TOOLS AND TECHNIQUES FOR LOCATING AND STEERING PARALLEL SIMULATIONS THROUGH BIFURCATION POINTS.

A dissertation submitted to
Kent State University in partial fulfillment of the requirements for the degree of Doctor of Philosophy

by
Daniel M. Bennett
December, 2010
Dissertation written by
Daniel M. Bennett
B.S., The University Of Akron, 1986
M.S., Kent State University, 1990
Ph.D., Kent State University, 2010

Approved by
Paul A. Farrell, Co-Chair, Doctoral Dissertation Committee
Arden G. Ruttan, Co-Chair
Johnnie W. Baker, Members, Doctoral Dissertation Committee
John J. Portman
Lothar Reichel

Accepted by
Jonathan I. Maletic, Chair, Department of Computer Science
John R. D. Stalvey, Dean, College of Arts and Sciences
# TABLE OF CONTENTS

LIST OF FIGURES .................................................. ix

LIST OF TABLES .................................................... xiv

LIST OF ALGORITHMS ............................................. xvii

LIST OF CODE EXAMPLES ......................................... xvii

ACKNOWLEDGMENTS ................................................ xviii

1 Introduction ..................................................... 1
   1.1 Accomplishments ........................................... 2
   1.2 Organization of Thesis ................................... 3

2 Simulating Defects in Liquid Crystal Structures ................. 6
   2.1 Liquid Crystal Material .................................... 7
   2.2 A Physical Theory of Liquid Crystalline Material .......... 9
   2.3 A Numeric Model of Liquid Crystalline Material .......... 14
   2.4 Numerical Methods ......................................... 16
   2.5 Numerical Continuation .................................... 19
   2.6 The MPIPATH Program ..................................... 21
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.6 Data Types</td>
<td>206</td>
</tr>
<tr>
<td>D.7 Primitive Objects</td>
<td>207</td>
</tr>
<tr>
<td>D.7.1 ClientBase Class</td>
<td>207</td>
</tr>
<tr>
<td>D.7.2 ConnectionBase Class</td>
<td>209</td>
</tr>
<tr>
<td>D.7.3 GuiBase Class</td>
<td>210</td>
</tr>
<tr>
<td>D.8 Basic Objects</td>
<td>212</td>
</tr>
<tr>
<td>D.8.1 Connection Objects</td>
<td>213</td>
</tr>
<tr>
<td>D.8.2 Filter Objects</td>
<td>215</td>
</tr>
<tr>
<td>D.8.3 Client Objects</td>
<td>216</td>
</tr>
<tr>
<td>D.8.4 Other Objects</td>
<td>220</td>
</tr>
<tr>
<td>D.9 Derived Objects</td>
<td>221</td>
</tr>
<tr>
<td>D.10 Compound Objects</td>
<td>221</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>1</td>
<td>A Nematic Liquid Crystal, from Senyuk [66]</td>
</tr>
<tr>
<td>2</td>
<td>Constrained Nematic Liquid Crystal, from Senyuk [66]</td>
</tr>
<tr>
<td>3</td>
<td>A line defect in a nematic Liquid Crystal, recreated from Collings [21]</td>
</tr>
<tr>
<td>4</td>
<td>A Point Defect in a Nematic Liquid Crystal Simulation</td>
</tr>
<tr>
<td>5</td>
<td>Magnetic order parameter in a ferrous material which is in a demagnetized</td>
</tr>
<tr>
<td></td>
<td>(a) and magnetized (b) state.</td>
</tr>
<tr>
<td>6</td>
<td>The relationship between the radii of a molecule and the eigenvalues of</td>
</tr>
<tr>
<td></td>
<td>the order parameter.</td>
</tr>
<tr>
<td>7</td>
<td>Natural Parameter Continuation</td>
</tr>
<tr>
<td>8</td>
<td>Pseudo-arclength continuation from Castillo [19]</td>
</tr>
<tr>
<td>9</td>
<td>Data Decomposition from Farrell [28]</td>
</tr>
<tr>
<td>10</td>
<td>Shared Data from Farrell [28]</td>
</tr>
<tr>
<td>11</td>
<td>Parallel Performance of Algorithm 4 (Without Bifurcation Routines)</td>
</tr>
<tr>
<td>12</td>
<td>A Bifurcation Diagram in a Liquid Crystal Simulation from Baglama [8]</td>
</tr>
<tr>
<td>14</td>
<td>Calling PCIRBLEIGS from a Language other than C++</td>
</tr>
<tr>
<td>15</td>
<td>Typical output for PCIRBLEIGS</td>
</tr>
<tr>
<td>16</td>
<td>Time PARPACK to find the Minimum Eigenvalues for Different Sized Matrices</td>
</tr>
</tbody>
</table>
17 Time PCIRBLEIGS to find the Minimum Eigenvalues for Different Sized Matricies ............................................. 46
18 Iterations per Second for PARPACK to find the Minimum Eigenvalues for Different Sized Matricies .................. 47
19 Iterations per Second for PCIRBLEIGS to find the Minimum Eigenvalues for Different Sized Matricies ............... 48
20 Time for PCIRBLEIGS to find the Minimum Eigenvalues for a 22,500 × 22,500 Matrix. ................................. 49
21 Iterations per Second for PCIRBLEIGS to find the Minimum Eigenvalues for a 22,500 × 22,500 Matrix. ............. 49
22 Experiment 2 Results for PCIRBLEIGS (typical) .................. 51
23 Experiment 2 Results for PARPACK (typical) ................... 51
24 Experiment 3 Example Results ........................................ 52
25 Interaction between a Simulation and a Viewer in an STV Environment ......................................................... 57
26 Model Coupling, Checkpointing and Steering in an STV Environment ............................................................... 59
27 A Chromium Network to Render on a Four Display Wall ......................................................... 74
28 A Chromium Network With Three Sources Rendering on a Four Display Wall ............................................. 75
29 A Chromium Enabled Viewer for CUMULVS Rendering on a Four Display Wall ........................................... 76
30 Performance of the Parallel Viewer ........................................ 78
31 Performance of the Parallel Viewer, $2^{14}$ Through $2^{20}$ Atoms ......................................................... 78
<table>
<thead>
<tr>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>51</td>
<td>Two versions of the Frames Per Second Viewer, a GUI Version on the Left and a Text Version on the Right</td>
</tr>
<tr>
<td>52</td>
<td>Two versions of the scaview Viewer, A GUI Version on the Left and a Text Version on the Right</td>
</tr>
<tr>
<td>53</td>
<td>Output from Listing 4</td>
</tr>
<tr>
<td>54</td>
<td>The History Viewer Constructed with pyCUMUILVS</td>
</tr>
<tr>
<td>55</td>
<td>The Architecture of a Gateway Viewer System</td>
</tr>
<tr>
<td>56</td>
<td>A Web Browser Connected to the Gateway Viewer</td>
</tr>
<tr>
<td>57</td>
<td>A Web Browser Connected to the Gateway Viewer Displayed on a Cellular Phone</td>
</tr>
<tr>
<td>58</td>
<td>A GUI Interface for LIFE</td>
</tr>
<tr>
<td>59</td>
<td>A STV-GUI Object Hierarchy</td>
</tr>
<tr>
<td>60</td>
<td>A STV-GUI Object Hierarchy with Multiple Inheritance</td>
</tr>
<tr>
<td>61</td>
<td>A GUI Object to Display and Steer a Single Parameter</td>
</tr>
<tr>
<td>62</td>
<td>A GUI Interface for mpipath</td>
</tr>
<tr>
<td>63</td>
<td>The Interaction Among pyGUIObjects</td>
</tr>
<tr>
<td>64</td>
<td>A Reset GUI Steering Client for Life Using pyGUIObjects</td>
</tr>
<tr>
<td>65</td>
<td>The Temperature ($AK0$) vs Minimum Eigenvalue of the Jacobian ($\lambda_{\text{min}}$)</td>
</tr>
<tr>
<td>66</td>
<td>An Interface to Steer Path following Parameters</td>
</tr>
<tr>
<td>67</td>
<td>The Temperature ($AK0$) vs Minimum Eigenvalue of the Jacobian ($\lambda_{\text{min}}$) as a Result of User Steering</td>
</tr>
<tr>
<td>68</td>
<td>The Interface to the Temporary Memory Added to mpipath</td>
</tr>
</tbody>
</table>
Applying the Bisection Method to \textit{mpipath} \hspace{1cm} \textup{130}

Data From Applying the Secant Method to a Run of \textit{mpipath} to Find a Singular Point \hspace{1cm} \textup{131}

The Main Steering Interface for \textit{mpipath} \hspace{1cm} \textup{134}

The Interface for the Root Finding Segment of \textit{mpipath} \hspace{1cm} \textup{135}

The Interface for the Secant Method Segment of \textit{mpipath} \hspace{1cm} \textup{136}

The Interface Total \textit{mpipath} Control \hspace{1cm} \textup{136}

Finding the Dimension of the Null Space with PCIRBLEIGS \hspace{1cm} \textup{137}

Finding Paths Out of the Singular Point \hspace{1cm} \textup{138}

The First Step on a Path Out of the Singular Point \hspace{1cm} \textup{139}

The Path to the Next Singular Point \hspace{1cm} \textup{139}

A Discontinuous Break in the Path Out of a Singular Point \hspace{1cm} \textup{140}

A Parameter List and a Parameter Value List \hspace{1cm} \textup{207}
LIST OF TABLES

1  mpath Running Time in Seconds ................................. 26
2  Time and Iteration Comparison for PCIRBLEIGS and PARPACK . . . 44
# LIST OF ALGORITHMS

1. The Basic MPIPATH Algorithm .......................... 14
2. The MPIPATH Pathfollowing Algorithm ...................... 18
3. The final MPIPATH Algorithm ............................. 22
4. MPIPATH with Singular Point Detection. .................... 33
5. The Basic Viewer Loop .................................. 113
6. The Basic STV Loop .................................. 126
7. The STV Loop Modified for User Control .................. 126
8. The Original \textit{mpipath} Control Algorithm ................ 132
9. The Modified \textit{mpipath} Control Algorithm ................ 133
10. The Final Interactive Version of the \textit{mpipath} Control Algorithm .................. 134
LIST OF CODE EXAMPLES

1  Initializing CUMULVS .................................................. 63
2  Registering a Parameter and a Field in CUMULVS ...................... 64
3  Performing a Data Update in CUMULVS ................................. 65
4  Initializing the STV Connection in pyCUMULVS ....................... 103
5  A Program to Watch a Single Parameter ............................... 104
6  pyCUMULVS Calls to Steer a Parameter ............................... 105
7  XML Request to the Gateway Viewer .................................. 109
8  Data Returned by the Gateway Viewer .................................. 109
9  A Simple Client Built with the Extensible Viewer Library ............ 115
10 pyGUIObjects Code to Create a Scatter Plot .......................... 119
11 pyGUIObjects Connection Code ....................................... 119
12 pyGUIObjects Filter Code .............................................. 120
13 The Full Code for Life Reset .......................................... 121
14 A C++ CIRBLEIGS Example ........................................... 161
15 A C++/MPI PCIRBLEIGS Example .................................... 168
16 A C++ CTK Example for Parameters .................................. 193
17 A C++ CTK Example for Fields ....................................... 196
18 A Simple Python Example .............................................. 201
19 A Python Viewer to Restart an Application ........................... 202
<table>
<thead>
<tr>
<th>Code Fragment</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 A Code Fragment Using the ConnectionTest Object</td>
<td>213</td>
</tr>
<tr>
<td>21 A Code Fragment Using the Connection Object</td>
<td>214</td>
</tr>
<tr>
<td>22 A Code Fragment Using the StaticFilter Object</td>
<td>216</td>
</tr>
<tr>
<td>23 A Code Fragment Using the ParameterViewer Objects</td>
<td>217</td>
</tr>
<tr>
<td>24 A Code Fragment Using the StaticButtonSteerer Object</td>
<td>220</td>
</tr>
<tr>
<td>25 Code to Create an UpDown Object</td>
<td>222</td>
</tr>
<tr>
<td>26 Code to Create an UpDownLine Object</td>
<td>223</td>
</tr>
<tr>
<td>27 A Code Fragment which Employs an UpDownLine Object</td>
<td>223</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

I would like to begin by thanking my advisers Dr. Paul Farrell and Dr. Arden Ruttan. I have worked with and for the two of you for more than 20 years. While others have shaped who I am, the two of you have refined that person, sometimes through fire, to what I have become. I appreciate the knowledge you have imparted, the wisdom you have shared, and the support you have provided. For you I can express nothing other than respect, admiration, and the deepest gratitude. You are my teachers in the greatest sense of the word.

Dr. Baker, I believe you encouraged me to work on a PhD my first day at Kent State, and continued to do so until you signed the approval page of this document. Without such encouragement this task never would have been started, let alone accomplished. Thank you.

Dr. Reichel, your interest in the project, your participation in related conversations, and your patience when I detained your Friday evening companions are all greatly appreciated. Thank you for your time, knowledge and experience.

I gratefully acknowledge the support and encouragement of my colleagues at Edinboro University of Pennsylvania. Thank you to Eric who provided the motivation and encouragement to pursue this task. Thanks to Trish, Pam, Ellen and Corinne who constantly express confidence in my work, especially when I feel that none is warranted. Thanks to John, Rick, Emily and Frank who are willing to listen to questions about mathematics.
far removed from their areas of expertise, read an occasional paper, and nod approvingly when I try to express a mathematical thought. Thanks to Joe and Dave who prove to be great sounding boards. Thanks to the remaining members of the Department of Mathematics and Computer Science for your support and encouragement. Thank you to John, who tried to explain order parameters, and to Rich, Jim and Gerry for their encouragement. Finally, to Paula and Jason, you are the greatest, nothing more needs to be said!

To my family, Thanks! I appreciate your support, confidence and especially your patience. I can never make up for the summers, holidays and weekends spent in my office working on this task, but I will try! Amanda, I appreciate you support and encouragement. Rachel and Elizabeth, I love you both and appreciate you more than you can know. Mom, your belief in my abilities goes beyond justifiable reason but it allows me to reach the heights you expect. Dad, I miss you more than I can express. Michele, I love you more than anyone can express.

Finally, thank you to the National Science Foundation and to the Pennsylvania State System of Higher Education Faculty Professional Development Grant program for providing generous financial support.
CHAPTER 1

Introduction

Many scientific simulations are based upon differential equations or difference equations and involve following the state of the system as a parameter, such as temperature, changes. Such simulations may encounter singular points, or points where multiple solutions are possible. Without correctly detecting and analyzing such points, as well as carefully selecting the proper solution leading out of such a point, the computation may produce erroneous results.

The purpose of this project is to produce tools and techniques which will allow the creator of a parallel simulation to build a user interface to locate, analyze and steer the simulation through a singular point.

The work in this project employs an existing simulation \textit{mpipath}, which tracks the free energy in a liquid crystalline material as the temperature changes, as the primary example for the application of the tools developed in this work. While this simulation is described in Chapter Two, and modifications to the simulation are described in subsequent chapters, it should be emphasized that the purpose of this project is not to develop this program for liquid crystal simulation but only to use it as a vehicle for testing the tools developed here for exploring singular points.
1.1 Accomplishments

As documented in Chapter Seven, singular points encountered during the execution of \textit{mpipath} can be successfully located and characterized, and paths exiting from these singular points can be explored. To the author’s knowledge, this work represents the first successful application of steering techniques to such a task.

Steering a computation through a singular point is achieved by employing a number of tools developed for this project. These tools include:

- PCIRBLEIGS: a parallel, C++ based implementation of IRBLEIGS. IRBLEIGS is a MATLAB program for computing a few eigenpairs of a large sparse Hermitian matrix. CIRBLEIGS, discussed in Chapter Three, is a port of IRBLEIGS to C++, while PCIRBLEIGS provides a parallel implementation of CIRBLEIGS.

- The CUMULVS Toolkit: a library extension to the CUMULVS steering and visualization middle-ware system, with language bindings for C++ and python. This toolkit simplifies the task of creating programs which allow user interaction with parallel computations via CUMULVS.

- pyGUIobjects: a set of python objects which promote the rapid development of customized graphical user interfaces for simulations instrumented with CUMULVS.

Finally, a working system, which allows computational scientists to explore the behavior of \textit{mpipath} near a singular point has been developed. This system includes a comprehensive user interface for monitoring and steering the computation, and is enhanced by a visualization system which permits interactive examination of the three
dimensional data produced by the simulation. While the specific details of this system, such as the elements of the user interface, are specialized for use with mpipath, the techniques involved are applicable to many problems, and are likely to be employed anytime steering through singular points is attempted. This system has proven useful and has formed the basis for deeper exploration of the behavior of mpipath.

