAPTER VIII

SUMMARY AND FUTURE RESEARCH

There have been a number of topics addressed in a number of fields for the fruition of this real-time simulation package. Topics in the realms of algorithm development, software engineering, and control theory have been discussed and implemented within the constructed software library. The major algorithm work dealt with the choice and construction of an ordinary differential equation solver. From this, a number of software engineering issues were also considered. An extensible framework was controlled by an object-oriented structure. This was used for the basic building blocks of simulation, as well as the GUI interface. A couple of examples of advanced blocks were built around the advanced control algorithms including NPID, ADRC and DTOC.

The construction of this framework for control design and tuning opens a door to future development areas, such as hardware monitoring, hardware tuning, and computer assisted control design.
8.1 Summary

The major work that has been accomplished is evidenced in the library framework for real-time tuning and control as an extensible library of classes, which allows rapid insight into the effects of the parameters in the control system. There are a few simple example applets that show a proof of concept to show a little of what the underlying classes can accomplish.

The object-oriented nature of Java, discussed in Chapter 2, was used to create an extensible framework that is open for additions, modifications and expansions. This is important for building up the library with future algorithms, control methodologies, and user interaction schemes. The objects in the library can be built upon and easily extended to form more complex systems.

The ability to simply construct, model and simulate nonlinear systems has been achieved, which is important for any practical controls simulation package. Both linear mathematical definitions and nonlinear structures can easily be connected and simulated seamlessly with respect to the user. This is done by building the simulation core around discrete difference equations using the Tustin method. From this point, all the linear and nonlinear functions are mapped to the basic difference equation engine. Once all the functions are in the same form, they can be connected to construct and model the entire system.

A major goal has been reached by basing the software in Java, which increases the software availability to many environments. A cross-platform focus allows the package to be available to a larger scope of users. Since Java is built into almost every web browser, the simulation package is available through web-sites, which can open the doors to many who do not have the resources of traditional simulation packages.

The simulator itself should be useful, but it should also help design practical
controllers. A hierarchal design structure defines higher level functions on top of lower level functions. This allows for extendibility but it also provides a transparent conversion from the upper level definitions of a controller to the lower level difference equations. This means that difference equations for code execution are automatically generated from the tuned plant or controller, so the equations are ready to use for a practical implementation.

8.1.1 Simulation Building Blocks

The software library, developed in this work, is broken down into basic simulation building blocks. Three major types of basic building blocks have been built, each with dozens of sub-blocks. The first type is a source that simply has an output, and consists of functions such as step responses, disturbances, and profiles. The second type of blocks, function blocks, make up the largest section of blocks. The most basic block of this function type is the difference equation, which supplies the basis for transfer functions and many other functions. The third major type of block is the sink that is mainly used for data output, as in plots.

Also, included in the constructed library are more advanced blocks that are composed of the basic building blocks. A few of these are some of the advanced control algorithms discussed in Chapter 5. PID with parameterization was implemented by extending a normal PID controller with specific bandwidth controls, which collectively modify the lower level PID parameters. NPID was composed of several basic blocks including a nonlinear GFunction, an integrator and a differentiator. ADRC was generated from transfer functions with overlaid parameterization. Since DTOC is a digital controller from the ground up, it is an example of an advanced controller that directly implements a function type block. From these blocks, a wide range of implementation schemes are represented.
8.1.2 Simple GUI Examples

The software framework built in this research can also be useful for presenting a control algorithm or design to a wide variety of people who might be interested. A simple interface for inserting a plant and testing new controllers has been built. A user wanting to explore and understand the design or algorithm could experiment with a plant without having to re-implement the controller in his or her own simulation package like Matlab. This same use could be extended for a control systems course to help the student learn control theory. A few example applets have been developed to demonstrate a few “canned” systems to explore what can be used for educational purposes and general control design assistance. There are a number of canned programs that have been developed including the General Control Loop, NPID, ADRC, and DTOC applets.

8.2 Future Research

With the entire framework of a simulation engine built, there is a large array of uses and potential avenues to continue with this project.

8.2.1 Additional Library Blocks

There are a number of areas to continue working and enhancing the library framework. A few of these areas include:

- Investigation of more ode solvers, and providing a selection of solver methods for different blocks;
- Speed of simulation for slower computers;
- Transfer Function generated from a step response by curve fitting and time response;
- Separate step-sizes settings for different blocks to simulate step size variance;
- TCP/IP/Serial streaming for real time input;
- Zero Order Hold (ZOH)\(^1\) and
- Saturation functions.

### 8.2.2 Graphical Setup of SimBlocks

The OO structure broke the problem down into nice blocks, which are simply connected to each other to generate the simulation system. For now, they are simply connected in the source code of the Java code with a simple text function setInput(). Since the system framework is built around this way of thinking, a GUI similar to Simulink could be built around these setInput() functions to create a highly configurable dynamic simulation package.

Currently, the order in which the blocks are added to the factory class is the same order in which the blocks are iterated for simulation. There is no algorithm for simulation order discovery. Therefore, some knowledge is needed about the order in which the blocks are connected, especially in those controllers that need the plant information before the controller is calculated. A simulation manager\(^2\) that can determine the order of the simulation would be helpful. If a future GUI is added that is similar to Simulink, a simulation-ordering algorithm will also need to be developed.

---

\(^1\)This could be implemented by skipping do\textit{ ITERATION} and the buffer do\textit{ ITERATION} for the accounted step size.

\(^2\)This could be implemented in a simulationManager and then extended to a guiManager for the GUI functions.
8.2.3 Extension to a Hardware Tuning and Monitoring Interface

The simtk library could be used to find new parameters for a plant, either manually by the user or automatically by the software. The user could simply guide and set the parameters with the interface, as discussed in Chapter 7. It is also possible for the program to monitor the status of the system and dynamically calculate and set new parameters for the system. Another beneficial use of a higher level tuning application like this would allow parameterization setting of multiple parameters by a single unifying parameter.