1.2 Organization of Thesis

The remainder of this document is organized as follows:

Chapter Two provides a brief overview of liquid crystals along with the mathematics and high level algorithms involved in simulating a region of a liquid crystalline material in the face of temperature change. An overview of the main application for this project, mpipath, is provided and the state of this simulation at the beginning of the work described here is presented. The chapter concludes with an examination of the performance of mpipath.

Chapter Three provides a brief overview of singular points, including bifurcation points, and discusses tools to locate and classify such points. This chapter also introduces PCIRBLEIGS a parallel C++ library for locating a few minimal or maximal eigenvalues in a large, sparse, symmetric, real matrix. PCIRBLEIGS was created for this project. A number of numerical examples demonstrating the performance of PCIRBLEIGS complete this chapter.

Chapter Four introduces steering and visualization middle-ware (STV) and the need for such software. This chapter provides a brief overview of STV software, including functionality and criteria for selection. The selection of the CUMULVS package for this
project is discussed. Details of instrumenting a number of parallel simulations with CUMULVS are given and suggestions for improving CUMULVS are provided.

Field Viewers, or viewers specifically designed to present field data are discussed in Chapter Five. The chapter begins by examining the viewers provided with CUMULVS. Two viewers created for this project are discussed, a Chromium enabled viewer for CUMULVS and a viewer designed especially for mpipath. The chapter concludes with the introduction of a toolkit to assist viewer implementers, which was designed as a part of this project.

Chapter Six presents parameter viewers. Once again, the viewers provided with CUMULVS are examined. To demonstrate the utility of parameter viewers, a set of simple viewers written to support the project are presented. A Python programming language interface to the toolkit introduced in the previous chapter is presented along with example viewers which employ this toolkit. The Python interface is used to produce an extensible viewer that is a framework which greatly simplifies the creation of specific parameter viewers. The extensible viewer is discussed together with a number of examples of its application. Finally, an object oriented hierarchy, which combines CUMULVS with a graphical user interface library is presented, along with an example application employing this hierarchy.

Chapter Seven presents the application of the tools developed in the previous chapters to the mpipath program to detect, analyze and steer this computation through a singular point. This chapter discusses changes made to mpipath to accomplish this task. A special purpose parameter viewer to control mpipath is presented. The chapter includes
an example where the interface is used to steer the computation through a bifurcation point.

Chapter Eight provides a summary of the entire project, a discussion of future directions and a set of conclusions.

This document has a number of appendices. These appendices are working versions of the user manuals for a number of the software products produced to complete the work described herein.
CHAPTER 2

Simulating Defects in Liquid Crystal Structures

The purpose of this project is to explore tools and techniques for steering a parallel simulation through a singular point. This chapter examines *mpipath*, a parallel simulation which involves tracking an equilibrium configuration of a liquid crystal model as the temperature of the system changes. This simulation has been developed at Kent State University [6,76] and will serve as the primary simulation for this project.

While the creation of this simulation is not the principal concern of this project, the controlling algorithm of the simulation must be altered to accomplish the project’s goals, therefore a basic understanding of the simulation is required. This chapter discusses the theoretical background for *mpipath* including a brief overview of the physics of liquid crystalline materials and an examination of the basic mathematical model and algorithms employed in this calculation. The purpose of this chapter is to provide an insight into the processes involved with such a simulation and describe the mathematical tools employed. In this chapter, the primary algorithm for *mpipath* is developed. This algorithm will be altered in the next chapter to allow the detection of singular points, and radically changed in chapter seven to permit interactive exploration of such points.

The background material in this chapter is sufficient to provide a basic understanding of the processes involved in *mpipath* in particular, and a class of simulations which encounter singular points in general. The intention of the chapter is to provide an overview
of this type of problem rather than a detailed exploration of the physics and numerical
methods discussed. Readers interested in further information regarding these topics are
referred to the individual works cited.

2.1 Liquid Crystal Material

A liquid crystal is matter in a state between liquid and crystal. Material in a liquid
crystal state possesses properties of both a liquid, such as fluidity, and a crystal, such as
molecular alignment [4]. Liquid crystalline material further possesses properties which are
unique to this state of matter. These include “very specific electrooptical phenomena,
which have no corresponding analogues in solids or in isotropic liquids [66].” These
properties can be controlled by weak electric or magnetic fields, and allow for the creation
of liquid crystal displays.

Liquid crystals can exist in a large number of phases including cholesteric, smectic,
and nematic, the latter being of primary phase of interest for this work. In the nematic
phase the molecules tend to align according to a direction vector called a director. The
director may vary locally, but tends to be uniform in unconstrained material. Figure 1
shows an unconstrained nematic liquid crystal. It should be noted that the sign of the
director is not important, or for a given director \( \vec{n} \), \( \vec{n} \equiv -\vec{n} \) as there is no difference in
the two ends of the molecule. In Figure 2 the local director changes due to deformations
imposed by external forces.

Defects may occur in crystalline structures when the regular arrangement of the un-
derlying molecular structure is interrupted. In nematic liquid crystals defects include
line defects (or disclinations) and point defects (or hedgehogs). Defects are identified by
Figure 1: A Nematic Liquid Crystal, from Senyuk [66]

Figure 2: Constrained Nematic Liquid Crystal, from Senyuk [66]
discontinuous changes in the director, and are produced by a number of influences such as surface geometry, phase transitions, and external factors. [4, 47]. Devices employing liquid crystals often generate a boundary alignment mechanically forcing such a defect to occur [28]. Defects are classified using topological methods which are beyond the scope of this document. Interested readers are directed to [4] and [67].

Figure 3 represents an artistic depiction of a line defect while Figure 4 displays a point defect in a more sophisticated simulation of a liquid crystal created by opposing boundary conditions. In this image, the structure is unable to maintain the alignment of the boundary conditions in the center and thus would have singularity in the director field. Defects form in different configurations and different locations depending on conditions such as temperature.

![Figure 3: A line defect in a nematic Liquid Crystal, recreated from Collings [21]](image)

2.2 A Physical Theory of Liquid Crystalline Material

The study of liquid crystalline material is a branch of condensed matter physics. Scientists in this field are concerned with “understanding and explaining the new laws that emerge when many particles interact [67].” In such a study, it is not practical nor
necessarily desirable to study all of the basic elements of the system under consideration. Instead, one attempts to locate the order parameters, which are the critical variables describing the order in the system. For example, in a magnet, one considers the local direction of magnetization, represented by a vector, which points in the direction of magnetization, or “north”.

The order parameter describes the order of the material locally and can globally describe the state of the entire material. In Figure 5 two different states of a magnetic material are displayed. The leftmost image represents material in a demagnetized state. In this case, the mean of the order parameter is approximately zero. This represents a system which is unordered or random. In the image on the right, the mean of the local order parameters is nonzero, which indicates the system is more ordered or in a magnetized state.

An order parameter field is a mapping from a position within a material to the order
Figure 5: Magnetic order parameter in a ferrous material which is in a demagnetized (a) and magnetized (b) state.

parameter representing the material near that position. For a slab \( \Omega = \{ (\alpha, \beta, \gamma) : 0 \leq \alpha \leq a, 0 \leq \beta \leq b, 0 \leq \gamma \leq c \} \), the order parameter field \( Q = Q(p) \), where \( p = (\alpha, \beta, \gamma) \in \Omega \), represents the state of the slab.

The selection of an order parameter for a material is a difficult task for the physical scientist, who must consider the material, as well as the states and transitions which are to be studied [67]. With many liquid crystal models the director is an appropriate choice as the order parameter. The choice of the director in this role assumes that molecules are cylindrical, or uniaxial. Models which employ the director as the order parameter are sufficient to account for point defects, but fail to allow for line defects. When considering line defects, one must allow for spatial ordering, while even more general models allow for biaxial fields [20,55]. Biaxial models assume that the order parameter can be represented by an ellipsoid where all three axes have different values.

The order parameter employed for biaxial models is a traceless symmetric tensor. This tensor is given in Equation (1), where \( q_i \) are real valued functions on \( \Omega \) and are based on the material [6]. The eigenvectors of \( Q \) give the axis of orientation, the eigenvalues provide the radii, and the principal eigenvector plays the role of the director. Please see Figure 6.
Figure 6: The relationship between the radii of a molecule and the eigenvalues of the order parameter.

\[
Q(p) = q_1(p) \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \end{pmatrix} + q_2(p) \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \\
+ q_3(p) \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} + q_4(p) \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \\
+ q_5(p) \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}
\]

(1)

Landau theory is an attempt to describe second order phase transitions in systems. A second order phase transition occurs when a system changes to a new state in a continuous manner [23]. This theory relates the order parameter to the free energy, or the amount of
energy in a system which is available to do work. Landau demonstrated that this theory applies equally well to magnets, superconductors and superfluids [67]. De Gennes later employed Landau theory, with the selection of the traceless symmetric tensor above as the order parameter to form the Landau-De Gennes theory for nematic liquid crystals [55]. A formulation of the Landau-De Gennes equations is given in Equations (2)-(5). The free energy \( F(Q,T) \) is given by

\[
F(Q,T) = F_{\text{vol}}(Q,T) + F_{\text{surf}}(Q) = \int_\Omega f_{\text{vol}}(Q,T) dV + \int_{\partial\Omega} f_{\text{surf}}(Q) dS,
\]

where \( T \) is equivalent to the temperature of the liquid crystal, and \( Q \) and \( \Omega \) are as defined above. Furthermore:

\[
f_{\text{vol}}(Q,T) = \frac{1}{2} L_1 Q_{\alpha\beta,\gamma} Q_{\alpha\beta,\gamma} + \frac{1}{2} L_2 Q_{\alpha\beta,\beta} Q_{\alpha\gamma,\gamma} + \frac{1}{2} L_3 Q_{\alpha\beta,\gamma} Q_{\alpha\gamma,\beta}
\]

\[
+ \frac{1}{2} A \ \text{trace}(Q^2) - \frac{1}{3} B \ \text{trace}(Q^2) + \frac{1}{4} C \ \text{trace}(Q^2)^2
\]

\[
+ \frac{1}{5} D \ \text{trace}(Q^2) \text{trace}(Q^3) + \frac{1}{6} M \ \text{trace}(Q^2)^3 + \frac{1}{6} M' \ \text{trace}(Q^3)^2
\]

uses the conventions that summation over repeated indices is implied and indices separated by commas represent partial derivatives. See Equation (4) for an example.

\[
Q_{\alpha\beta,\gamma} Q_{\alpha\gamma,\beta} = \sum_{\alpha=1}^{3} \sum_{\beta=1}^{3} \sum_{\gamma=1}^{3} \frac{\partial Q[\alpha,\beta]}{\partial x_\gamma} \cdot \frac{\partial Q[\alpha,\gamma]}{\partial x_\beta}.
\]

The parameters \( L_1, L_2 \) and \( L_3 \) are elastic constants, and \( A, B, C, D, M \) and \( M' \) are bulk constants. The bulk parameter \( A \) is assumed to have the form \( A = A_0 (T - T_0) \), where \( A_0 \) and \( T_0 \) are constants. In this thesis we take \( T_0 = 0 \) and \( A_0 = 2 \) which gives \( T = \frac{1}{2} A \). Moreover, assume

\[
f_{\text{surf}}(Q) = W \ \text{trace}((Q - Q_0)^2),
\]
where $W$ is a constant and the tensor $Q_0$ is determined by the boundary conditions for
the functions $q_i$.

Landau-De Gennes theory relates a physical description of liquid crystalline material, the order parameter, to the free energy in this material and provides a foundation upon which a model of liquid crystal material can be constructed. This model allows for the study of both point and line defects and allows for the assumption that liquid crystal material may be biaxial. The next section discusses how such a model can be implemented numerically. Full details of the Landau-De Gennes theory can be found in [32].

2.3 A Numeric Model of Liquid Crystalline Material

The purpose of mpipath is to provide a platform for studying defects in liquid crystalline material as the temperature is allowed to change. Algorithm 1 outlines the basic operation of mpipath. This section focuses on the calculation of $Q_i = F(Q_{i-1}, T_i)$.

**Algorithm 1** The Basic MPIPATH Algorithm

Interpolate $Q_{surf}$ to form $Q_{guess}$

Calculate $Q_0 = F(Q_{guess}, T_{start})$ such that $Q_0$ is the minimal energy equilibrium state.

Let $\Delta T = \frac{T_{end} - T_{start}}{\text{steps}}$

for $T = T_{start}$ to $T_{end}$ by $\Delta T$ do

Calculate $Q_i = F(Q_{i-1}, T)$ such that $Q_i$ is the minimal energy equilibrium state.

end for

The minimal energy equilibrium state is found at each stage by solving the Euler-Lagrange equations for Equation (2). The Euler-Lagrange equations are differential equations whose solution is a set of values for which the original function is stationary, or
has achieved a local maximum or minimum [61]. The Euler-Lagrange equations when applied to an integral such as 
\[ J = \int_{a}^{b} F(y, y', x) dx, \]
produce an equation in the form
\[ 0 = \frac{\partial F}{\partial y} - \frac{d}{dx} \left( \frac{\partial F}{\partial y'} \right). \]
Applying the Euler-Lagrange equations ensures that the system obeys the principle of minimum energy, a restatement of the second law of thermodynamics.

Application of the Euler-Lagrange Equation to the Landau-De Gennes formulation yields Equations (6)-(7)

\[ \frac{\partial f_{\text{vol}}(Q, T)}{\partial Q} - \frac{\partial}{\partial x_1} \left( \frac{\partial f_{\text{vol}}(Q, T)}{\partial Q_{x_1}} \right) - \frac{\partial}{\partial x_2} \left( \frac{\partial f_{\text{vol}}(Q, T)}{\partial Q_{x_2}} \right) - \frac{\partial}{\partial x_3} \left( \frac{\partial f_{\text{vol}}(Q, T)}{\partial Q_{x_3}} \right) = 0 \quad (6) \]
on \Omega, and

\[ \frac{\partial f_{\text{surf}}(Q)}{\partial Q} + \frac{d_{x_1}}{d_n} \frac{\partial f_{\text{vol}}(Q, T)}{\partial Q_{x_1}} + \frac{d_{x_2}}{d_n} \frac{\partial f_{\text{vol}}(Q, T)}{\partial Q_{x_2}} + \frac{d_{x_3}}{d_n} \frac{\partial f_{\text{vol}}(Q, T)}{\partial Q_{x_3}} = 0 \quad (7) \]
on \partial \Omega, where \( \frac{d_{x_1}}{d_n}, \frac{d_{x_2}}{d_n}, \frac{d_{x_3}}{d_n} \) are obtained from Green’s Theorem and are the direction cosines of the outward normal of the boundary \( \partial \Omega \) [6].

The finite difference method provides a means for approximating the solution to differential Equations such as (6) and (7). This method approximates the derivatives of a function through algebraic expressions, given a function \( f(x) \), \( f'(x_i) \approx \frac{f(x_{i+1})-f(x_{i-1})}{h^2} \) and \( f''(x_i) \approx \frac{f(x_{i+1})-2f(x_i)+f(x_{i-1})}{h^2} \) for sufficiently small \( h \). An equation such as \( y'' = f(x, y, y') \) is approximated by

\[ f \left( x_i, y_i, \frac{y_{i+1} - y_{i-1}}{2h} \right) = \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}, \]

\[ \{ i = 1, 2, ..., n - 1 \} \]
\[ y_0 = \alpha, y_n = \beta \]

where \( \alpha = x_0 < x_1 < ... < x_n = \beta \) and \( x_i = x_0 + ih \). Such a computation relies on a boundary conditions \( (f(x_0) = \alpha, f(x_n) = \beta) \) [44].

This solution gives rise to \( (n - 1) \) equations in \( (n - 1) \) unknowns \( (y_i) \) and can be solved using Newton’s method. It can be shown that the discretization error associated
with this finite difference approximation is \(O(h^2)\). Therefore, decreasing \(h\) to improve accuracy increases the problem size. A \(20 \times 20 \times 20\) cube with 5 variables from \(Q\) at each point will have \(4 \times 10^4\) equations in \(4 \times 10^4\) unknowns.

Equation (8) represents the final version of the discretized Euler Lagrange equation applied to the Landau-De Gennes equation. In this equation, \(Q \in \mathbb{R}^n\) is considered a variable, while \(T\) is a parameter. The parameter, representing temperature, is set, and numerical methods are applied to Equation (8) to find valid values of the variable, which represents the properties of the liquid crystal material.

\[
G(Q, T) = 0. \tag{8}
\]

2.4 Numerical Methods

Newton’s Method, mentioned above, is one of several numerical methods employed in \textit{mpipath}. This section provides a brief overview for the methods employed. Full treatment can be found in standard numerical analysis texts such as [44,61].

Newton’s method is an iterative method employed to find zeros, or roots of a function. Given a function \(f(x)\), the first derivative of the function \(f'(x)\), and an initial guess \(x_0\), the statement \(x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}\) for \(n = 1, 2, \ldots\) describes this method. If \(x_0\) is sufficiently “near” a root \(x_r, f(x_r) = 0\), the method will converge to \(x_n \approx x_r\) in “reasonable” time. The definitions of “near” and “reasonable” depend on the function, and the specific root under consideration. This method encounters difficulties near local critical points, and implementations may require intervention to successfully compute a desired root.

Newton’s Method can be employed for solving systems of \(n\) equations in \(n\) unknowns.
as well. If \( x = (a_1, a_2, ..., a_n) \) and

\[
F(x) = \begin{bmatrix}
f_1(x) \\
f_2(x) \\
... \\
f_n(x)
\end{bmatrix}.
\]

In this version of the method, the Jacobian matrix

\[
J(x) = \begin{bmatrix}
\frac{\partial f_1(x)}{\partial a_1} & \cdots & \frac{\partial f_1(x)}{\partial a_n} \\
\vdots & \ddots & \vdots \\
\frac{\partial f_n(x)}{\partial a_1} & \cdots & \frac{\partial f_n(x)}{\partial a_n}
\end{bmatrix}
\]

serves in place of \( f'(x) \) in the corresponding univariate version. Equation (9) is a statement of the multivariate version of Newton’s Method. In general, it is computationally difficult to calculate \( J(x_n)^{-1} \) thus the an iterative method such as MINRES is generally employed to solve the corresponding system:

\[
x_{n+1} = x_n - J(x_n)^{-1}F(x_n).
\] (9)

The MINRES, or minimal residual method, solves equations in the form \( Ax = b \), where \( A \) is an invertible matrix of size \( m \times m \). Both \( x \) and \( b \) are vectors of size \( m \) and \( b \) is known and has been normalized [33]. To employ this method, let \( z_n = x_{n+1} - x_n \) from (9). Equation (10) represents the final equation to be solved with this method. Utilizing this equation, the new mpipath algorithm is given in Algorithm 2

\[
J(x_n)z_{n+1} = -F(x_n)
\] (10)
Algorithm 2 The MPIPATH Pathfollowing Algorithm

Let $G(Q, T)$ be the discretized Euler-Lagrange equations applied to the Landau-de-Gennes model.

Interpolate $Q_{surf}$ to form $Q_{guess}$

Form the Jacobian $J(Q_{guess}, T_{start})$

repeat

Use MINRES to solve $J(Q_{guess}, T_{start})(Q_0 - Q_{guess}) = -G(Q_{guess}, T_{start})$

until $||Q_0 - Q_{guess}|| \approx 0$

Let $\Delta T = \frac{T_{end} - T_{start}}{\text{steps}}$

for $T = T_{start}$ to $T_{end}$ by $\Delta T$ do

Form the Jacobian $J(Q_i, T)$

repeat

Use MINRES to solve $J(Q_i, T)(Q_{i+1} - Q_i) = -G(Q_i, T)$

until $||Q_{i+1} - Q_i|| \approx 0$

end for
Newton’s method is a powerful technique which is frequently employed with excellent results in many settings. The method, however, fails under a number of circumstances. It can be shown that for a reasonable set of assumptions, this method will converge if the initial guess is sufficiently close to a root. These assumptions include restrictions on the first derivative, that it is non-zero near the root and that it exists. When the first derivative is zero, the function has reached a local extreme point, and the method may fail. The same is true if the function approaches an asymptote, or the value of the first derivative approaches infinity. In this case, the method can stall, as the second term in \(x_n - \frac{f(x_n)}{f'(x_n)}\) will approach zero. These two restrictions play an important role in *mpipath* and require the introduction of path following techniques, also referred to as numerical continuation.

2.5 Numerical Continuation

The approach for computing new values of the parameter \(T\), implied in Algorithm 2 involves selecting a value, \(\Delta T = (T_{\text{end}} - T_{\text{start}})/\text{steps}\), and computing \(T_n = T_{\text{start}} + n \times \Delta T\). This approach is pictured in Figure 7 and is known as natural parameter, or zeroth-order continuation. For the example pictured in this figure, values of \(T\) near \(T_a\) correspond to relatively low values of the derivative and produce reasonably accurate initial points which are fairly close to the curve. In this case values of \(T\) near \(T_p\) result in initial guesses much further from the curve and thus present potential problems for Newton’s method.

In the two-dimensional case above, natural parameter continuation encounters problems when the slope of the tangent line approaches a vertical line. It is clear that small changes in the parameter will result in large changes in the system, thus a fixed step size
for the parameter is a poor choice. This is true in the multidimensional case, where the tangent is determined by the Jacobian. When the Jacobian becomes singular, or cannot be inverted, the computation is approaching a singular point, which will be discussed in the next chapter. As a computation approaches a singular point a different method of path following is required.

Pseudo-arclength continuation attempts to solve the problems of natural parameter continuation by introducing a new parameter based on the length of the path. This pseudo-arclength parameterizes the curve by introducing the new parameter $s$, where both $Q$ and $T$ are a function of $s$. This is pictured in Figure 8. Solving the system $F(Q(s,T), T(s)) = 0$ by natural parameter continuation on $s$ will yield a new point on the path. By augmenting the Jacobian with the tangent, a nonsingular matrix $F_x$ is
formed, and thus natural parameter continuation on $s$ is possible. For details on pseudo-arclength continuation see [25, 46]. Pseudo-arclength is more computationally expensive than natural parameter continuation and should only be used when required.

![Figure 8: Pseudo-arclength continuation from Castillo [19]](image)

2.6 The MPIPATH Program

The original mpipath program is a C based implementation of Algorithm 3 using the Message Passing Interface (MPI) for parallel communications. This program is designed to operate in a batch type environment, where all input and output, including intermediate results are stored in files. Each intermediate output file serves as a checkpoint, and the program can be restarted from any file. The choice of continuation method is hard coded into the program, and changing methods requires recompilation and a restart. Diagnostic messages, including convergence rates, are written to the standard output, providing the user with some level of knowledge concerning the progress of the computation.
**Algorithm 3** The final MPIPATH Algorithm

Let $G(Q,T)$ be the discretized Euler-Lagrange equations applied to the Landau-de-Gennes model.

Interpolate $Q_{surf}$ to form $Q_{guess}$

Form the Jacobian $J(Q_{guess}, T_{start})$

repeat

Use MINRES to solve $J(Q_{guess}, T_{start})(Q_0 - Q_{guess}) = -G(Q_{guess}, T_{start})$

until $||Q_0 - Q_{guess}|| \approx 0$

$T = T_{start}$

while $T < T_{end}$ do

Form the Jacobian $J(Q_i, T)$

repeat

Use MINRES to solve $J(Q_i, T)(Q_{i+1} - Q_i) = -G(Q_i, T)$

until $||Q_{i+1} - Q_i|| \approx 0$

Compute $\Delta T$ using either natural parameter or pseudo-arclength continuation

$T = T + \Delta T$

end while
The main data structure for mpipath is the discretized data representing the liquid crystal volume. Each point contains the five data values of the tensor given in Equation (1). This data is distributed across processes such that each process contains a slab of the data. Each slab consists of all data in the XY plane for a range of Z values (see Figure 9.) Computations are based on the nearest neighbor, so each process may require data from the processes above or below. To minimize communications, ghost copies of the boundary slabs are stored in the neighboring processes (see Figure 10.) With this decomposition, each process stores approximately $n^3/p$ data points, where $n$ is the dimension of the cube and $p$ is the number of processes. Each iteration, a process must compute the new value of its $n^3/p$ data points and exchange the $n^2$ slab with the process above and below. [28]. A new implementation of mpipath is under development with a more general data decomposition scheme, however, this should have very little impact on the work described herein.

![Figure 9: Data Decomposition from Farrell [28]](image)
The primary interest in parallelizing mpipath is to provide sufficient memory for data associated with the problem. While every effort has been made to produce efficient code, the primary consideration is a working product. The current data distribution limits the effective number of processes to \( n \) when working on an \( a \times b \times n \) slab of data. Future versions of this code should eliminate this restriction. In order to provide a benchmark for future reference, a simple performance test was conducted. This test involved an implementation of Algorithm 4 presented in the next chapter, with the bifurcation routines removed. Timing runs were conducted on a \( 32 \times 32 \times 32 \) slab with an initial starting temperature of 5. The initial boundary conditions were radial, or all elements point toward the center. The test was conducted for \( p = 2^i, i = \{0,...,5\} \) processes, and each run was stopped when the minimum eigenvalue of the Jacobian became negative. All code produced essentially the same solution, finding a singular point occurring at a temperature just below 2.94. The results of this experiment are presented in Figure 11 and
Table 1. It can be seen that both mpipath and the eigenfinder (PARPACK) do scale well with the number of processes. Full explanation of the environment for the experiment is described in the next chapter.

Figure 11: Parallel Performance of Algorithm 4 (Without Bifurcation Routines)

As seen in Algorithm 3, the majority of the processing in mpipath is matrix and vector based. The matrix is sparse as each equation in the discretization relies on the 26 nearest neighbors, and each point contains 5 tensor values (see Equation (1)), thus each row contains no more than $5 \times 26$ values. To conserve memory, a matrix-free representation is employed, where the entries are not stored, but computed as needed [28]. Custom routines are provided to perform matrix-matrix, matrix-vector and vector-vector operations. These routines are modeled on the BLAS library.
Table 1: *mpipath* Running Time in Seconds

<table>
<thead>
<tr>
<th>Processes</th>
<th>Total Time</th>
<th>Solver</th>
<th>Eigenfinder</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3240.81</td>
<td>1382</td>
<td>1843</td>
</tr>
<tr>
<td>2</td>
<td>1339.98</td>
<td>656</td>
<td>632</td>
</tr>
<tr>
<td>4</td>
<td>698.17</td>
<td>337</td>
<td>353</td>
</tr>
<tr>
<td>8</td>
<td>368.71</td>
<td>184</td>
<td>182</td>
</tr>
<tr>
<td>16</td>
<td>193.09</td>
<td>93</td>
<td>95</td>
</tr>
<tr>
<td>32</td>
<td>183.61</td>
<td>117</td>
<td>62</td>
</tr>
</tbody>
</table>

The original method for exploring a liquid crystal configuration using this program requires the user to create a file containing the starting configuration, set the path following method to natural parameter continuation and run the program. The user monitors the diagnostic output until the number of iterations of the Newton solver becomes too large. This indicates that natural parameter continuation has begun to encounter a problem and it is time to switch continuation methods. At this point, the user kills the program, changes the continuation method, recompiles, and restarts the computation from a save file. The process is continued until the computation is complete, at which point the output files are analyzed and processed for visualization.

The implementation of Algorithm 3 is the starting basis for the work described herein. This implementation of *mpipath* is capable of tracking the configuration of a liquid crystal slab through temperature changes. It requires user attention to monitor the progress of the computation and occasional restarts to handle special conditions which occur during
the execution. A number of simple modifications can be made to make the restart process more simple, such as adding control flags to the initialization file to specify parameters such as the continuation method to employ. These additions, however, would only handle part of the need for user intervention. A number of parameters needed for control of the process, such as limits for iteration counts and tolerances depend on interpretation and require human intervention. Singular points, which are described in the next chapter, present even greater challenges to the computation, and proper handling of such points demands human intervention.
CHAPTER 3

Bifurcation Points

In a dynamic system, a bifurcation occurs when a small smooth change in a parameter results in multiple solutions to the system. The parameter value where this occurs is called a bifurcation point. Bifurcations arise in systems of differential equations as well as systems of difference equations [22,69]. This work is primarily concerned with locating, identifying, and steering a scientific computation through bifurcation points encountered as a parameter changes. The example application selected is mpipath where the parameter varied represents temperature.

Bifurcations within systems are frequently depicted in a bifurcation diagram. In this diagram, the parameter is normally treated as the independent variable and a value, representing the state of the system, is depicted as a dependent variable. Figure 12 is a bifurcation diagram for a liquid crystal simulation. In this diagram the independent variable is the parameter representing temperature $T$, and the dependent variable is the minimal eigenvalue of the Jacobian. A bifurcation point occurs at $T = 0.3012$, resulting in four solutions or “paths” out of the bifurcation point. Each of these paths represents different a equilibrium state of the liquid crystal model, however only paths 1 and 2 represent stable solutions. This work was performed on a two dimensional liquid crystal model and is fully documented in [8].
The previous example illustrates the need to detect and correctly deal with bifurcation points in physical simulations. In this example, paths 3 and 4 represent physically impossible solutions, which should be detected and avoided if useful results are to be obtained. Numerically there is little difference between these solutions and each is likely to be selected. It is well known that Newton’s method will find different roots based on minor changes in the initial value, and the continuation methods discussed in the previous chapter produce initial guesses which are “close to” the current path. At a bifurcation point, all paths converge, and an unguided computation will select an arbitrary path which it will follow.
3.1 Definitions from Linear Algebra

Several definitions from linear algebra are required for the proper treatment of bifurcation points. These definitions are taken from [41] unless otherwise noted.

**Definition 1.** An $n \times n$ matrix $A$ is nonsingular, if there exists another $n \times n$ matrix $B$, such that $AB = BA = I$. If no such $B$ exists, then $A$ is singular.

**Definition 2.** The number of vectors in a basis for a vector space $V$ is the dimension of $V$, denoted by $\dim(V)$.

**Definition 3.** For any matrix $A$, the range of $A$, denoted by $\mathcal{R}(A)$, is the set of all linear combinations of the columns.

**Definition 4.** For a matrix $A$, the null space of $A$, denoted by $\mathcal{N}(A)$, is the set of all solutions to the homogeneous equation $Ax = 0$.

**Definition 5.** The algebraic multiplicity of an eigenvalue $\lambda$ is the multiplicity of the corresponding root $\lambda$ of the characteristic polynomial of $A$.

**Definition 6.** The geometric multiplicity of an eigenvalue $\lambda$ is the number of linearly independent eigenvectors associated with $\lambda$ for a matrix $A$.

3.2 Detection of Bifurcation Points

Bifurcation points are a special case of singular points. Singular points occur when the Jacobian becomes singular, or in some sense, the tangent to the curve is undefined. Singular matrices have a number of properties, which include: *The matrix $A$ is singular, if and only if at least one eigenvalue of $A$ is 0* [41]. This property can be exploited to
both locate and classify a singular point as either a limit point or a bifurcation point, as well as to select which path following method to employ.

Let $G(Q, T)$ be a dynamic system, with $Q \in \mathbb{R}^n$ and $T \in \mathbb{R}$. Let $(Q_0, T_0)$ be a solution to $G(Q, T) = 0$. Let $G_Q^0 \equiv G_Q(Q_0, T_0)$ and $G_T^0 \equiv G_T(Q_0, T_0)$ be the Frechet derivative with respect to $Q$ and $T$ respectively. The point $(Q_0, T_0)$ can be classified as follows [6,45]:

1. If $\dim(\mathcal{N}(G_Q^0)) = 1$ and $G_T^0 \notin \mathcal{R}(G_Q^0)$ then $(Q_0, T_0)$ is a simple limit point.

2. If $\dim(\mathcal{N}(G_Q^0)) = 1$ and $G_T^0 \in \mathcal{R}(G_Q^0)$ then $(Q_0, T_0)$ is a simple bifurcation point.

3. If $\dim(\mathcal{N}(G_Q^0)) > 1$ and $G_T^0 \notin \mathcal{R}(G_Q^0)$ then $(Q_0, T_0)$ is a multiple limit point.

4. If $\dim(\mathcal{N}(G_Q^0)) > 1$ and $G_T^0 \in \mathcal{R}(G_Q^0)$ then $(Q_0, T_0)$ is a multiple bifurcation point.

Each of these cases is illustrated in Figure 13. Classifying a point as a single limit point (case 1) is important as it is possible for $mpipath$ to simply step through such a point on the path using pseudo-arclength continuation. For the purposes of $mpipath$ there is little difference between cases 2, 3 and 4, other than informing the choice of method for computing tangents to outgoing paths.