Since the project is built on Java, the TCP/IP functions are built into the software and can be used for hardware connection. This allows any hardware that has a TCP/IP connection to communicate, monitor, configure, and tune the parameters. If the communication protocol is a self-designed system, this could also be implemented in Java with user-defined drivers.

8.2.4 Extension to Automated Computer Assisted Design for Controls

Finally, the most interesting and exciting plan and potential use of this library is the creation of an ACAD (Automated Computer Assisted Design) for controls: a system that would greatly assist engineers, both novice and experienced, in providing systematic and intelligent control designs.

The papers presented by Zhiqiang Gao, [8] and [10], in the past years focus on systematic understanding of the basics of control systems in order to get a full understanding of the issues in real-world engineering problems. These basic systematic understandings of control systems can be programmed into a computer program
to automatically do the work for general engineers, which they used to arrive at through trial and error. The real-time simulation software library combined with the discussion about active optimization, in Chapter [7] and the recent fundamental understandings of control theory paves the way for a move toward automated control design.
BIBLIOGRAPHY


APPENDIX
APPENDIX A

ADDITIONAL REFERENCES

A.1 Project Link

http://academic.csuohio.edu/aerl/simtk/

This web-site contains all of the information for the simtk project. This site is also kept up-to-date with the most recent work on the project. It includes the Sample Applets which can be run from anywhere with a Java-enabled browser. This site also includes the simtk documentation of the classes and papers and slides to support the project.

A.2 Simtk API Documentation

Simtk Javadoc in HTML

The javadoc program is a powerful documentation tool to help document Java coded programs. It is a standard documentation of classes and methods. The ‘simtk’ javadoc html documentation of classes is supplied at the following link:
Simtk Javadoc in PDF

Javadoc can take plug-ins to convert to other formats instead of html. One type of format is \LaTeX which can easily be converted to pdf. The ‘simtk’ javadoc documentation in pdf format is provided at the following link:

http://academic.csuohio.edu/aerl/simtk/latexdocs/simtk.pdf

The plug-in or Doclet for the javadoc to produce \LaTeX files is called TexDoclet and can be found at:

http://www.xosoftware.dk/txdoclet.html

PtPlot

The API information for PtPlot which is the basis of the plotting class can be found at:

http://ptolemy.eecs.berkeley.edu/java/ptplot5.1p1/ptolemy/plot/doc/index.htm

Sun Microsystem’s Java API Documentation

The powerful base library for the entire java language can be found at:

http://java.sun.com/j2se/1.3/docs/api/index.html

A.3 Project Tools

The following is a list and description of the tools and software used to help build the simtk project. Many of the tools are open source programs available under General Public License (GPL).
A.3.1 Software Packages

Java

Sun Microsystems is the originator and home for the Java platform and development.

http://java.sun.com

Mathematica

Mathematica is a powerful computer algebra system that was used to develop many of the algorithms. Wolfram Research also hosts the informative Mathworld math library as a mathematics reference.

http://www.wolfram.com

SysQuake

SysQuake is the interactive control design package that spawned the idea for this project.

http://www.calerga.com

Matlab

Matlab is the popular control design package produced by Mathworks which has a GUI system called GUIDE for graphical user interface development. The most recent version 6 has a more user friendly interface. However since this project started from scratch for the solver equations, it was simple to implement it in another package such as java that has some other interesting advantages.

http://www.mathworks.com
Ptolemy

Ptolemy is the project from which the open source plotting package PtPlot was obtained. The people at Ptolemy also created a system that is very similar to the focus of this project although it is focused on the simulation of communication systems rather than controls.

http://ptolemy.eecs.berkeley.edu/java/ptplot5.1p1/ptolemy/plot/doc/index.htm

Java Math Exploration

The following is a site that explores odes with java.

http://www.integretechpub.com/examples/interactive/SODE_java.htm

Octave

Octave is an open source program that almost mirrors Matlab. It is a powerful Unix program that is heavily script-able.

http://octave.sourceforge.org

Perl

Perl was used as the glue to hold and bring the whole project together. It was used extensively for document management, project compilation, table and figure generation, as well as some initial algorithm testing and many other uses.

http://www.perl.com/

Gnuplot

Gnuplot was used to test the original solver algorithms. Gnuplot simply plots data points within text documents.
Simple Ode Solvers

The following site is a great resource for learning about ode solvers.

http://www.math.ubc.ca/~feldman/demos/demo2.html

Hartmath

Hartmath is an interesting computer algebra system built entirely in Java. A few of the derivations for the algorithms were used with this program. It is also another interesting engineering computer program built on Java that is accessible to everyone.

http://www.hartmath.com/

Vim

Vim is a powerful text editing program. The thesis and the source code were written with this program.

http://www.vim.org/

Graphviz

Graphviz is a powerful package which generates graphs from the dot language.


\LaTeX

\LaTeX was used as the formatting system for the documentation and thesis. It is based of the powerful open source \TeX package designed by Donald Knuth which has been unparalleled in precision for scientific publications.

http://www.tug.org/
A.3.2 Constructed Tools

Aaindex

This is used for generating web pages for download-able files and papers.

Aao2tex

This is an increasingly powerful personal formatting system to generate $\LaTeX$ files and maintain papers and presentations within the same source and allow auto import of new simulation results and code updates.

Aao2pdf

This is a wrapper script to use the aao2tex program but beyond the step of creating the $\LaTeX$ file it also converts it to pdf files.