Algorithm 3 can now be modified, as shown in Algorithm 4, to detect when the computation approaches singular points. Given this information the program can select an appropriate continuation method, or, if a bifurcation or multiple limit point is detected, switch to a mode where such points are dealt with. This modification assumes the existence of routines to deal with bifurcation points, which will be discussed next, and
the ability to find the minimum eigenvalue(s) of a matrix, the discussion of which will conclude this chapter.

3.3 Algebraic Bifurcation Equations

Once a singular point has been detected, the algebraic bifurcation equations (ABEs) and limit point bifurcation equations (LPBEs) provide a mechanism for identifying the paths out of the bifurcation point. These equations take the vectors comprising the null space of the Jacobian and the tangent, with respect to the parameter, at the singular point and produce tangent vectors for each outgoing path. Let $\{\theta_j\}_{j=1}^m$ denote the vectors of $\mathcal{N}(G^0_Q)$. Equations (11), (14) and (15) are the algebraic bifurcation equations, while equations (12), (13) and (14), where $\ddot{T}_{\text{sing}}$ is the second derivative of $T$ at the singular point, represent the limit point bifurcation equations. It should be noted that all values for these equations can be computed explicitly within mpipath. Full details of the ABEs
Algorithm 4 MPIPATH with Singular Point Detection.
Set \( Tol_{small} \) and \( Tol_{large} \)

Form \( G(Q, T) \) and find \( Q_0 \) as in Algorithm 3.

\[
\text{for } T = T_{start}, T < T_{end} \text{ do }
\]

Find \( J(Q_i, T) \) and \( Q_i+1 \) as in Algorithm 3.

\( \lambda_{min} = \text{Minimum Eigenvalue}(J(Q_i, T)) \)

\[
\text{if } \lambda_{min} < Tol_{small} \text{ then }
\]

Call Bifurcation Routines

\[
\text{else if } \lambda_{min} < Tol_{large} \text{ then }
\]

Compute \( \Delta T \) using pseudo-arclength continuation

\[
\text{else }
\]

Compute \( \Delta T \) using natural parameter continuation

\[
\text{end if }
\]

\[ T = T + \Delta T \]

\[
\text{end for }
\]
and the LPBEs are beyond the scope of this work and interested readers are directed to [6, 45].

\[ \sum_{k=1}^{m} \sum_{j=1}^{m} a_{ijk} \xi_j \xi_k + 2 \sum_{j=1}^{m} b_{ij} \xi_j \xi_0 + c_i \xi_0^2 = 0 \quad i = 1, 2, \ldots, m, \quad (11) \]

\[ \sum_{k=1}^{m} \sum_{j=1}^{m} a_{ijk} \xi_j \xi_k + \eta d_i = 0 \quad i = 1, 2, \ldots, \quad (12) \]

\[ \eta = \ddot{T}_{\text{sing}} \quad (13) \]

\[ \xi_0^2 + \xi_1^2 + \ldots + \xi_m^2 = 1, \quad (14) \]

\[ a_{ijk} = a_{ikj} = \theta^T_i G_{QQ}^0 \theta_j \theta_k \]

\[ b_{ij} = \theta^T_i (G_{QQ}^0 \theta_0 + G_{QT}^0) \theta_j \]

\[ c_i = \theta^T_i (G_{QQ}^0 \theta_0 \theta_0 + 2G_{QT}^0 \theta_0 + G_{TT}^0) \]

\[ d_i = \theta^T_i G^0_T \quad (15) \]

where \( \theta_0 \) is given by \( G_{QQ}^0 \theta_0 + G_{TT}^0 = 0 \)

and \( \theta_0^T \theta_i = 0, i = 1, \ldots, m. \)

Both the ABEs and the LPBEs produce a system of \( m + 1 \) unknowns (\( \xi_i, 0 \leq i \leq m \) for the ABEs, and \( \eta, \xi_i, 1 \leq i \leq m \) for the LPBEs) in \( m + 1 \) multivariate equations, where \( m = \text{dim}(\mathcal{N}(J(Q_i, T))) \). The degree of each polynomial is 2, so there are \( 2^m \) possible solutions. Of these solutions, only those which are real are of interest, and there can be at most \( 2^{m-1} \) paths at the singular point [45]. Once the solutions to the ABE’s or
LPBE’s have been found, the tangent to potential paths out of the singular point can be calculated using Equation (16) for ABEs and Equation (17) for LPBEs for real values of $\xi_i$.

$$\dot{Q}_{\text{sing}} = \sum_{j=0}^{m} \xi_j \theta_j, \quad \xi_0 = \dot{T}_{\text{sing}}$$  \hspace{1cm} (16)

$$\dot{Q}_{\text{sing}} = \sum_{j=1}^{m} \xi_j \theta_j, \quad \xi_0 = \dot{T}_{\text{sing}} = 0.$$  \hspace{1cm} (17)

3.4 Eigenvalues and Eigenvectors

For an $n \times n$ matrix $A$, an eigenvalue $\lambda$ and an associated eigenvector $x$ are solutions to $Ax = \lambda x$ for nontrivial values of $x, (x \neq 0)$. There are a large number of engineering and physical problems which require the computation of the eigenvalues of a matrix, and the subject has been studied extensively. As a result of this study there are a large number of algorithms for finding the eigenvalues and eigenvectors of a matrix, from general purpose to specialized [68].

For a $n \times n$ matrix $A$, let $k$ be the geometric multiplicity of $\lambda$, an eigenvector of $A$ such that $\lambda = 0$, then $\dim(\mathcal{N}(A)) = k$. By finding the geometric multiplicity of $\lambda_0 = 0$ for $J(Q_i, T)$ it is therefore possible to find $\dim(\mathcal{N}(J(Q_i, T)))$. Since the Jacobian in this case is real and symmetric, the algebraic multiplicity is equal to the geometric multiplicity, so we shall henceforth refer only to the multiplicity for this case.

The primary matrix of concern in $mpipath$ is the Jacobian, $J(Q_i, T)$, which is large, sparse, symmetric and real. While there are a large number of software packages available for finding the eigenvalues of sparse matrices (see [40]), the package most suited
for mpipath is ARPACK/PARPACK [53]. ARPACK is designed to find a few minimal eigenvalues, and the associated eigenvectors for large, sparse, matrices. It employs a reverse communication interface, which relies on the client package for matrix vector multiplication. This is especially important as the Jacobian in mpipath is never stored, but calculated as needed. The parallel version of ARPACK is PARPACK [56], which is implemented using MPI, and like ARPACK is a Fortran 77 library. ARPACK is mature software, frequently named as the de facto standard, serves as the basis for the MATLAB eigs routine, and is frequently used for performance comparisons [5,7,36,40]. PARPACK can be employed in Algorithm 4 to compute the minimum eigenvalue of $J(Q_i, T)$.

ARPACK employs the Arnoldi Method for finding a few eigenvalues of a large sparse matrix. This is an iterative method where an approximate solution is continually refined until a sufficiently accurate solution is found. Frequently, the starting value of the solution is a random vector, or set of vectors. ARPACK implements the implicitly restarted Arnoldi Method, which allows the computation to maintain interesting portions of the answer while discarding those portions which do not contribute to the desired final solution. In this case, an approximate eigenvector associated with the desired approximate eigenvalue is stored, the undesirable values are replaced with new random vectors and the computation is restarted. Implicitly restarting allows the computation to minimize storage requirements and increase the speed of computation of desired elements. When the matrix is symmetric, the Arnoldi method becomes a Lanczos method. For full details on these methods see [11,53].
ARPACK maintains a single eigenvector for each eigenvalue computed. This is sufficient if the geometric multiplicity of the eigenvalues under consideration is one, or there is no interest in the multiplicity of an eigenvalue. When the geometric multiplicity of an eigenvalue is of interest, such as when computing $\dim(\mathcal{N}(J(Q_i,T)))$, additional computation is required. Two possible approaches to finding all of the eigenvectors associated with a given eigenvalue involve matrix deflation and employing a block variation of the Arnoldi/Lanczos method. Matrix deflation is the process of reducing the size of a matrix after an eigenvector has been computed. This is the preferred method for ARPACK, and either requires the formation of a new matrix to complete the process, or significantly increases the number of vector matrix multiplications.

When matrix deflation is too complex, or as in the case of mpipath unavailable, then an alternative approach, such as a block method must be considered. Block methods can maintain several approximate eigenvectors for each eigenvalue. Employing the techniques described above, a block method is capable of directly computing multiple eigenvectors associated with a given eigenvalue. This method is employed in the IRBLIGS package which is implemented as a MATLAB function [7,8].

3.5 PCIRBLEIGS

As this work began, there was no parallel implementation of the block Lanczos algorithm available, so an effort was undertaken to parallelize IRBLIGS. The resulting libraries, CIRBLEIGS, a C++ implementation, and PCIRBLEIGS, parallel C++ implementation employing MPI for parallel communications, are the results of these efforts. PCIRBLEIGS retains nearly the same interface as and functionality of IRBLEIGS. This
includes the ability to provide a function to perform matrix-matrix multiplication. PCIRBLEIGS does not support the ability to find eigenvalues about an arbitrary value, but can only locate maximum or minimum eigenvalues. While this functionality would not be difficult to implement, it was not needed for the current project.

3.5.1 Porting PCIRBLEIGS

PCIRBLEIGS was ported taking advantage of GNU Octave, an open source high level mathematically oriented language with a syntax which is compatible with MATLAB [26]. Octave provides a mechanism for directly calling routines written in other languages including C/C++. Porting the software began by replacing leaf routines in IRBLEIGS with equivalent code written in C++. This allowed an orderly approach to porting and allowed parallel testing of results in the existing code as well as in the newly created C++ code. Using this scheme, built in MATLAB functions for comparing and manipulating matrices were employed to enhance debugging and code verification. The sequential code was developed first, in order to increase knowledge of the code and to provide an opportunity to discover opportunities for parallelism, followed by the development of the parallel code.

The code for P/CIRBLEIGS employs both objects (for matrices) and elements of the C++ standard template library. The objects were developed with a sufficient interface for the problem, and included data encapsulation, but do not enforce data hiding. The decision to allow direct access to data was made to provide the ability to reduce function call overhead for actions such as matrix indexing. Client software must provide a matrix-matrix multiplication routine which takes and returns a parallel matrix object, as well as a
driver routine which will employ a number of matrix type objects. Access to this package for other languages can be accomplished by providing a C++ driver routine and a matrix multiply routine. The driver routine creates all needed objects, calls PCIRBLEIGS, places return values in data structures appropriate to the calling language and destroys the objects. The matrix multiply routine must handle matrix multiply, either directly, or by calling an appropriate routine in the calling language. See Figure 14.

Figure 14: Calling PCIRBLEIGS from a Language other than C++

3.5.2 PCIRBLEIGS Parallelization

The primary parallel data structure for PCIRBLEIGS is a “skinny” matrix. This matrix is $n \times j \cdot k$, where $n$ is the size of the matrix for which the eigenvalues are sought and $k$ is the blocksize for the block Lanczos method and $j$ is the number of blocks. It is assumed that $n$ is much larger than $j \cdot k$, and that each process will hold at least $j \cdot k$ rows of the matrix. The last assumption is not necessary, but was made to simplify coding. Finally, for similar reasons, it is assumed that for process $p_i, i = 0, 1, 2, ..., m$, when $i \neq 0, p_{i-1}$ holds the rows immediately above the data stored in $p_i$, and when $i < m, p_{i+1}$ holds the rows immediately below.
The primary purpose for implementing PCIRBLEIGS is to provide a parallel eigenvalue finder for use with mpipath capable of computing the multiplicity of the minimum eigenvalue. To this end, most work associated with this package has been involved with porting, debugging and accuracy testing. Limited performance tuning has been attempted and further work in this area should be pursued.

3.5.3 Numerical Experiments

The following experiments were performed on the “gandalf” cluster at Kent State University, hereafter referred to as gandalf. This cluster consists of 23 nodes each with either two dual core (2220) or quad core (2350) AMD Opteron processors (4 or 8 cores per machine). Each node has 16GB of main memory. At the time of testing, 16 of these nodes were quiescent and were used for test runs. Processes were assigned to nodes in a round robin manner, thus for tests requiring more than 16 processors, multiple processes were run on nodes as required. In no case were more processes than cores assigned to a node. A 1GB/sec Ethernet network was employed for communications and the system was mostly quiescent.

Tests were performed to compare PARPACK, patched to the latest level, with PCIRBLEIGS. The programs were compiled with MPICC or MPIF77 both employing the Gnu GCC 3.4.6 release, and full optimization flags were set. The OpenMPI implementation of MPI was used for communications. For each test, default parameters were employed with two exceptions. Tolerances were adjusted to achieve comparable accuracy, and block sizes, unless otherwise stated, were set to permit finding the correct multiplicity of the minimum eigenvalue. The MPI_Wtime() function was employed for timing calls to
initialize parameters and to computing eigenvalues.

Random vectors are employed in both PARPACK and PCIRBLEIGS which affect both convergence and processing time, therefore direct comparison of time is difficult. Initial vectors were not coordinated between packages nor between runs.

Example 1

The first experiment is part of the PARPACK test library [54].

Suppose we want to solve $A \cdot x = \lambda \cdot x$ in regular mode, where $A$ is derived from the central difference discretization of the 2-dimensional Laplacian on the unit square with zero Dirichlet boundary condition.

It is known that the largest multiplicity of any eigenvalue for this problem is 2 and that

$$\lambda_1 < \lambda_2 = \lambda_3 < \lambda_4 < \lambda_5 = \lambda_6 < \ldots,$$

(18)

For a given value of $m$, let $n = m^2$. Let $A$ be a $n \times n$ matrix defined as:

$$A = \begin{bmatrix}
T & -I & 0 & 0 & \cdots & 0 & 0 & 0 \\
-I & T & -I & 0 & \cdots & 0 & 0 & 0 \\
0 & -I & T & -I & \cdots & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & 0 & -I & T & -I \\
0 & 0 & 0 & \cdots & 0 & 0 & -I & T
\end{bmatrix}.$$
where $I$ is the identity matrix and

$$T = \begin{bmatrix}
4 & -1 & 0 & 0 & \cdots & 0 & 0 & 0 \\
-1 & 4 & -1 & 0 & \cdots & 0 & 0 & 0 \\
0 & -1 & 4 & -1 & \cdots & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & \cdots & 0 & -1 & 4 & -1 \\
0 & 0 & 0 & \cdots & 0 & 0 & -1 & 4
\end{bmatrix}. $$

The goals of this test were to provide performance comparisons with PARPACK as well as scalability and accuracy testing. For this experiment the value of $m$, was varied between 40 and 100 by ten, yielding matrix sizes between $1,600 \times 1,600$ and $10,000 \times 10,000$. For each matrix size, the programs were run with between 1 and 16 processes. In this experiment the first 10 minimal eigenvalues were sought, with a tolerance of $1.0 \times 10^{-8}$. Every run was programatically compared to a base run to ensure accuracy. Typical output can be found in Figure 15. The relationship between minimal eigenvalues stated in equation (18) can be seen in this output. In general, both PARPACK and PCIRBLEIGS were completely successful in finding the requested eigenvalues.

To establish a base for comparison, both programs were run with $m = 100$, or a matrix size of $10,000 \times 10,000$ using 15 processes. Each program was run 30 times with process time and iteration count recorded for each run. The averages and standard deviations are summarized in Table 2. In this case, as in most other cases, PARPACK required less time and PCIRBLEIGS required fewer iterations.
Figure 15: Typical output for PCIRBLEIGS

Total Time 0.608058

Ritz values and direct residuals

<table>
<thead>
<tr>
<th>Col 1</th>
<th>Col 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 0: 5.3036404606809e-03</td>
<td>7.6705606276505e-08</td>
</tr>
<tr>
<td>Row 1: 1.3252069001162e-02</td>
<td>5.5441165488762e-08</td>
</tr>
<tr>
<td>Row 2: 1.3252069001166e-02</td>
<td>6.7990796222141e-08</td>
</tr>
<tr>
<td>Row 3: 2.1200497541649e-02</td>
<td>1.7746938229511e-08</td>
</tr>
<tr>
<td>Row 4: 2.6476028048185e-02</td>
<td>4.6273158516343e-08</td>
</tr>
<tr>
<td>Row 5: 2.6476028048186e-02</td>
<td>2.1522551071947e-08</td>
</tr>
<tr>
<td>Row 6: 3.4424456588669e-02</td>
<td>3.8756529861298e-08</td>
</tr>
<tr>
<td>Row 7: 3.4424456588676e-02</td>
<td>4.1547143071044e-08</td>
</tr>
<tr>
<td>Row 8: 4.4940450039626e-02</td>
<td>3.2114951126889e-08</td>
</tr>
<tr>
<td>Row 9: 4.4940450039642e-02</td>
<td>6.4064645379442e-08</td>
</tr>
</tbody>
</table>

PCIRBLEIGS

Size of the matrix is 3600
The number of processors is 16
The number of Ritz values requested is 10
The number of Arnoldi vectors generated is 0
What portion of the spectrum: SE
The number of converged Ritz values is 10
The number of Implicit Arnoldi update iterations taken is 126
The number of OP*x is 875
The convergence criterion is 1.0000000000000e-08
Table 2: Time and Iteration Comparison for PCIRBLEIGS and PARPACK

<table>
<thead>
<tr>
<th>Program</th>
<th>Measure</th>
<th>Time</th>
<th>Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCIRBLEIGS</td>
<td>Average</td>
<td>3.26</td>
<td>131</td>
</tr>
<tr>
<td></td>
<td>Standard Deviation</td>
<td>0.62</td>
<td>24.15</td>
</tr>
<tr>
<td>PARPACK</td>
<td>Average</td>
<td>2.6</td>
<td>703</td>
</tr>
<tr>
<td></td>
<td>Standard Deviation</td>
<td>0.1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figures 16 through 19 represent the results of running each program once for each matrix size. Both packages showed improvement as the number of processes was increased, up to a limit determined by the matrix size. Furthermore, both packages generally exhibited an increased number of iterations per second as the number of processes increased for matrix sizes beyond a given matrix size. This trend was more apparent in PCIRBLEIGS. Finally both packages exhibited better performance when the number of processes was a power of two. There is no apparent reason for this in the algorithms or code and it is suspected that this may be related to \texttt{MPI.Gather()} operations. This behavior bears further investigation.

Finally, this experiment was repeated for PCIRBLEIGS with a size of $m = 150$ or a $22,500 \times 22,500$ matrix. For this experiment, the values on the diagonal of $T$ were changed to be 100 and the all other non-zero values were set to $-25$. This was done to increase the size of the minimal eigenvalue, which, in this problem is related to the size of $T$. The number of processes ranged from 1 to 32. Figures 20 and 21 present the results of time and iterations per second for this experiment. Once again it can be observed that
Figure 16: Time PARPACK to find the Minimum Eigenvalues for Different Sized Matrices

PCIRBLEIGS performance scales with the number of processes up to a limit for a given matrix size.

Example 2

Example 2 is modified from Castillo [19]. It is designed to test an eigenvalue finder’s ability to locate multiple minimum eigenvalues. For a given size $m$, let $n = m^2$. Create matrix $A$ of size $n \times n$ such that:
Figure 17: Time PCIRBLEIGS to find the Minimum Eigenvalues for Different Sized Matrices

\[ A = \begin{bmatrix}
E & Z & Z & Z & \cdots & Z \\
Z & T & Z & Z & \cdots & Z \\
Z & Z & F & Z & \cdots & Z \\
Z & \cdots & \cdots & \cdots & \cdots & Z \\
Z & \cdots & Z & Z & F & Z \\
Z & \cdots & Z & Z & Z & F
\end{bmatrix}, \]

where \( E = \text{diag}(\lambda, \lambda, \lambda, \lambda) \), \( Z \) is a zero matrix of proper size, \( T \) is a truncated form of \( F \).
Figure 18: Iterations per Second for PARPACK to find the Minimum Eigenvalues for Different Sized Matrices
Figure 19: Iterations per Second for PCIRBLEIGS to find the Minimum Eigenvalues for Different Sized Matrices
Figure 20: Time for PCIRBLEIGS to find the Minimum Eigenvalues for a $22,500 \times 22,500$ Matrix.

Figure 21: Iterations per Second for PCIRBLEIGS to find the Minimum Eigenvalues for a $22,500 \times 22,500$ Matrix.
(it is of dimension $m - 4 \times m - 4$), and

$$
F = \begin{vmatrix}
  a & b & 0 & \cdots & 0 \\
  b & a & b & \cdots & 0 \\
  0 & b & a & \cdots & 0 \\
  \vdots & \vdots & \vdots & \ddots & \vdots \\
  0 & 0 & 0 & \cdots & a \\
\end{vmatrix},
$$

$a \in \mathbb{R}^+, b = -a/2$. By selecting a sufficiently small $\lambda$, it will be a minimal eigenvalue of multiplicity 4.

The goal of this test is to determine each package’s capability for finding eigenvalues with multiplicity greater than one. Tests were run for multiple matrix sizes with multiple numbers of processes. In each test, PCIRBLEIGS was able to locate all four minimum eigenvalues ($2.222222222 \times 10^{-3}$) for every case run, while PARPACK was not able to do so for any case. Typical results are displayed in Figure 22 and Figure 23.

Example 3

Example three is a practical example involving *mpipath* and is an extension of the tests from chapter 1. In this example a $32 \times 32 \times 32$ liquid crystal slab is simulated. This decomposition was selected to ensure as even a distribution of data as possible for each test run. The initial boundary conditions are radial, that is all elements point towards the center of the volume. In this example, 5 eigenvalues were computed at each point along the path with a requested tolerance of $10^{-6}$. The matrix was dumped to file at each stage and selected results were checked using MATLAB.

The matrix in this example has minimum eigenvalues $\lambda_1 < \lambda_2 < \lambda_3 = \lambda_4 < \lambda_5$. In
Figure 22: Experiment 2 Results for PCIRBLEIGS (typical)

Total Time 28.9399

Ritz values and direct residuals

<table>
<thead>
<tr>
<th>Row</th>
<th>Col 1</th>
<th>Col 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2.2222222220170e-03</td>
<td>1.2299610001640e-06</td>
</tr>
<tr>
<td>1</td>
<td>2.2222222220392e-03</td>
<td>1.7822811202790e-06</td>
</tr>
<tr>
<td>2</td>
<td>2.2222222228378e-03</td>
<td>6.5219348782740e-07</td>
</tr>
<tr>
<td>3</td>
<td>2.222222223714e-03</td>
<td>4.1053807574413e-06</td>
</tr>
<tr>
<td>4</td>
<td>1.221430593725e-02</td>
<td>1.7914581986937e-06</td>
</tr>
<tr>
<td>5</td>
<td>1.2214305934734e-02</td>
<td>1.8571248276682e-06</td>
</tr>
</tbody>
</table>

Size of the matrix is 40000
The number of processors is 16
The number of Ritz values requested is 6
What portion of the spectrum: SE
The number of converged Ritz values is 12
The number of Implicit Arnoldi update iterations taken is 757
The number of OP*x is 3024
The convergence criterion is 1.0000000000000e-08

Figure 23: Experiment 2 Results for PARPACK (typical)

Total Time 3.98882317543030

Ritz values and direct residuals

<table>
<thead>
<tr>
<th>Row</th>
<th>Col 1</th>
<th>Col 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.3714654212898E-03</td>
<td>5.1059015122239E-03</td>
</tr>
<tr>
<td>2</td>
<td>2.2222223657714E-03</td>
<td>4.0118816901457E-08</td>
</tr>
<tr>
<td>3</td>
<td>2.2821821215725E-03</td>
<td>9.0926372578007E-04</td>
</tr>
<tr>
<td>4</td>
<td>1.2163123669209E-02</td>
<td>2.0645995822048E-03</td>
</tr>
<tr>
<td>5</td>
<td>1.219560207917E-02</td>
<td>4.4450639516101E-04</td>
</tr>
<tr>
<td>6</td>
<td>1.2214304776920E-02</td>
<td>2.88289416556E-06</td>
</tr>
</tbody>
</table>

Size of the matrix is 40000
The number of processors is 16
The number of Ritz values requested is 6
What portion of the spectrum: SM
The number of converged Ritz values is 6
The number of Implicit Arnoldi update iterations taken is 490
The number of OP*x is 5300
The convergence criterion is 9.999999939225290E-009
all cases, PCIRBLEIGS was able to detect the multiplicity of \( \lambda_3 \), while PARPACK was able to do so in only 4 of the 14 cases or less than 30% of the time. These results came at the price of nearly 300% more time spent in computation of the eigenvalues. Figure 24 displays typical results.

Figure 24: Experiment 3 Example Results

### Iteration 3:

**PARPACK Results**

<table>
<thead>
<tr>
<th>Lambda</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.736349e+00</td>
<td>1.893098e-12</td>
</tr>
<tr>
<td>1.736702e+00</td>
<td>1.756700e-12</td>
</tr>
<tr>
<td>1.780232e+00</td>
<td>1.112197e-11</td>
</tr>
<tr>
<td>1.780773e+00</td>
<td>2.369838e-11</td>
</tr>
<tr>
<td>1.849489e+00</td>
<td>1.517609e-07</td>
</tr>
</tbody>
</table>

**PCIRBLEIGS Results:**

<table>
<thead>
<tr>
<th>Lambda</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.736349e+00</td>
<td>7.828504e-07</td>
</tr>
<tr>
<td>1.736702e+00</td>
<td>6.741016e-07</td>
</tr>
<tr>
<td>1.780232e+00</td>
<td>1.385046e-06</td>
</tr>
<tr>
<td>1.780773e+00</td>
<td>1.570632e-06</td>
</tr>
<tr>
<td>1.849489e+00</td>
<td>8.453363e-07</td>
</tr>
</tbody>
</table>

### 3.6 Conclusions

When *mpipath* runs, it is likely to encounter singular points. The tools in this chapter provide the means to detect, classify and compute paths out of these points. PARPACK is sufficient for detecting bifurcation points, while PCIRBLEIGS provides the means for computing the degree of the nullspace of the Jacobian once such a point is located, and the ABEs and LPBEs provide the tangents for the paths out. While portions of this process can be automated, working near such points is relatively novel and currently requires human intervention. While this intervention can be performed in an environment where
checkpoints and restarts are employed, a better solution involves giving the user direct access to the running code. Methods for providing such interaction will be explored in the next chapter.
CHAPTER 4

Steering and Visualization Software

Previous chapters provided numerous examples of how human interaction is either desired or required during the course of execution of a scientific simulation such as mpi-path. Such interfaces can be constructed along with the simulation, when new software is developed or added to legacy software as the need arises. When adding an interface to a simulation, care needs to be taken to create and maintain software which can be, if desired, executed in the batch environment supported by most supercomputer centers. Steering and visualization (STV) middle-ware provides a viable option for constructing an interface while maintaining compatibility with batch processing.

4.1 Batch Computing

Batch computing, among the earliest and most primitive forms of operating environments, is still employed at supercomputing centers. In this environment, a job is submitted to a queue and when resources become available the job is executed and results are returned to the user for post process analysis. The user is not permitted to interact with the running computation, so tasks such as viewing progress or changing parameter values are not possible. The user has been “removed from the loop”. Placing the computational scientist “back into the loop” by allowing them to alter parameters in a running simulation is referred to as computational steering [38].

Supercomputer centers frequently employ the batch environment to improve efficiency
and maximize the utilization of scarce resources. It is common to limit interactive users to extremely short run times with restrictions placed on memory and processor usage. The pressure to use the batch environment leads to the development of programs which lack facilities for user interaction. STV packages provide an option to build a user interface which can be “disabled” in the batch environment, and enabled for environments where computational steering is permitted. A program designed for such an environment can be constructed with reasonable default parameters which are used during a non-interactive run. When a front end, or viewer is attached, these parameters can be altered in real time by the user to better achieve the desired results.

4.2 Overview of STV Software

Computational steering provides a number of benefits. These include the ability to monitor the progress of a computation so that the computational scientist can detect and correct problems as the simulation progresses, stop a computation which has become stuck or produced erroneous results, or even roll back to a previously saved checkpoint and restart the computation [17]. By interacting with the running simulation, the scientist gains insight into the working of the model and possibly into the phenomena being modeled. Such insight allows the user to fine tune parameters leading to less time wasted on simulation runs which have been incorrectly configured [18]. These realizations, along with the availability of low cost commodity computing clusters led to the development of STV middle-ware systems beginning in the mid to late 1990s.
These early systems included CUMULVS [31, 50], SCIRun [65], CSE [73], FALCON [35] and MAGELLAN [74]. Of this original set, only SCIRun and CUMULVS appear to still exist. SCIRun has grown to become a problem solving environment, where simulation code developed within the SCIRun framework can call upon this framework to deliver a wide array of tools including communications and visualization [75]. Simulations developed within this framework must always be executed within this environment. Portions of CUMULVS have gone on to become the basis for Global Arrays [24] and the entire CUMULVS library has been incorporated into the Common Component Architecture (CCA), a set of tools for parallel scientific programming [3]. The original CUMULVS source code remains available, but is no longer being actively developed. MAGELLAN and FALCON were part of the research efforts at Georgia Technical Institution. This work has culminated in the M-WARE project, which focuses on information flow in distributed applications [30, 51]. Active projects include VISIT [27] and Reality Grid [17, 60]. Both of these products are aimed at grid based applications, and include grid related services and functionality. This list is not comprehensive, but serves to illustrate the background and continuing efforts in this field.

For the remainder of this chapter, discussion will be focused on STV software which can be used to provide a steering and visualization interface to existing simulations such as mpipath. For this reason, problem solving environments such as SCIRun are not considered.
4.3 Tasks Performed by STV Software

The primary task of a STV library is to provide for the flow of information between a user interface, or viewer, and simulation software. For each state of a simulation, the STV software must extract and synchronize the desired data from the simulation, package this data into a network neutral format, and deliver the data to connected viewers. Viewers produce steering information, which the STV middle-ware delivers to the simulation. This process is illustrated in Figure 25.

![Figure 25: Interaction between a Simulation and a Viewer in an STV Environment](image)

For the purposes of this document, two distinct types of data are recognized. Field data refers to distributed data, while parameter data refers to data which is identical across all processes. Field data is most frequently associated with large data structures, such as the discretized liquid crystal data field in mpipath, but can be as small as a one dimensional array, where each process holds an unique value. The major challenge, when dealing with fields, is to ensure that field data transmitted is composed of data
from the same state or iteration. To ensure this consistency, STV packages rely on some form of loose synchronization. Parameter data, on the other hand, must be delivered to all processes in the computation during the same state, which again requires some form of synchronization. In general, field data is large and requires considerable resources, memory and network bandwidth for transmission. As such, this data is normally only transmitted when requested. Parameter data, on the other hand, is normally small and is transmitted to all attached viewers during every update.

The ability to extract and synchronize distributed data, which many STV packages provide, can be easily extended to include other services such as checkpointing and model coupling. A checkpoint occurs when the state of the entire computation is saved, providing data which can be used to restart the computation later. To perform a checkpoint save, the STV software needs to extract relevant field and parameter data, which is transported to a client that saves the data. A restart is accomplished by a checkpoint client loading a saved checkpoint and returning to a computation via the STV middle-ware. In this process, the checkpoint client can be viewed as a viewer, which may or may not allow user interaction.

Model coupling allows two simulations to share run time data in what is sometimes called an \( m \times n \) exchange where data divided between \( m \) processes is transported to and divided among \( n \) processes. In model coupling, each participating simulation can be seen as a viewer. Consider two simulations, a large scale vegetation model, which uses climate data to determine where vegetation grows, and a weather model, which includes surface cover to determine heat dissipation. By coupling these models, the weather simulation
can receive simulated land cover data from the vegetation model, and in turn provide rainfall and temperature data needed to predict plant growth. In this case the STV software provides the data extraction and transportation services required, see Figure 26.

![Figure 26: Model Coupling, Checkpointing and Steering in an STV Environment](image)

**4.4 Desirable STV Characteristics**

In order to better evaluate STV packages the following set of criteria has been established by the author. These criteria are based upon experiences employing STV software, working with computational scientists, and maintaining cluster computing environments.

**Minimally invasive.** It should require very little code, and perhaps no restructuring to instrument most simulations for use with an STV package. If a major code overhaul is required to employ a package it will not be used [18].

**Minimal impact when not in use.** When no viewers are attached to the package it should not impact the runtime of the computation.
Low impact when in use. When the middle-ware package is in use with connected viewers, the impact on the system should be minimized. Unless fields are requested, such data should not be transmitted. The middle-ware package should only block when absolutely necessary.

Full set of features. The software must support the tasks the user wishes to accomplish. These tasks should be addressed directly in the API.

Easy to use. Use of the software should be straight forward with full documentation and clearly written examples.