Aao.vim

This is a Vim syntax highlighting description for the aao personal formatting code.

Java2dot

This is a perl script to create the directed graphs of the class structure hierarchy. This is used to generate a graphviz dot file directly from the Java code.

Containment

This creates a hierarchal structure of the classes in the aao outline format that are contained within each other. This shows some insight into the interconnection of the data flow for the software. This generates a graphviz dot file directly from the java code.
Aao2dot

This converts the aao outline format to directed graphs in the dot format, which graphviz compiles.

ExtractClassInfo

This generates an aao outline table from a stripped html javadoc format. This automatically imports the current classes and descriptions directly for the java code at least to the current javadoc generated files.

Compressgif

A Perl script to convert any graphics file to a compressed GIF, the script is a wrapper around the Unix/imagemagick algorithm from http://abel.math.harvard.edu/computing/gif/

Csuthesis.sty

LaTeX file style sheet adopted from uthesis.sty from the university of Texas extended from the work Shahid Parvez had done.

http://www.utexas.edu
APPENDIX B

CLASS DESCRIPTION TABLES

The following tables summarize the entire list of the classes and a short description of the function. There are three major types of classes. First of all there are the simulation components in the simtk family. Then there are the applets of the example uses of the library. And finally the specific dynamic variable family.

Table II: Class list and description for the 'simtk' package

<table>
<thead>
<tr>
<th>Class name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ContainsFactory</td>
<td>Interface to simply make sure that the proper functions are implemented in-</td>
</tr>
<tr>
<td></td>
<td>order to contain a factory.</td>
</tr>
<tr>
<td>HasInput</td>
<td>Interface for any Block that has an input; this way it is easy to test if a</td>
</tr>
<tr>
<td></td>
<td>block has an input with instanceof.</td>
</tr>
<tr>
<td>HasOutput</td>
<td>Interface for any Block that has an output; this way it is easy to test if</td>
</tr>
<tr>
<td></td>
<td>a block has an output with instanceof.</td>
</tr>
<tr>
<td>SimAdjustable</td>
<td>Adjustable abstract type, (a new method based around the individual variables is now used for adjusting the blocks for getPanel()).</td>
</tr>
<tr>
<td>Class</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SimPanelable</td>
<td>If a class extends this class it has the ability to make an adjustment panel</td>
</tr>
<tr>
<td>SimReLoadable</td>
<td>Interface ensures that a method has a <code>calculateAndPlot()</code> method.</td>
</tr>
<tr>
<td>Simulatable</td>
<td>Interface ensures that a method has a <code>doSimulation()</code> method and the <code>get-SimParams()</code> method.</td>
</tr>
<tr>
<td>ScrollablePicture</td>
<td>Displays pictures in a scroll-able form, implemented from Sun’s Java example on ScrollPanes, (not used in the simtk library).</td>
</tr>
<tr>
<td>Sim1stOrder</td>
<td>First order function with a single variable ‘a’ (the pole location).</td>
</tr>
<tr>
<td>Sim2ndOrder</td>
<td>Second order function with 2 variables ‘w’ (omega) and ‘z’ (zeta).</td>
</tr>
<tr>
<td>SimAdd</td>
<td>Addition block which adds multiple inputs together which results in a single output.</td>
</tr>
<tr>
<td>SimADRC</td>
<td>Compilation of SimBlocks to create the ADRC (Active Disturbance Rejection) block</td>
</tr>
<tr>
<td>SimBase</td>
<td>The main abstract block for many of the classes, providing reload mechanisms, as well as titles and description methods.</td>
</tr>
<tr>
<td>SimBlock</td>
<td>Main abstract block for all types of SimBlocks, sets defaults and sets up a fresh variable container and GUI panel.</td>
</tr>
<tr>
<td>SimBuffer</td>
<td>Buffer block, the glue or wires that connects the SimBlocks together.</td>
</tr>
<tr>
<td>SimContinuousProfile</td>
<td>Compilation of SimStep and Sim2ndOrder to create a profile.</td>
</tr>
<tr>
<td>SimDialogManager</td>
<td>An old class which at one point was used to create JDialogs for the tuning of SimBlocks, but was replaced with the OO variables.</td>
</tr>
<tr>
<td>Class Name</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SimDiff</td>
<td>Second order approximate differentiator defined by a Stf and is a double pole.</td>
</tr>
<tr>
<td>SimDiffEq</td>
<td>Difference equation which is provided for an intermediate class as well as display of the difference equation for implementation purposes.</td>
</tr>
<tr>
<td>SimDisturbance</td>
<td>Disturbance sources which include frequency and gain variable sine and square waves.</td>
</tr>
<tr>
<td>SimDoubleIntegrator</td>
<td>Directly calculated Tustin double integrator.</td>
</tr>
<tr>
<td>SimDoubleTriangle</td>
<td>Double triangle profile function, which after integration produces an S curve profile.</td>
</tr>
<tr>
<td>SimDTOC</td>
<td>The DTOC (Discrete Time Optimal Control) Controller; uses two inputs, input1: error, input2: error differentiated.</td>
</tr>
<tr>
<td>SimDTOCDiff</td>
<td>The DTOC algorithm with an internal differentiator so a single input of error is required.</td>
</tr>
<tr>
<td>SimFactory</td>
<td>Factory class which makes and controls the simulation environment both for the entire system and hierarchal hybrid blocks.</td>
</tr>
<tr>
<td>SimFilter</td>
<td>Filter with a selection of multiple types of Butterworth filters.</td>
</tr>
<tr>
<td>SimFunction</td>
<td>Defines the bare bones of a single output Function block with multiple inputs.</td>
</tr>
<tr>
<td>SimFunction2Input</td>
<td>Defines the general class for 2 inputs and 1 output (this was replaced with the more general SimFunction).</td>
</tr>
<tr>
<td>SimFunctionFactory</td>
<td>Compilation internalizes the SimFactory into a SimFunction for complex hybrid function blocks such as NPID composed from simple building blocks.</td>
</tr>
<tr>
<td>Class</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SimFunctionSelector</td>
<td>Provides dynamic selection of blocks by the user at runtime; it provides a list of simulation blocks which replaces the current block.</td>
</tr>
<tr>
<td>SimFunctionTest</td>
<td>A test block to test internal SimFactory block's.</td>
</tr>
<tr>
<td>SimFunctionXInput</td>
<td>Multiple input SimFunction (has been replaced with the more general SimFunction which can now have multiple inputs).</td>
</tr>
<tr>
<td>SimGain</td>
<td>Simple function applies a gain to the input.</td>
</tr>
<tr>
<td>SimGeneralClasicController</td>
<td>A hybrid block in the general controller form that allows a simple controller with the classic “r-y = e” internally calculated to appear to a purely error based controller.</td>
</tr>
<tr>
<td>SimGFunc</td>
<td>A non-linear gain function for use in shaping PID gains for NPID.</td>
</tr>
<tr>
<td>SimImage</td>
<td>Allows a picture such as a block diagram to be loaded on a panel.</td>
</tr>
<tr>
<td>SimImageFactory</td>
<td>Common utilities for loading pictures from files.</td>
</tr>
<tr>
<td>SimInput</td>
<td>Defines the bare bones of a multi-input block for multi-input functions and sinks.</td>
</tr>
<tr>
<td>SimIntegrator</td>
<td>A directly derived Tustin integrator.</td>
</tr>
<tr>
<td>SimMultiply</td>
<td>Multiplication of two inputs with the result at the output.</td>
</tr>
<tr>
<td>SimNoise</td>
<td>Source which after each iteration outputs a random value from -1 to 1.</td>
</tr>
<tr>
<td>SimNothing</td>
<td>Outputs zeros, this can be used for test cases of zero noise instead of having some noise.</td>
</tr>
<tr>
<td>SimNPID</td>
<td>Compilation of SimBlocks with a couple SimGFunc to create the NPID(Non-linear Proportional Integral Derrivative) Controller.</td>
</tr>
<tr>
<td>Class</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SimNPIDwc</td>
<td>Extends the SimNPID into a parameterized PID in terms of omega for a double integrator plant.</td>
</tr>
<tr>
<td>SimParameters</td>
<td>Stores and keeps all the important global type simulation parameters within this duplicated class which each block recieves.</td>
</tr>
<tr>
<td>SimPid</td>
<td>Approximate differentiator PID, this is a hybrid block composed of simpler blocks.</td>
</tr>
<tr>
<td>SimPolyProfile</td>
<td>Provides a smoother response for position, velocity, and acceleration and Jerk in place of a step function or trapezoidal function.</td>
</tr>
<tr>
<td>SimProfile</td>
<td>A dynamically create-able profile which has a selection of types of inputs(step, trapezoidal, triangle) and a selection of types of filters(integrators, 2nd order, butterworth).</td>
</tr>
<tr>
<td>SimPtPlot</td>
<td>The main plotting engine depends on the Ptolemy PtPlot package; this adds interfaces for plot exporting and connections.</td>
</tr>
<tr>
<td>SimSaturation</td>
<td>Simple saturation function, contains and upper and lower bound limit.</td>
</tr>
<tr>
<td>SimSink</td>
<td>Defines the bare bone necessities to a simBlock which only has inputs.</td>
</tr>
<tr>
<td>SimSource</td>
<td>A Bare-bones definition of a source which only has an output.</td>
</tr>
<tr>
<td>SimSourceFactory</td>
<td>Compilation internalizes the SimFactory into a SimSource for complex hybrid source blocks such as some profiles.</td>
</tr>
<tr>
<td>SimSourceSelector</td>
<td>Provides dynamic selection of blocks by the user at runtime; it provides a list of simulation sources which replaces the current block.</td>
</tr>
<tr>
<td>Class name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SimStat</td>
<td>Provides statistical functions, such as average and standard deviation, on incoming data over a specified range of samples.</td>
</tr>
<tr>
<td>SimStdout</td>
<td>A sink that simply sends the current output to the standard output.</td>
</tr>
<tr>
<td>SimStep</td>
<td>A simple source Step function which reaches a specified gain value as fast as possible.</td>
</tr>
<tr>
<td>SimStf</td>
<td>Powerful block that simulates S domain transfer functions and converts them to discrete transfer functions!</td>
</tr>
<tr>
<td>SimSubtract</td>
<td>Subtracts the second input from the first input; &quot;out = in1 - in2&quot;.</td>
</tr>
<tr>
<td>SimTest</td>
<td>A test of extending the SimFunctionFactory.</td>
</tr>
<tr>
<td>SimTextBoxOutput</td>
<td>A sink block that simply send the current output to a formatted output in a swing text box.</td>
</tr>
<tr>
<td>SimTimeIndex</td>
<td>Outputs the current time index which is useful for testing blocks over a range of inputs.</td>
</tr>
<tr>
<td>SimTrapezoid</td>
<td>Trapezoidal profile which can be integrated to provide a smoother response for position, velocity, and acceleration in place of a step function.</td>
</tr>
<tr>
<td>SimTrapezoidalProfile</td>
<td>Trapezoidal profile combined with an integrator to form a smooth profile.</td>
</tr>
<tr>
<td>SimZtf</td>
<td>Z domain transfer function that automatically creates difference equations.</td>
</tr>
<tr>
<td>SimZtfDirect</td>
<td>A Z domain transfer functions that calculates the output directly without handing the output down to the diffEq class.</td>
</tr>
</tbody>
</table>