Minimal support software requirements. The software should require few, if any, additional packages for installation and operation.

It is recognized that no STV package may meet all of these criteria.

4.5 CUMULVS

Based partially on the above criteria, the CUMULVS package was selected for use with this project. Additional considerations included software maturity and availability. Several other system, such as ECHO and VISIT were available, but were still in development stages, while newer packages such as the Reality Grid did not yet exist at the onset of this project.

CUMULVS meets the previously established criteria for STV packages in most of the categories. This package is minimally invasive, and has been found to have minimal impact on instrumented packages [14]. The task of instrumenting code is reasonably simple, depending on the data decomposition employed, and is fully documented. Viewer
creation is less well documented, more difficult, and will be addressed in later chapters. The feature set is full, but could be expanded as noted below. CUMULVS relies on a single support package, the Parallel Virtual Machine or PVM, which is available for most systems.

CUMULVS, or Collaborative, User Migration, User Library for Visualization and Steering, is a middleware package developed at Oak Ridge National Laboratory. Built on top of the PVM parallel communications and programming library, CUMULVS has bindings for FORTRAN and C. It supports checkpointing, fault tolerance, and model coupling. The software comes with a set of example applications and a set of field viewers [50].

In order to extract field data from a simulation, the user must describe how that data is distributed through a data decomposition description. CUMULVS is designed to support a number of pre-defined regular data decompositions, user defined decompositions, and a “particle” decomposition for unstructured data. It supports all standard (built in) C and FORTRAN data types. CUMULVS assumes that the application provides process synchronization and will block only when necessary, or when requested to do so by a connected viewer. CUMULVS consists of two libraries: an application-side library, which becomes part of the simulation and a viewer library which is the basis for constructing viewers. A good overview of CUMULVS’ capabilities is available in [50], while full details are available in [49].

An application instrumented with the application-side library can support dynamic connections from multiple viewers. This means that at any time during the computation,
a viewer can attach and begin receiving data, or stop monitoring the simulation and detach. At each iteration, or update, the application-side library transmits all scalar parameters and any requested data fields to connected clients. The library supports the extraction and synchronization of field data, which may be filtered in a number of ways. A discussion of deploying this library will occur in the next section.

The viewer library provides a set of routines for connecting and interacting with an application instrumented with the application-side library. This library provides routines to specify which fields, and what parts of the selected field, the viewer is interested in, to receive that field data, and to send and receive parameter values. Further routines provide flow control and allow specification of frequency of frame transmission. CUMULVS viewers will be discussed in the next chapters.

4.5.1 Instrumenting a Simulation with CUMULVS

The first step to employing CUMULVS to steer a simulation, or visualize the associated data, is to instrument the application code. This process consists of three steps: initializing the CUMULVS library, defining and describing data needed for steering and visualization, and performing data updates. This section contains code examples from the CUMULVS Tutorial [12] and represents the instrumentation of a parallel implementation of Conway’s Game of Life [34]. Code segments presented in this section provide the basis for later discussion, and are not meant to be a comprehensive introduction to instrumenting code with CUMULVS. Interested readers should see [12, 49].

Initialization is illustrated in Listing 1. The application is identified by the first parameter, which must be known and is used by all viewers wishing to connect to the
application. The second parameter is a message tag, which must be unique to the application. The third and fourth parameters provide information about the size of the computation (number of processes involved) and the rank of the process making the call. This call must be invoked once in each process.

Identifying the fields and scalar parameters to be communicated via CUMULVS is the next step. The procedure differs for fields, for which a data decomposition must be defined, and scalar parameters, which simply must be registered. When defining a field data decomposition, parameters specifying type, dimensions, and distribution of the data must be supplied. The data decomposition for the LIFE program requires approximately 10 lines and can be found in [12]. Providing a data decomposition description is technically detailed and beyond the scope of this document.

Registering fields and scalar parameters differs slightly. Parameters are registered by providing the address, name, type and use of the parameter. Fields registration must also include a decomposition, and information about how that decomposition is stored on the local process. Routines to provide a mapping between a scalar parameter or field name, and the associated data address are illustrated in Listing 2. In this listing, \texttt{maxAge} is an integer containing the number of iterations the oldest cell in the computation has been alive, and is registered using the name \texttt{Max Age}. \texttt{board1} is an array of integers, which holds a copy of the game board and it is referred to by the name \texttt{First Game}.
paramIDs[1] = stv_paramDefine("Max Age", &maxAge, stvInt, stvVisOnly);
fieldId[0] = stv_fieldDefine(board1, "First Game Board", decompId, offsets, decl, stvInt, paddr, stvVisOnly);

Listing 2: Registering a Parameter and a Field in CUMULVS

**Board.** The name becomes the primary mechanism for accessing the associated data for attached viewers. The final parameter stvVisOnly, a constant, will be addressed in the last section of this chapter. These routines register the address of the data structure with the application-side library for manipulation, and all addresses must remain valid throughout the period of steering. A major disadvantage to this scheme is that it tends to encourage programmers to introduce global variables to maintain accessibility throughout the period of steering and visualization.

The final step to employing CUMULVS is to notify the application-side library that new data is available and should be transferred to any connected viewers. This is performed by a call to stv_sendReadyData(), is illustrated in Listing 3. All processes participating in the computation must call this function, and when necessary, CUMULVS will block faster processes at this point. This routine is called whenever the computation is ready for a data transfer and results in a two way data communication: active fields and parameters are sent to attached viewers and data from viewers is returned to the simulation. New parameter values are stored directly in the associated variables within the simulation. The return value of this function indicates the number of parameters which have been changed and, with additional processing, the simulation can determine which values have been modified.
4.5.2 Parallel Simulations Instrumented with CUMULVS

In the course of this project a number of parallel applications have been instrumented. These applications, selected to test CUMULVS’ ability to deal with different data decompositions, include two implementations of Conway’s Game of Life, a molecular dynamics simulation ALCMD, and mpipath. Custom viewers, which will be discussed in the next chapter, were written for these applications.

Conway’s Game of Life involves a two dimensional array of integers in a nearest neighbor computation. The usual implementation involves two arrays, a current board and the next board, where data on the current board is used to compute the value of data on the next board. At the end of an iteration the role of each board is exchanged. This exchange is normally accomplished by swapping pointers or changing an array index. The board has periodic boundary conditions and generally a one cell thick border layer is exchanged between neighboring processes at the end of each round of computation. The two different versions of this simulation instrumented were MLIFE2D [34] and PARLIFE [63].

MLIFE2D is C code employing MPI’s MPI_Dims_create() routine which provides a division of processors in a Cartesian grid. This routine attempts to create a balanced distribution of processes in each coordinate direction. The distribution will change based on the number of processes and the dimensions of the data grid [29]. A custom CUMULVS data decomposition is required as MPI_Dims_create() may produce a decomposition
which does not match CUMULVS’ `stvBlock` definition. The data required to define the
decomposition is easily derived from the data returned by `MPI_Dims_create()`.

PARLIFE is a demonstration program for the `GhostPoint` library [62] written in
C++. The `GhostPoint` library is designed to support nearest neighbor computations
on a distributed multidimensional matrix and automatically performs the memory allo-
cations and data exchanges required for such computations. Instrumenting distributed
multidimensional arrays in this package is trivial as all of the data required to define a
CUMULVS data decomposition is stored in the `GhostPoint` data structure, and adding
utility routines to automatically instrument `GhostPoint` programs would be equally triv-
ial.

The molecular dynamics simulation instrumented is the Ames Lab Classical Molecular
Dynamics (ALCMD) simulation [72]. This is a particle simulation written in FORTRAN
77 with MPI as the underlying parallel library. Data is unstructured and decomposed
spatially, with each process containing all particles for a given region of simulated space.
Data of this type is registered as a `stv_particleField` due to its irregular structure.
Such a field is somewhat difficult to define and requires more coding effort to instrument.
The supporting documentation for this type of decomposition contains example code,
helping to make the task more manageable.

Data in `mpipath` is decomposed as described in Chapter 2 and is stored in a four
dimensional array composed of three spatial dimensions plus an additional dimension to
store the elements of the tensor. This data is defined with three dimensions collapsed,
or not distributed, while the fourth dimension requires a custom description. Boundary
data is stored independently in six three dimensional arrays, and requires a separate data
decomposition definition for each face type, as the data distribution can differ for each.
Once again, while a custom data decomposition is required, it is fairly straightforward
and reasonably simple to instrument.

4.5.3 Critique of CUMULVS and Suggestions for Improvement

In the course of instrumenting the simulations mentioned in the previous section, a
number of areas where CUMULVS could be improved were noted. These areas are ex-
plored below, with suggestions on how the improvements could be implemented provided
when applicable.

Instrumenting simulations using CUMULVS is well developed and well documented.
Instrumenting parameters is well defined with adequate documentation and example
code. The existing decomposition structures for field data, based on the high precision
FORTRAN (HPF) data compositions, are easy to employ. Defining new decompositions,
while not trivial is a manageable task. The software could be made more usable by
adding decompositions for standard non-HPF decompositions such as those returned
by MPIDims_create(). Further usability could be achieved by providing extensions
to data decomposition and transport libraries such as GhostPoint which automatically
instrument supported field decompositions.

It is common for simulations to employ two structures, containing a current value
and the next value, as was done in both implementations of life. Currently simulations
employing such a data storage scheme must instrument both structures and CUMULVS
will extract and transport the data for both structures each system update, leading to
redundant data transmission. Furthermore some scheme must be established whereby viewers can detect which field is current. By establishing a special decomposition for such data, CUMULVS would become more efficient and more robust. Such a decomposition would be reasonably easy to implement. The user must specify the data decomposition, but in place of a single address, an array of alternate addresses could be specified along with a scalar index. By updating the index, a process most likely to occur in the computation in any case, CUMULVS could extract and transport only the “current” data field. This enhancement would simplify the construction of viewer software, as it would automate the task of tracking the currently active field.

When instrumenting parameters, CUMULVS allows the programmer to distinguish between parameters which are sent to viewers (\texttt{stvVisOnly}), checkpointing processes (\texttt{stvCpOnly}) or both (\texttt{stvVisCp}). These parameters are then treated as a pass by reference, in that the values can be modified by CUMULVS after a call to \texttt{stv\_send\_ReadyData}. When an attached viewer sends a new value for a parameter, CUMULVS copies this value directly into the associated variable in the simulation. The current mechanism to protect a parameter is cumbersome. By adding the ability to declare a parameter for viewing (\texttt{stvReadOnly}), the package could greatly reduce the work required to instrument a simulation while protecting parameter values from unwanted change.

Requiring all variables to be in scope at the time of the \texttt{stv\_send\_ReadyData} call is a problem that tends to lead to an over-abundance of global variables. If a user wishes to monitor a variable declared within a subroutine, they must promote this variable to the scope of the routine where the parameters are declared. The parameter must either be
passed to the subroutine, which, if possible, may require a substantial code rewrite, or make the scope of the variable global. One solution to this problem would be to emulate ECHO, and handle each stream independently. This solution, however, would greatly increase the complexity of steering. There appear to be no good solutions to this problem and it bears further investigation.

A number of changes can be recommended to CUMULVS’ data update structure. All interaction takes place during a call to \texttt{stv\_sendReadyData()}\!. During the execution of this routine all data is exchanged and connections are managed. It is expected that this routine will be called once each iteration of the main loop \([49]\), and in simulations where this assumption is valid, and execution of the main loop is on the order of seconds to minutes, this is a viable model. Problems arise when main loop execution times become longer, or when parameters need to be updated independently of field extraction.

CUMULVS’ support for dynamic viewer connection creates a problem in the first case, long running main loops. Connections are established across multiple calls to \texttt{stv\_sendReadyData()}\!. A long interval between calls to this routine leads to two problems. Users generally expect interfaces to respond quickly and prolonged delay in connecting causes problems. This can be mitigated by displaying warning messages and connection status, but it remains a problem if the delay between updates is more than a minute. A more critical problem relates to program timeout. Default values compiled into PVM packages require connections within seconds. While these values can be changed, and PVM recompiled, some default must be set and the problem will persist. A simple solution is to provide an API call to handle connections. A routine such
as \texttt{stv\_UpdateConnections()} could be called periodically to alleviate this problem, although difficulties could still arise when a simulation has performed a long running library call, which includes code to which the instrumenter has no access and would prevent inclusion of such a call.

A second problem with an all encompassing \texttt{stv\_sendReadyData()} call occurs when there is a desire to steer the computation without updating field data. Such a situation can occur at the bottom of a computation loop when there are a number of actions that the user might select via a parameter steering viewer. The simulation may go into a small loop, receiving and updating parameters until it receives a message to continue with the main computation. With the current API, all data, including field data, is exchanged each iteration of the inner loop. This situation could be corrected by adding a call to the API such as \texttt{stv\_sendParameterData()}. Both routines, \texttt{stv\_UpdateConnections()} and \texttt{stv\_sendParameterData()}, as well as a complementary routine \texttt{stv\_sendFieldData()} should be relatively easy to implement by decomposing \texttt{stv\_sendReadyData()} into these underlying component routines.

There are a number of minor bugs and inconsistencies in CUMULVS, which should be cleaned up in a new release of the product. Passing strings as a parameter is supported but the code to deal with this is broken. There are a number of minor memory leaks, and a configuration problem when the maximum allowable dimension of a data decomposition is changed. There is an inconsistency in a function parameter when building a particle list in FORTRAN, where a pass by value is expected. All of these issues are minor and could be easily fixed.
CUMULVS’ most outstanding weakness is a lack of documentation [58]. Procedures and routines for instrumenting code are supported by a full set of manual pages, examples, and tutorials. The steering API, on the other hand is almost completely void of documentation. Existing routines are listed without any form of formal documentation, and example code is complex and lacks comments. The interface employs a number of data structures, which are undocumented.

CUMULVS’ complement of parameter viewers, along with the viewer API, provides a final area for improvement. In contrast to the reasonably full array of field viewers provided, the package comes with a single, all inclusive text based steering viewer [58]. This viewer is cumbersome and difficult to use, poorly documented and the code is monolithic in nature. This code serves as a poor example for potential viewer creators.

The next chapter discusses field viewers, including a number of viewers designed for this project. The following chapter examines parameter viewers and steering in general. In these chapters, additional solutions to problems presented here will be addressed.
CUMULVS Field Viewers

A field viewer is a STV client specifically designed to deal with field data. Many of the STV systems described in the previous chapter were originally aimed at providing access for such a viewer. Accordingly most systems provide a rich array of field viewers for the visual display of such data. This chapter will examine CUMULVS field viewers, including two produced especially for this project.

CUMULVS evolved from a system called PVMAVS, which was designed to simplify the task of connecting a distributed computation to the visualization environment AVS [48]. As such, CUMULVS has had a viewer for AVS, a commercial visualization product developed by Advanced Visual Systems Inc. from the beginning of the project. Other field viewers range from a simple web viewer, “Gif-O-Matic,” which presents the user with a 2D slice of the selected field accessed via a web browser, to a VTK/CAVE viewer which employs the Visualization Toolkit (VTK) in the CAVE immersive environment. An example viewer, called “Slicer,” built upon TK/TCL is shipped as part of the CUMULVS source code. For more information on these viewers, see [50].

One area not explored by the developers of CUMULVS was the production of a viewer to run on a visualization cluster. Such a viewer could be built upon the Chromium library for distributed graphics. Chromium became fully available only after active development of CUMULVS had ceased, but is a natural fit for visualizing distributed data. To explore
this area, a Chromium aware CUMULVS viewer was developed for this project. Before discussing this viewer, the Chromium system will be described.

5.1 Chromium

Chromium [43] is a distributed system for rendering computer graphics. Based on the industry standard OpenGL graphics library, Chromium provides the ability to build an OpenGL based parallel program to be run on a single processor or on a rendering cluster and display the results on devices ranging from a single monitor to a “wall” type tiled display [39]. To fully understand Chromium, a quick overview of OpenGL is necessary.

OpenGL has been described as a state machine which processes a stream of graphics commands [1]. Using the OpenGL API, a programmer issues commands which either change the state of the machine, such as turning on lights, or send data through the machine, such as drawing a line. OpenGL responds to these commands and produces an image. Chromium takes advantage of this architecture by intercepting the stream and, when necessary, duplicating and transferring it to other computers within the cluster.

Chromium accomplishes this task using Stream Processing Units or SPUs. Chromium includes a number of SPUs with each designed for a specific task. These tasks include operations such as rendering an image (Render SPU), packing a portion of the stream into a buffer for transport across the network (Pack SPU), or sorting data to decide where it should be sent (Tilesort SPU) [42]. There are additional SPUs but these three are sufficient for a simple distributed graphics application.