Table III: Class list and description for the ‘simtk.applets’ package

<table>
<thead>
<tr>
<th>Class name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADRC</td>
<td>GUI ADRC example applet of the simtk library.</td>
</tr>
<tr>
<td>Console</td>
<td>Console example of the simtk library to allow plot exports in non-applet mode</td>
</tr>
<tr>
<td>Class name</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>GCADRC</td>
<td>GUI ADRC example applet using the General Controller model.</td>
</tr>
<tr>
<td>GeneralControlLoop</td>
<td>GUI example applet using the General Controller model for selectable controllers.</td>
</tr>
<tr>
<td>SimpleLoop</td>
<td>GUI example applet illustrating a super simple control loop.</td>
</tr>
<tr>
<td>SimpleLoopNN</td>
<td>GUI example applet illustrating a super simple control loop with no noise.</td>
</tr>
<tr>
<td>TestBlock</td>
<td>GUI example applet with no loop, simply meant to test block functions with an input and output.</td>
</tr>
</tbody>
</table>

Table IV: Class list and description for the ‘simtk.variable’ package

<table>
<thead>
<tr>
<th>Class name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimCheckBox</td>
<td>A check box boolean variable with a title, which refreshes when the value changes.</td>
</tr>
<tr>
<td>SimComboBox</td>
<td>A combo box that can be used for selection of options in a block; when a new value is selected the simulation will refresh; it is mainly an upper level base object.</td>
</tr>
<tr>
<td>SimComboBoxNumber</td>
<td>A combo box that can be used for selection of SimNumbers.</td>
</tr>
<tr>
<td>SimComboBoxString</td>
<td>Holds a combo box that can be used for selection of strings.</td>
</tr>
<tr>
<td>SimDoubleArray</td>
<td>An array that for now holds variables of double type, the array is displayed as a comma separated list which can be used for polynomial coefficient setting.</td>
</tr>
<tr>
<td>SimInteger</td>
<td>A SimNumber that for now holds an integer rather than a double type.</td>
</tr>
<tr>
<td>SimNumber</td>
<td>A variable that holds a double type; this is the main variable used for real-time GUI parameters with a slider and animation button.</td>
</tr>
<tr>
<td>Class Name</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SimStaticNumber</td>
<td>A SimNumber that has an overloaded getPanel() to simply display the value of the number and not be varied by the user.</td>
</tr>
<tr>
<td>SimString</td>
<td>A variable that holds a string, and refreshes the GUI when a new string is entered.</td>
</tr>
<tr>
<td>SimTest</td>
<td>Purely a test class to test the package and compiler paths</td>
</tr>
<tr>
<td>SimTextBox</td>
<td>A text box variable, for display of text material</td>
</tr>
<tr>
<td>SimVariable</td>
<td>The main abstract block for all types of SimVariables, which provides requirements for the reloading of the GUI mechanism.</td>
</tr>
<tr>
<td>VariableContainer</td>
<td>A container which is used within a block to holds each of the required SimVariables, from this the GUI panel is later dynamically constructed.</td>
</tr>
</tbody>
</table>
APPENDIX C

USER’S GUIDE

This section contains a short introduction to the usage of the simtk GUI demonstration applets. Again, more detailed information to the features can be found in the javadoc documentation, but this provides a short overview of the interface structure, peculiarities, and common troubleshooting points.

C.1 Obtain the Java Runtime Environment

In order to run the simulation package, the latest Java Runtime Environment (JRE) from Sun is required. This can be downloaded from sun micro-systems at http://java.sun.com, which has links to download the JRE. The JRE 1.4 was used for the compiling and testing of this package, but it has also been successfully compiled and run on version 1.3.
C.2 Interface Structure

The graphical user interface of the primary demonstration applet is shown in Figure 4. There are three major sections, including the block diagram, plots, and configurable parameters selectable via tabs. Most should be self explanatory which is partly the purpose of GUI’s however there are a few discipline specific explanations.

C.2.1 Plots

The plots or scopes are provided through the mechanisms provided by the PtPlot Java package. This provides navigation of the graph through smooth and easy rectangular mouse selection to zoom in and out. Zooming in is provided by creating a rectangle by clicking and dragging from the top to the bottom of a desired region. Dragging from the bottom to the top zooms out.

The plot can be opened in a new window for resizing however the old window will lose its reference to the plot. Returning to the original setup can be done by refreshing the applet.

The PtPlot package also provide the support for exporting the plots to Encapsulated Postscript (EPS). However, since it is saved to a file, it can not run in applet mode. The console version of the example simtk applet can be used to save plot outputs. To run in this mode, the Console version must be downloaded and run. One important point to note about saving plots is the memory size of the saved files. If the step size is too small there may be many points in the plot output resulting in a large memory size.

C.2.2 Tabbed Blocks

Each simulation block, that is included in the simulation, is added to the list of tabs in the upper left of the window. These tabs have a hierarchical structure; there
may be many tabs under a single tab. For example, a NPID block has nonlinear, proportional, integral, and derivative tabs each separate for their own parameters.

**This and Super Tabs**

More advanced blocks are extensions of more primitive blocks. An advanced block usually has some parameters that are in control of the lower level parameters. The upper level tabs are under the default “this tab” and the lower level parameters can be viewed under the “super tab”.

**Simulation Parameters**

Global values, used within many blocks, are contained within the simulation parameters tab. This includes parameters such as step-size and final time. These are important as global variables because some blocks reference the step-size for solver calculations.

**Diagrams**

Block diagrams explain the interconnection of the blocks that make up the selectable tabs. A few of the controllers have a tab called “diagram” that is available to display the interconnection of the controller. Combined with the overall diagram this can show the full structure of the system.