A Chromium application is built by forming a digraph, where the vertices are SPUs and the edges are connections between SPUs. The connections can represent a transfer
between SPUs on the same machine or a network transfer of data. The most basic distributed graphics system involves an application generating an OpenGL stream directly connected to a Pack SPU. The Pack SPU will transport the data over a network connection to a Render SPU, which will render the stream. A more complicated network is presented in Figure 27. In this figure, an application sends an OpenGL stream to a Tilesort SPU. The Tilesort SPU sorts the data according to where it will eventually be displayed and sends the data, over the network, to an appropriate Render SPU, which in turn will generate an image to display on an attached display device. A final example consists of three applications, each with an attached Tilesort SPU generating parallel OpenGL streams. Each Tilesort SPU is connected to four Render SPUs, which drive a display wall. This example is pictured in Figure 28.

![Figure 27: A Chromium Network to Render on a Four Display Wall](image)

There are two choices when employing Chromium with an OpenGL program. A legacy program can take advantage of Chromium’s application faker (`crappfaker`). The `crappfaker` replaces the default system supplied OpenGL libraries with a Chromium library. The Chromium library, which mimics OpenGL’s API, passes the stream to a Pack SPU. The Pack SPU is the first node in a Chromium digraph and sends the stream through
the system. By using crappfaker, legacy code can, among other things, be displayed on a tiled display wall. The second choice is to build a Chromium enabled application. In this scenario, the code is built using the Chromium API and linked against the Chromium library. An application built in this manner becomes the initial vertices in a Chromium digraph.

5.2 A Chromium Enabled Viewer for CUMULVS

By combining CUMULVS’ ability to perform an $m \times n$ data exchange with Chromium’s ability to perform parallel rendering, a distributed viewer becomes possible. Figure 29 provides an overview of the system architecture for such a viewer. In this figure, the application is connected to three visualization engines. These engines receive a portion of the field data from the application and generate the OpenGL calls required to generate a final image. All processes in this figure, represented by rectangles, can be run on different nodes of a cluster. In this figure, 5 nodes are dedicated to the application, three to visualization, one to user interaction, and four to image rendering.

To demonstrate the feasibility of the design represented in Figure 29, a special purpose particle viewer was constructed [14]. The viewer is an MPI-based program which
Figure 29: A Chromium Enabled Viewer for CUMULVS Rendering on a Four Display Wall

consists of a number of visualization processes and a user interaction process. The process
responsible for user interaction provides coordination among all processes of the viewer.
This process communicates changes in the configuration, such as rotation and display
parameters, to all visualization processes via MPI, and provides flow control information
to the application via CUMULVS. As described above, other processes in this viewer
receive field data from CUMULVS and generate the appropriate OpenGL streams.

Using a common set of graphics routines, two versions of this viewer were implemented
in C++, a sequential viewer based on the architecture in Figure 27, and a parallel viewer
as described in Figure 28. For comparison purposes, both viewers employed the same
OpenGL based drawing routines.

Performance was tested on a 16 node Beowulf cluster composed of 3.2 GHz P4 pro-
cessors, with 1.5 GB of main memory, connected via Gigabit Ethernet. Six nodes were
connected to flat panel displays forming a $3 \times 2$ tiled wall with resolution $3840 \times 2048$.
In trial runs, ALCMD was executed on four nodes with atom counts ranging from $2^8$ to
All processes, simulation, visualization and rendering, were assigned to nodes such that only one process was executed per node. In the most extreme case 13 nodes were employed (four simulation, four visualization, one user interaction and four rendering).

Data was collected using a FPS viewer, which will be described in the next chapter. This viewer measures the frames per second, or iterations per second, achieved by a computation. Since the visualization process consumes more time per iteration than the simulation process, the simulation process will be blocked between frames. The results of the experiment are presented in Figures 30 and 31. In these Figures, the base performance of ALCMD is marked with the label FPS. For all other data, items beginning with a $D$ represent runs of the distributed viewer, while items beginning with an $S$ are from the sequential viewer. The number following the dash represents the number of nodes used for rendering. In Case $D-4$ the distributed viewer was run with four rendering nodes, while in case $S-1$ the sequential viewer was run with one rendering node.

As expected $D-4$ performs better than the other viewers when there is sufficient data to warrant its use. $D-1$ performs better than either sequential case, and the sequential cases are essentially indistinguishable. As the number of atoms increase, the frame rates of $D-4$ approach 50% of the computation, compared to 3% achieved by $S-1$ and $S-4$. These results indicate that using Chromium along with CUMULVS produces a viewer, which in a wall environment produces a significant improvement in frame rendering rates.

5.3 A CUMULVS Viewer for $mpipath$

In order to better understand the processes occurring within the $mpipath$ simulation, a viewer, $xpathview$, was created. This viewer is built upon several projects [9, 57], and
Figure 30: Performance of the Parallel Viewer

Figure 31: Performance of the Parallel Viewer, $2^{14}$ Through $2^{20}$ Atoms
provides interactive visualization for both real time and saved data from \textit{mpipath}. The viewer is designed to be compatible with Chromium for display on either a single monitor or a display wall. The program has a number of special features, including the ability to sort and filter data, the ability to examine individual data points, and the use of low cost input devices for human interaction.

5.3.1 \textit{xpathview} Control

Control of \textit{xpathview} is accomplished through a number of input devices. Traditional input devices such as the mouse and keyboard provide basic levels of interaction. Keyboard commands can be used to accomplish all tasks within the viewer, and the user is presented with a context dependent guide (illustrated in Figure 32) displayed on the terminal where the process was started. There is a drop down Graphical User Interface (GUI) menu available, when interacting with a mouse, which allows the user to control most tasks. Keyboard interaction is cumbersome, and mouse interaction becomes problematic when employing Chromium on a wall type display, therefore \textit{xpathview} has been built to support additional input devices including a 3DConnection Space Mouse and a Nintendo Wii Remote.

The Wii Remote, commonly referred to as a Wiimote, is an input device for the Nintendo Wii game system. Pictured in Figure 33, this device communicates via a Bluetooth connection. The device is equipped with a eight buttons, a touch pad, a motion detector and an infrared camera. An attached nunchuck (also pictured in Figure 33) provides an additional two buttons, motion detector, and joystick. The joystick, motion sensors, and the touch pad provide two dimensional input, much like a traditional mouse,
while the buttons generate keypress type events. The ability to communicate wirelessly allows a Wiimote user to move about and interact with a large wall display.

Figure 33: The Nintendo Wii Remote with Nunchuck

\( \texttt{xpathview} \) incorporates the Wiimote using the \texttt{wiiuse} library [52]. The Wiimote can be used to select most configuration options and to advance frames. The Nunchuck is used to provide mouse like functions, such as pointer movement and button presses, and gives
the user alternative access to the GUI menu system. Individual elements can be selected via a cursor, which can be controlled horizontally with the touchpad and vertically using the motion detector in the Wiimote. Buttons are assigned various roles depending on the context of the display, for example, when the clip box is active, the touch pad is used to control the position of the walls. The IR camera is not employed as experiments indicated that while it is acceptable for small screen display, its use is impractical with a wall display.

In addition to the Wiimote, \textit{xpathview} employs a 3DConnection Space Navigator. This device is a 3Dimensional mouse which supports six degrees of freedom \cite{2}. Pictured in Figure 34, this device allows a user to naturally input three dimensional translations and rotations. Equipped with a joystick and two buttons, the device is programatically accessed through the spacenav \cite{71} library. This device is used in \textit{xpathview} to allow the user to rotate and translate the field data.

![The 3DConnetion Space Navigator](image)

\textbf{Figure 34: The 3DConnetion Space Navigator}

Figure 35 contains a screen shot of \textit{xpathview} displaying data of a $16 \times 16 \times 16$ cube.
Each point of data is displayed as a solid sphere, and colors are based on a test for axially.
A value in the middle of the front edge of the top slice has been selected, as noted by
the small wireframe box, hereafter referred to as the cursor. Information relating to the
selected data value, including a graphical representation is displayed on the Heads Up
Display (HUD) on the right side of the screen. The filter has been set to the color bin
test, and the pop-up menu in the upper right hand side of the screen will allow the user
to select which color spheres to display or hide. No data filters have been applied in this
screen shot and all 4,096 elements are displayed in the image. The Wiimote is currently
set to change the scale of the elements, and the mouse pointer is positioned near the
lower right corner of the screen.

![Image of the Heads Up Display (HUD)](image)

Figure 35: *xpathview*

5.3.2 Filtering Data

Even in the minimum discretization pictured in Figure 35, it is hard to discern details
and spot patterns within the data. Colorization schemes assist in this task, but often
fall short due the the number of data points and the three dimensional nature of the data. To assist users in spotting patterns, a number of features have been implemented in *xpathview*. These features include scaling and shape options, clip boxes, and color filtering schemes.

To illustrate the utility of these features, the visualizations in this section are from a $22 \times 22 \times 22$ cube with a twist boundary condition, the simulation is very near a singular point. All elements are drawn with a cylinder corresponding to the direction of the eigenvector of the largest eigenvalue, and are uniformly scaled. The data values, along with boundary values are displayed for a total of 13,552 data points. The color scheme applied is a Biaxial Test designed to highlight the axiality of a data point. In this test, let $|\lambda_1| \leq |\lambda_2| \leq |\lambda_3|$ be the eigenvalues at the point. Let $h = 320 \times (1 - (|\lambda_2| - |\lambda_1|)/|\lambda_2|)$. Values where $|\lambda_2| \approx |\lambda_1|$ will produce a value of $h$ close to 320, while data points where $|\lambda_1| \ll |\lambda_2|$ will produce values of $h$ close to 0. The value of $h$ becomes the hue for generating a color in the HSV color space.

Hue Saturation Value (HSV) color space is pictured in Figure 36, and provides a convenient method for automatically generating scaled colors. The basic color is determined by hue, which is an angle equal to $0^\circ$ for red and $300^\circ$ for magenta. Saturation determines the purity of the color, with a saturation of 1 being a pure color and a saturation of 0 white. Value determines the lightness of the final color, where 0 is black and 1 is light. Conversion from HSV to Red, Green and Blue (RGB) colorspace is accomplished via a simple algorithm [10].

Using the value of $h$ computed above for Hue, and letting Saturation and Value each
be 1, a color coding for axiality can be achieved. In this scheme, uniaxial areas, or areas
where the two minor axes of the representation are roughly equal are magenta colored,
while biaxial areas, or areas where the two minimum eigenvalues differ greatly are colored
red. Areas where $|\lambda_2| - |\lambda_1| \approx 0$ are colored black.

Figure 36: The Hue Saturation Value (HSV) Color Space

In Figure 37 the entire data set is displayed. Figure 38 displays the initial boundary
conditions for three sides, a view accomplished by only selecting the three associated boundary fields for display. In figure 39, only nearly biaxial elements are displayed where a typical element has eigenvalues of $-0.137, -0.015, 0.151$. In Figure 40 the two extreme cases are displayed, where the typical magenta element has eigenvalues of $-1.78, -1.72, 3.5$. Finally in Figure 41 values where $2 \times \lambda_1 \leq \lambda_2$ are displayed. This final image has been clipped so that only the internal values of half of the cube are visible. It is possible to filter exterior values as well, but such images are more difficult to see in stationary images and therefore, have not been presented.

In *xpathview* the primary filtering mechanism is built around data point colorization.
Figure 37: Visualization of a Liquid Crystal Simulation, Unfiltered

Figure 38: Boundary Values of a Liquid Crystal Simulation
Figure 39: Biaxial Values of a Liquid Crystal Simulation

Figure 40: Biaxial and Uniaxial values of a Liquid Crystal Simulation
There are currently thirteen different colorization schemes. Schemes range from simple error detection (illustrating problems in the tensor value), inter-frame changes (illustrating changes in the magnitude of the tensor or alignment of the director for a data point), to attempts to identify properties such as energy and axiality. Roughly one half of these color schemes are from previous visualization packages. xpathview is built in a modular fashion allowing for the easy addition of new colorization schemes.

The filtering mechanism allows the user to select data points based on a number of color characteristics. The most useful, Color Bin Test, illustrated above, allows the user to filter based on the hue of a data point. Tests to filter on saturation and value, as well as tests based on the Hue, Saturation and Light (HSL) color space are included. Figure 42 illustrates the HSL Light filter when the Axality Order color scheme is applied. In this filter the lightness of the color space is divided into 10 bins, 0% to 9%, 10% to 19%, ..., 90% to 100%, from which the user can select which ranges should be displayed. In the images in Figure 42, the selected bins are displayed in the popup menu. As with
color schemes, the filtering module is designed to be extensible and it is relatively simple to add new filtering routines.

Figure 42: A Liquid Crystal Simulation Colorized by Axality Order

The system is equipped with two clip boxes, the inside clipbox which clips (or does not display) data points within the box, and the outside clip box which clips points outside of the box. Figure 43 displays a clipped portion of a simulation colored by Order, with the axis for each point displayed. Portions of this figure display the data clipped to a single plane in each dimension, the outside clip box, the inside clip box and both clip boxes. Controls to permit the movement of the outside clip box enable the user to step through slices of the data. Figure 44 shows a series of screen shots as the clip plane travels through the liquid crystal volume.
5.3.3 Additional Features and Comments

Users have a number of additional features to select from. A stride value can be set along each axis allowing users to display every $n^{th}$ element. Scaling values can be set to normalize against the largest eigenvalue in the data set or to uniform a size, and the scale factor for elements can be changed for the entire dataset. Frame control allows for single frame advance (forward or backward in off line mode), or continuous advancement, providing a level of animation. The entire data set can be translated and rotated in three dimensions. The system works on a single desktop, as pictured in Figure 45 or via Chromium on a display wall shown in Figure 46.

There has been no effort to produce a Chromium enabled viewer for *mpipath*. In general, for real sized cases, execution time for a single iteration of *mpipath* significantly
Figure 45: xpathview on a Desktop

Figure 46: xpathview on a Wall
exceeds rendering time. While the creation of such a viewer would improve the inter-
active performance of xpathview, it was believed that improved filtering provided better
performance and yielded greater information from the graphical presentation.

The entire xpathview package would benefit from use by a specialist in the area of liquid
crystal physics. The colorization schemes are mostly aimed at exposing the mathematical
properties of the model and underlying simulation. Additional filtering techniques, based
on properties such as element size, could easily be added, while more advanced techniques,
such as isosurface construction or streamlines indicating element alignment would add to
the interpretation of the data.

5.4 A Steering Toolkit for CUMULVS

While there is a wide variety of field viewers available for CUMULVS, the need for
special purpose viewers will always exist. Packages such as AVS and VTK assist in the
creation of viewers, but the desire, or even need, to build custom viewers at a lower
level requires programmers to employ CUMULVS’ steering library. The steering library
is a very low level API, which requires multiple calls to perform simple tasks, such as
connecting to a simulation.

To connect a viewer to an instrumented simulation, the viewer must create a unique
I.D., initialize the viewer and initiate steering. This task involves four API calls and
some memory management and error checking. Two viewer specific parameters, the
application name and update frequency are needed and a pointer must be maintained.
CUMULVS’ ability to have a viewer connect to multiple simulations creates much of
the work associated with this task. Most viewers will be created to connect to a single
simulation, so default parameters will suffice nearly every time a viewer connects. For most viewers, a single call, such as \texttt{InitSTV(appname, frequency)} will simplify the creation of viewers.

A toolkit is a set of routines which accompanies a library and provide a high level interface to the API. OpenGL has several associated toolkits, including GLU, the OpenGL Utility Library. GLU provides, among other things, utility functions which group multiple OpenGL routines into a single call. It is common to create a toolkit to accompany a library as it allows the library to contain a clean, non-redundant interface for low-level tasks while providing users high level commands to accomplish common tasks. Users who need options not provided by the toolkit can employ the library API.

Beginning with the creation of the Chromium viewer, a viewer toolkit has been developed to accompany CUMULVS. This toolkit is written in C++ and provides easy access to common tasks such as connection, access to fields and parameters, and access to identification information for fields and parameters. While still in pre-release form, this toolkit reduces both the complexity of viewer creation and the knowledge required to accomplish such a task. The relationship between \texttt{xpathview}, the toolkit, and CUMULVS’ viewer library is shown in Figure 47.