**C.2.3 Parameters**

Parameters are the dynamic adjustable variables that can be varied in real-time to adjust design. There are a couple of main types of parameters; each have their own graphical user interface methods.

Some parameters may not be able to be adjusted if there is an upper level
parameter. In this case the parameter will simply remain the same and is only available for viewing.

**Numbers**

Numbers are the most common parameter type. These are adjusted by sliders and text fields. The sliders behave in an exponential varying manner. A slider can vary the value of a number by a magnitude of 4 up or down. Each time the slider is released the new value becomes the center point. The center point can also be manually reset via a text box which requires a **<CR>** Carriage Return to refresh the simulation with the new value.

A recent addition to the package is a small check box next to a slider of a number variable. This button puts the simulation into animation mode which swings the variable slider back and forth around the last set point to automatically view the changes. This is a valuable feature to allow varying multiple parameters at the same instant. The computer can automatically vary a parameter while the user manually adjusts another. Selecting the animation buttons becomes taxing on the processing power of a computer. If too many animation buttons have been selected, the refresh rate of the simulation can be significantly reduced.

**Arrays**

Arrays are comma delimited numbers. These are used in functions such as transfer functions for the S domain and Z domain with class titles of SimStf and SimZtf respectively.

For example, a Stf numerator array of coefficients

\[ 1, 2, 3, 4 \]
Would result in a Laplace polynomial of

\[ s^3 + 2s^2 + 3s + 4 \]  \hspace{1cm} (C.1)

This is similar to the format that both Matlab and SysQuake use to define transfer functions.

### C.3 Example Usage Steps and Tips

The following is an ordered list of the steps to take when using the example controls applet for the simtk library.

1. Open applet from a web page\(^1\) or a local disk

2. Select and modify a plant under the plant tab

3. Select an input reference (usually a step or a profile is used as an appropriate reference)

4. Select a controller under the controller tab

5. Adjust the final time and simulation step size to match the design requirements as well as an appropriate refresh rate. If the product of final time and step size is too large, the computer might not be able to simulate in “real-time” for each parameter update.

6. Select noises and disturbances to test the stability of the selected controller for the selected plant.

7. If the controller is parameterized, adjust the bandwidth of the controller along with the bandwidth of the reference profile to appropriately control the system.

\(^1\)The applet is currently available at [http://academic.csuohio.edu/aerl/simtk/](http://academic.csuohio.edu/aerl/simtk/)
C.4 Troubleshooting

There are a few common troubleshooting points that are addressed in the following sections.

Slow Update Time or Frozen Interface

If the product of final time and step size is too large, the computer might not be able to simulate every iteration quick enough to appear real time. Try reducing the simulation step-size, final time or both.

Large Plot Exports

If the step size is too small, there may be many points in the plot output resulting in a large memory size.

Entering a Transfer Function

For now, the transfer functions are limited to 4th order. If the plant has an order greater than this, two transfer functions can be cascaded to produce the required order.
APPENDIX D

ADDITIONAL FIGURES

There are additional figures that, while not strictly important to be included in the core of the document, could prove useful for future information.

D.1 Inheritance Structure

The following are some additional figures to complement the figures from Chapter 2.
Figure 32: The class structure for the blocks
Figure 33: The entire hierarchical class structure of the project (without implementation classes)

Figure 34: The class structure for the dynamic variables
Figure 35: The applet class structure

Figure 36: The class structure for the dynamic variables (without implementation classes)
Figure 37: The applet class structure (without implementation classes)
D.2 Containment Structure

The following are some additional figures to complement the figures from Chapter 3.

The un-simplified, entire containment of the classes for the GeneralControlLoop applet is illustrated in Figure 38.

Figure 38: Example Containment of classes for the GeneralControlLoop applet

D.3 Screen Shots

A screen shot of an early version of the GUI for the sample applet of the simtk library applet is shown in Figure 39.
Figure 39: Early screen shot for the simtk applet GUI

The structure is rather confusing and a more organized view is shown in Figure 40.
Figure 40: Early organized screen shot for the simtk applet GUI

The current screen shot for the sample simtk applet GUI is shown in Figure 4 in Chapter 2.
APPENDIX E

SAMPLE CODE

Due to the amount of source code for the project a few of the important code listings are cited here. The entire source can be obtained from the web site or from a CD-ROM.

Listing E.1: GeneralControlLoop applet example of library use

```java
/*
 <applet
  width="1000"
  height="600"
  code="simtk.applets.GeneralControlLoop.class"
  codebase="../../
  archive="ptolemy/plot/plotapplet.jar"
  alt="Your browser understands the &lt;APPLET&gt; tag but isn’t running the applet, for some reason.">
</applet>
*/

Fri May 3 12:18:40 EDT 2002

package simtk.applets;

//import ptolemy.plot.*;
import simtk.*;
import javax.swing.*;
import java.awt.*;

/**
 * Sample of the javasim library
 * @author Aaron Radke
 * title: GeneralControlLoop
 * <br>
 * author: Aaron Radke
 * <br>
 * date: 2/5/03
 * <br>
 * abstract: Sample of the javasim library making use of the general classical control structure so any algorithm with any plant can be selected!
 */
```
public class GeneralControlLoop extends JApplet implements Simulatable{
    /** Return a string describing this applet. */
    public String getAppletInfo() { return "by Aaron Radke in AERL\n"; }
    /** Initialize the applet. Here we step through an example of what the
     * the applet can do.
     */
    //simulation parameters
    SimParameters params = new SimParameters();
    //give the factory these parameters
    SimFactory factory = new SimFactory(params);
    public void init() {
        super.init();
        simulation_init();
    } //end of init()
    public SimParameters getSimParams(){
        return params;
    }
    void simulation_init(){
        factory.setReloadBlock(this);
        params.setReloadBlock(this);
        //initialize simblocks
        //create some blocks to play with
        SimTextBoxOutput textout = new SimTextBoxOutput();
        SimPtPlot outputscope = new SimPtPlot();
        outputscope setHidden(true);
        SimImage image = new SimImage();
        image.setFileName("adrc_closedloop.gif");
        image setHidden(true);
        SimSubtract feedbacksub = new SimSubtract();
        SimFunctionSelector fsC = new SimFunctionSelector();
        fsC.cb setValue(SimFunctionSelector.GAIN);
        SimFunctionSelector fsP = new SimFunctionSelector();
        fsP.cb setValue(SimFunctionSelector.INT);
        SimStep step = new SimStep();
        SimSourceSelector tstep = new SimSourceSelector();
        SimGain rgain = new SimGain();
        SimGain ygain = new SimGain();
        SimSourceSelector ssR = new SimSourceSelector();
        SimSourceSelector ssN = new SimSourceSelector();
        ssN.cb setValue(SimSourceSelector.NOTHING);
        SimSourceSelector ssD = new SimSourceSelector();
        ssD.cb setValue(SimSourceSelector.NOTHING);
        SimADRC adrc = new SimADRC();
        SimAdd distAdder = new SimAdd();
        SimAdd noiseAdder = new SimAdd();
        SimGeneralClassicController genc = new SimGeneralClassicController();
        //set simblock initial values
        params.setParameters(10,0.1,4);
        //============factory add api
        //factory.add(outputblock, "output block title", inputblock);
        //-------------
        SimSource ref = ssR; //ssR
        SimFunction controller = genc; //adrc; //genc; //fsC
        SimFunction plant = fsP; //fsP
        SimSink output = textout;
        SimSource plant_dist = ssD; //ssR
        SimSource sens_noise = ssN; //ssR
        factory.add(image,"ImageBlock");
    }
}
factory.add(ref, "input");
factory.add(plant_dist, "Plant\text{\textregistered}\ Disturbance");
factory.add(distAdder, "dist\text{\textregistered}\adder\text{\textregistered}", controller, plant_dist)
factory.add(plant, "Plant\text{\textregistered}\distAdder");
factory.add(sens_noise, "Sensor\text{\textregistered}\Noise/\text{\textregistered}\Disturbance");
factory.add(noiseAdder, "output\text{\textregistered}\noise\text{\textregistered}adder\text{\textregistered}\Plant\text{\textregistered}\sens_noise");
//factory.add(output, "Output Text", plant);
factory.add(outputscope, "Output\text{\textregistered}\Scope", ref);
outputscope.addLegend(0, "Reference");
outputscope.addInput(plant);
outputscope.addLegend(1, "Plant\text{\textregistered}\output");
outputscope.addInput(controller);
outputscope.addLegend(2, "Controller\text{\textregistered}\Output");
// $e = r - y$ // ref\text{\textregistered}\plant to controller
factory.add(controller, "General\text{\textregistered}\Controller", ref, noiseAdder);

Container cp = getContentPane(); //default is border layout
// Create a split pane with the two scroll panes in it.
JSplitPane horzSplitPane = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
factory, getPanel(), outputscope, getPanel());
horzSplitPane.setOneTouchExpandable(true);
horzSplitPane.setDividerLocation(250);
JSplitPane vertSplitPane = new JSplitPane(JSplitPane.VERTICAL_SPLIT,
horzSplitPane, image, getPanel());
vertSplitPane.setOneTouchExpandable(true);
vertSplitPane.setDividerLocation(420);
cp.add(GridLayout.CENTER, vertSplitPane); //put factory and plot in center
 cp.add(BorderLayout.NORTH, new JLabel("General\text{\textregistered}\Control\text{\textregistered}\System",
 SwingConstants.CENTER)); //put button blocks on the bottom
doSimulation();

} //end of simulation init

/** this is not really used anymore??*/
public void doSimulation(){
 factory.doSimulation();
}
} //end

Listing E.2: 2nd Order function example of adding to the library

package simtk;
import simtk.variable.*;
import java.util.*;
import javax.swing.*;
/** (w^2)/(s^2 + 2*w*z*s + w^2)
 */
public class Sim2ndOrder extends SimZtf implements SimPanelable{
  /**omega is w as in
   * (w^2)/(s^2 + 2*w*z*s + w^2)
   */
  SimNumber omega = new SimNumber("Omega\text{\textregistered}(Bandwidth)", 1);
  /**zeta is z as in
   * (w^2)/(s^2 + 2*w*z*s + w^2)
   */
  SimNumber zeta = new SimNumber("Zeta\text{\textregistered}(Damping\text{\textregistered}coefficient)", 1);
  /** constructor set omeg = 1 and zeta = 1
   * (w^2)/(s^2 + 2*w*z*s + w^2)
   * set a to 1 so default of 1/(s+a) is 1/(s+1)
   */
  public Sim2ndOrder(){
    setTitle("2nd\text{\textregistered}order");
    setDescription("2nd\text{\textregistered}order\text{\textregistered}critically\text{\textregistered}damped");
    getVariableContainer().add(omega);
    getVariableContainer().add(zeta);
public void redefine()
{
    /** Recalculates with new parameters, step sizes parameters.
     * This is a method that might be good to have defined in the SimBlock
     * as abstract.
     */
    double stepsize = getSimParams().getStepSize();
    // tustin
    zeroNumAndDen();
    // get temp variables
    double omega = this.omega.getValue();
    double zeta = this.zeta.getValue();
    // 2nd order plant
    double TTww = stepsize * stepsize * omega * omega;
    double Tzw4 = stepsize * zeta * omega * 4;
    setNum(new double[] { TTww, 2*TTww, TTww, 0 });
    setDen(new double[] { 4+Tzw4+TTww, 2*(-4+TTww), 4-Tzw4+TTww, 0 });
    // now redefine the inherited blocks
    super.redefine();
}

}// end of Sim2ndOrder
INDEX

aaindex, 93
aaovim, 93
aao2dot, 94
aao2pdf, 93
aao2tex, 93
ACAD, 83
ADRC, 46 64 81
algorithms, 28
API, 17
Application Programmers Interface, 17
auto-tuning, 66
Automated Computer Assisted Design, 83
bandpass, 57
bandstop, 57
Bilinear, 28
butterworth, 56
classical representation, 62
Coefficient of Variation, 74
coefficient of wavelet variation, 77
compressgif, 94
containment, 93
containment.pl, 21
Continuous Time Optimal Control, 49
control system, 61
COV, 74
csuthesis.sty, 94
CTOC, 49
developers, 19
difference equation, 12 39
differential equation solvers, 27
differentiator, 68
discrete, 6
DTOC, 48 81
Encapsulated Postscript, 104
engineering requirements, 67
EPS, 104
ESO, 46
Euler, 32
Extended State Observer, 46
extractClassInfo, 94
filters, 51
Fundamental building blocks, 12
Gao, 83
general controller block, 62
general feedback system, 62
<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>general representation</td>
<td>62</td>
</tr>
<tr>
<td>Gnuplot</td>
<td>12, 91</td>
</tr>
<tr>
<td>Graphical User Interface</td>
<td>3</td>
</tr>
<tr>
<td>Graphviz</td>
<td>92</td>
</tr>
<tr>
<td>GUI</td>
<td>3, 17</td>
</tr>
<tr>
<td>GUIDE</td>
<td>90</td>
</tr>
<tr>
<td>Hartmath</td>
<td>92</td>
</tr>
<tr>
<td>HasFactory</td>
<td>21</td>
</tr>
<tr>
<td>high-pass</td>
<td>57</td>
</tr>
<tr>
<td>improved general controller block</td>
<td>63</td>
</tr>
<tr>
<td>Java</td>
<td>7, 90</td>
</tr>
<tr>
<td>Java code</td>
<td></td>
</tr>
<tr>
<td>s2z</td>
<td>38</td>
</tr>
<tr>
<td>z2d</td>
<td>40</td>
</tr>
<tr>
<td>java2dot</td>
<td>13, 93</td>
</tr>
<tr>
<td>Least Mean Square</td>
<td>75</td>
</tr>
<tr>
<td>linear regression</td>
<td>75</td>
</tr>
<tr>
<td>LMS</td>
<td>75</td>
</tr>
<tr>
<td>low-pass</td>
<td>57</td>
</tr>
<tr>
<td>MacOS</td>
<td>7</td>
</tr>
<tr>
<td>Maple</td>
<td>2</td>
</tr>
<tr>
<td>Mathematica</td>
<td>2, 90</td>
</tr>
<tr>
<td>Matlab</td>
<td>2, 3, 90</td>
</tr>
<tr>
<td>maximum sample</td>
<td>71</td>
</tr>
<tr>
<td>minimum sample</td>
<td>71</td>
</tr>
<tr>
<td>Most Significant Bandwidth Threshold</td>
<td>69</td>
</tr>
<tr>
<td>MSBT</td>
<td>69</td>
</tr>
<tr>
<td>noise</td>
<td>67</td>
</tr>
<tr>
<td>NPID</td>
<td>4, 81</td>
</tr>
<tr>
<td>object-oriented</td>
<td>12</td>
</tr>
<tr>
<td>Octave</td>
<td>91</td>
</tr>
<tr>
<td>ode45</td>
<td>27</td>
</tr>
<tr>
<td>oo</td>
<td>12</td>
</tr>
<tr>
<td>optimization threshold</td>
<td>67</td>
</tr>
<tr>
<td>optimized</td>
<td></td>
</tr>
<tr>
<td>actively</td>
<td>66</td>
</tr>
<tr>
<td>overshoot</td>
<td>69</td>
</tr>
<tr>
<td>Parameterization</td>
<td>42</td>
</tr>
<tr>
<td>Perl</td>
<td>12, 91</td>
</tr>
<tr>
<td>PID</td>
<td>2, 42, 63</td>
</tr>
<tr>
<td>pre-filters</td>
<td>51</td>
</tr>
<tr>
<td>profiles</td>
<td>51</td>
</tr>
<tr>
<td>proportional-integral-derivative</td>
<td>42</td>
</tr>
<tr>
<td>Ptolemy</td>
<td>91</td>
</tr>
<tr>
<td>PtPlot</td>
<td>7</td>
</tr>
<tr>
<td>quantifying noise</td>
<td>70</td>
</tr>
<tr>
<td>rate of change</td>
<td>68</td>
</tr>
<tr>
<td>Runge-Kutta</td>
<td>30, 32</td>
</tr>
</tbody>
</table>
s-functions, 4
s2z, 36
SimBlocks, 13
SimBuffer, 17
SimFactory, 17 20
SimFunction, 13
Simple Euler, 28
SimSink, 13
SimSource, 13
Simulink, 3
standard deviation, 73
steady-state, 71
SysQuake, 2 4 90
Time Optimal Control, 70
Trapezoidal, 28 51
Tustin, 28
University of British Columbia, 28
Unix, 7
users, 19
variance, 73
Vim, 92
wavelet, 77
wavelet variance, 77
Windows, 7
Wolfram, 90