![Figure 47: xpathview, the CUMULVS Viewer Toolkit and the CUMULVS Viewer Library](image)

The toolkit is written in C++ and utilizes this language’s generic containers. Field
and parameter names are transported as lists of strings and field dimensions are lists of integers. The toolkit is designed to work with a single simulation, and connection data associated with this simulation are stored and maintained by the toolkit, freeing the user from the task of maintaining and manipulating this information. It would be reasonably simple to provide a C interface to the toolkit; however, such an interface was not required for this project. Enhancements and extensions to this toolkit will be discussed in the next chapter.

5.5 Conclusions

CUMULVS has a full set of field viewers which can be supplemented by user created viewers. The task of creating such a viewer can be simplified with the addition of a toolkit which provides high level routines to accomplish common viewer related tasks. A Chromium enabled viewer couples nicely with CUMULVS to create a parallel graphics front end to a parallel computation.
CHAPTER 6

Parameter Viewers

Parameter viewers are specifically designed to allow the user to interact with the parameters of a computation. On first inspection, these viewers should be trivial and relatively uninteresting, an opinion which appears to be shared by the designers of most STV systems. The facilities for extracting and transferring parameters are included in the system, but minimal attention is paid to developing and presenting a robust set of useful and extensible parameter viewers.

In CUMULVS, the only generic parameter viewers/steerers are text based. These viewers, written in C, are provided as working examples and library test routines. They are not user friendly and are aimed at steering the application. The interface is pictured in Figure 48. It is the author’s opinion that the lack of ready to deploy, easy to use parameter viewers is one of the major factors contributing to under utilization of STV packages by the cluster computing community.

Simulation users are often not only interested in field data in order to discover information based on the results of the simulation, but are equally interested in parameter data to know when such an effort is required. Data such as the temperature of the matter in a molecular dynamics simulation, the number of cells alive in the game of life, or the minimal eigenvalue in *mpipath* may be monitored at every step of the computation. An interface such as the one pictured in Figure 48 offers no improvement, and indeed
Figure 48: The Supplied CUMULVS Parameter Steering Client
may detract from the information delivered in a text file output. The ability to monitor, graphically and in real time, the value of a parameter, or possibly even the relationship between parameters, as pictured in Figure 49, enhances the understanding of the processes involved and increases the usability of the simulation as a tool of discovery. Aside from the viewer presented in Figure 48, all viewers and viewer libraries presented in this chapter are the work of the author.

![Figure 49: Temperature (AK0) vs Minimum Eigenvalue in mpipath](image)

6.1 Viewers Created for CUMULVS

There have been a number of CUMULVS parameter viewers created as a part of this project. These viewers range in scope from simple single-task viewers to a complex steering system for mpipath. The earliest viewers developed tend to be aimed at specific tasks, while later viewers tend to be more general. Not all viewers developed are graphical
in nature, nor do all provide a mechanism for human interaction.

6.1.1 Simple Viewers

Early in the project a set of parameter viewers was proposed [13] and a number have been developed. These viewers, like the X-Y plot viewer in Figure 49, are stand alone and single task oriented. Examples of these viewers include the X-Y plot viewer, the FPS (Frames Per Second) viewer, mentioned in the previous chapter, and the scaview viewers. Each of these viewers will be discussed in turn.

The X-Y plot viewer provides a tool for the user to observe the relationship between parameters in a simulation. Figure 50 contains a screen shot of this viewer along with a terminal containing the real time text output from \textit{mpipath}. The relationship between the iteration number and tangent vector multiplications performed, presented in the upper graph, shows that each time the temperature is changed more iterations are required. The lower graph displays the relationship between the temperature and the minimum eigenvalue of the Jacobian, demonstrating that the computation is approaching a singular point.

The X-Y plot viewer is capable of generating multiple plots from a single instance of the viewer. The interface, which is partially visible in the figure, allows the user to select the parameter to be associated with the X and Y axes, as well as a title for the window. This viewer has proven to be particularly useful, and provides rapid insight into the relationship between parameters displayed. In the case displayed, the computation is approaching a singular point, as evidenced by the decreasing eigenvalue, along with the increased number of iterations required to complete the Newton iteration.
Figure 50: Running *mpipath* While Observing the Relationship Between Multiple Parameters
The Frames Per Second (FPS) viewers are perhaps the most simple parameter viewers to create, and are available as either a GUI or text based viewer. These viewers connect to a simulation and count the number of times \texttt{stv\_sendReadyData()} is called in a given time period. This effectively determines the frames, or iterations per second that a computation has achieved. Based upon the UNIX \textit{uptime} command, these viewers allow a user to monitor the average speed of a program for multiples of a base time interval specified by the user. The viewers are pictured in Figure 51 monitoring a modified version of \textit{MLIFE3d} where each iteration is preceded by a random delay. In this figure, the base monitoring time is one second, with multipliers of five applied to the next two classes, producing a one, five and twenty five second average. The text based interface on the right displays (*) next to numbers where insufficient data is available to properly compute the average.

These viewers were written in C++ using FLTK [70], a cross platform, multiple language, graphical user interface toolkit. These viewers employ a system timer to record data for the base time period. Built from a common code base, these viewers have been shown to have minimal impact on the simulation to which they are connected [16].

The scaview viewers, both GUI and text based, allow a user to monitor the values of parameters. These viewers, pictured in Figure 52, expose some of the difficulties facing viewer implementers. In the figure, the names of the available parameters are listed in the checkbox, allowing the user to select the parameters they wish to view. With no mechanism to provide meta-information about available parameters, there is no possibility to remove parameters, such as \texttt{Quit} which is used for steering, from this list.
Figure 51: Two versions of the Frames Per Second Viewer, a GUI Version on the Left and a Text Version on the Right

Given a method for distinguishing between parameters associated with the computation and parameters associated with steering, generic tools could automatically filter out parameters which make little sense in a given context.

The lack of meta-information is less of a problem with field data as field viewers tend to be constructed especially for viewing a particular simulation. Parameters, on the other hand, tend to be more generic and can be displayed and controlled by more generic devices, such as the viewers described above. This information could be coded into the parameter name, a string, which would provide the information but make accessing parameters far more difficult. A better solution would be to extend the parameter description in CUMULVS to include a meta-data field encoded in XML, the extensible markup language. XML will be discussed later in this chapter. The XML description
Figure 52: Two versions of the scaview Viewer, A GUI Version on the Left and a Text Version on the Right

could include a standard schema for parameter information such as type, value limits, parameter name and usage information. By parsing this field, parameter viewers could perform tasks such as presenting better descriptions of parameters and performing error checking.

A second problem relates to the type of data transmitted. CUMULVS is written in C, which lacks capabilities for true generic programming, forcing viewer implementers to struggle to build routines which properly handle data in a generic way. CUMULVS provides a “data to string” function, which the scaview viewers employ. Using this function, the viewers receive a data string associated with each parameter name, which is suitable for display. This scheme is less successful when the viewer has a need to manipulate the data. There are a number of ways to overcome this problem including building truly polymorphic clients, creating a unique interface for each data type supported, or employing a language capable of dealing with arbitrarily typed data, such as Python. The latter has been adopted for this project.
6.2 A Python Extension to the CUMULVS Toolkit

Python is an interpreted object oriented language that supports dynamic typing. In Python, all data is treated as an object, with a value and an associated type. Types are not checked until an operation is performed on an object, allowing Python to implement truly generic containers and routines. A high level routine can iterate over a generic list of objects, passing each to a low level routine without ever dealing with the object’s type. Low level routines can, through an object’s methods, request the object be interpreted as the type required by the routine. Furthermore, objects provide methods for determining the type of data they contain. This ability to deal with data in a truly generic manner was one of the leading reasons for selecting Python as an alternative language for the extension to the CUMULVS Toolkit.

A number of additional features contributed to the selection of Python. These features include the support for a large number of utilities and libraries, the ability to perform rapid prototyping and application development using this language, and a growing level of acceptance of Python by the scientific programming community [59]. Python provides a set of routines which simplifies the creation of C/C++ to Python interfaces. By combining these routines with Python’s generic containers, the creation of a Python-CUMULVS Toolkit, pyCUMULVS, was reasonably straightforward.

Most interaction with pyCUMULVS is accomplished through lists. In Listing 4 the viewer is attempting to attach to the simulation named LIFE. If the connection is successful, three lists are returned. The lists contain the names of the parameters (vars), normal fields (fields) and particle fields (particles) respectively. The program then
[vars, fields, particles] = pyCUMULVS.pySTVInit('LIFE')
print 'Parameters:', vars
print 'Fields:', fields
print 'Particles:', particles
pyCUMULVS.pySTVHalt()

Listing 4: Initializing the STV Connection in pyCUMULVS

prints the values of the lists and disconnects from the simulation. The output of this
program is pictured in Figure 6.2, where formatting has been altered slightly to better
fit in this document. In this figure the parameter names are returned as a list of lists.
This is done to accommodate CUMULVS’ vector parameters, which consist of a main
parameter name, as well as a name for each element of the vector. Parameters such as
Max Age and Quit contain a single value, while Checkpoint Name, a vector parameter, is
composed of the main parameter name and the names of the four member values: First
Checkpoint Number through Fourth Checkpoint Number. The pattern of dealing with
parameters as a list of lists is continued throughout pyCUMULVS.

Listing 5 contains a complete program which will connect to the LIFE simulation and
watch the parameter Max Age. After connecting to the simulation, the program goes into
a loop checking to see if new data has arrived from the simulation via pySTVNextFrame().
This routine will return the frame number of the current frame, thus, by observing when
this value changes, the program can determine when new data has arrived. The new
value of the parameter is accessed via pySTVParamValue(paramName), which returns a
list containing the values of the parameter. Max Age is a simple parameter, so only the
first field of the returned list is accessed, while the application would be required to
iterate through the list for a vector parameter.
Figure 53: Output from Listing 4

```
[bennett@gandalf python]$ test.py
Connecting to LIFE

CUMULVS Version 1.2.0 (Elwood)
Oak Ridge National Laboratory, Oak Ridge, TN
(C) 1996-1998 All Rights Reserved
CUMULVS was funded by the U.S. Department of Energy.

Parameters: [['Generation Number'], ['Max Age'], ['Cells Alive'],
['Average Age'], ['Restart Board'], ['Quit'], ['Pause'],
['Checkpoint'], ['Checkpoint Name', 'First Checkpoint Number',
'Second Checkpoint Number', 'Third Checkpoint Number',
'Fourth Checkpoint Number'], ['Checkpoint Return']]

Fields: ['board1', 'board2']

Particles: []

[v, f, p] = pyCUMULVS.pySTVInit('LIFE')
paramName = 'Max Age'
frameTime = 0
while frameTime != -1:
    newFrameTime = pyCUMULVS.pySTVNextFrame()
    if newFrameTime != frameTime:
        frameTime = newFrameTime
        newValue = pyCUMULVS.pySTVParamValue(paramName)
        print paramName, newValue[0]
```

Listing 5: A Program to Watch a Single Parameter
if 0 <= pyCUMULVS.pySTVAcquireToken(paramName) :
    pyCUMULVS.pySTVNewParamValue(paramName, paramValue, 0)
    pyCUMULVS.pySTVSendParams()
    pyCUMULVS.pySTVReleaseToken(paramName)

Listing 6: pyCUMULVS Calls to Steer a Parameter

Steering a parameter is the process of changing the value in a viewer and transmitting
the value back to the simulation. The steps required to do this include acquiring a steering
token for the parameter to be steered, setting the value of the parameter, sending the
value to the system and releasing the steering token. The code to accomplish this is given
in Listing 6. The routines in this listing match the tasks above directly.

The field viewing sections of this pyCUMULVS are incomplete for a number of rea-
sons. The primary reason is that field viewer creation with C++ is reasonably simple
and straightforward, thus the need for a Python client is less critical. The interface is
reasonably complete and there is less of a call for generic tools. Secondly, as an inter-
preted language Python has slower performance than a compiled language such as C++.
While parameter data is small, and the manipulation of this data is, for the most part,
not computationally intensive, this is not true for field data. Processing and visualizing
field data can require intensive computations at each data point, thus requiring maxi-
mum performance from the display environment. Completing the pyCUMULVS library
to include processing field data is straightforward and would involve placing the data
into a Python multidimensional array provided by the NumPy library.

A number of additional viewers has been produced using pyCUMULVS. Besides those
mentioned above, two viewers of particular interest are a gateway viewer and a history
viewer. The history viewer, written using the pyFLTK library for GUI components, is pictured in Figure 54, and employs a tab system to present different options to the viewer. The purpose of this viewer is to allow the user to observe the history of a set of parameters. The picture in the upper left illustrates the connection process, with all other tabs inaccessible. The upper right shows how parameters to watch are selected, while the lower left is the active display of parameter values, with the most recent first. The image in the lower right shows how the display can be reconfigured to better suit the user’s needs. The code for this viewer is approximately 400 lines and over 90% is of this code is related to GUI construction and maintenance. The actual CUMULVS related code is remarkably similar to that in Listing 5.

One drawback to the current implementation of CUMULVS is that all participants in the steering and visualization environment must be part of a PVM local machine. This requirement becomes a problem in the face of security restrictions such as firewalls and limited network port access. PVM accomplishes all communications via rsh, the remote
shell, and frequently nodes of a cluster are accessible only through a front end “head node” and consequently are not visible to the outside world. Furthermore, machines attempting to run a viewer must have PVM installed, a software package which is not commonly available on portable platforms. To overcome these limitations a gateway viewer, inspired by the “Gif-O-Matic” field viewer [50] was constructed.

The gateway viewer provides an additional layer of communications. This viewer, which has no screen presence, is responsible for connecting to a CUMULVS instrumented simulation on one side and listening to a set of predefined network ports on the other. The gateway viewer, run on a machine which is part of a PVM virtual machine, is pictured in Figure 55. The viewer becomes a single point of interaction with the simulation for the outside network where all data is exchanged in the form of an XML document.

![Figure 55: The Architecture of a Gateway Viewer System](image)

XML, or the extensible Markup Language, is a standard language for document
markup. By defining a syntax used to create tags, this language provides a convenient, extensible, machine independent means to store and transfer data. When a user defines a set of tags, or an application, the language can be used to create documents which include both data and meta-data. Most languages, including Python and C++, provide libraries to create and parse XML documents [37].

The gateway viewer employs XML to transport data between the viewer and clients of the viewer. A typical client request is given in Listing 7, with the corresponding response in Listing 8. Tags are used to mark the data, such as parameters (<param>), and actions, such as a request for data ( <request>). By adding tags, the data transported, as well as the requests conveyed can be easily extended.

It is relatively simple to employ an XML document in this system. The client generating the request must encode this request as an XML document and transmit this document to the viewer, which will service the request. To provide the service, the viewer parses the XML document, most likely with an existing library, and extracts the information required to service the request. The response is encoded in a new XML document, once again most likely through a series of calls to an existing library, and transmitted to the client.

Figure 56 displays a screen shot of a web browser running a web based client, which is connected to a instance of the gateway client. In this system, the web browser requests a web server execute a common gateway interface (CGI) program written in Python. The CGI program connects to an instance of the Gateway client and submits an XML request similar to that in Listing 7. After receiving a response much like that in Figure 8, the CGI
Listing 7: XML Request to the Gateway Viewer

Listing 8: Data Returned by the Gateway Viewer
program formats the response in the Hyper Text Markup Language (HTML), which is in turn returned to the browser. By setting HTML meta tag `HTTP-EQUIV="REFRESH"` with an argument representing the computed frame rate for the simulation, most browsers will update once per iteration and display current parameter values.

![Frame Number: 14877](image)

The gateway viewer listens at a set of predefined ports to provide a number of different services. The services are currently identified by connection type, one time or continuous, as well as by interactive content. The web browser example uses a one time connection with interaction, where the CGI program requests a subset of the parameters. An equivalent client exists where the connection is done without interaction, in this case the values of all parameters are transferred from the gateway viewer to the gateway client. When connected for continuous operation, the gateway viewer will push out the values of all parameters each time an update from the simulation is received.

The gateway viewer is a tool for receiving data from CUMULVS and exporting this data to clients, which lack the ability to participate in a PVM Virtual machine. An
example of this viewer displaying on a cellular phone is presented in Figure 57. While all current example programs employ web browsers, this is by no means a requirement of the system. A client can be constructed on any system which supports access to network sockets. There are a number of features which could be added in future work to make the system more complete. These features include the ability to steer parameters and transfer field data.

![Gateway Viewer Displayed on a Cellular Phone](image)

**Figure 57**: A Web Browser Connected to the Gateway Viewer Displayed on a Cellular Phone

The gateway viewer is a practical example of how viewers can be used to provide functionality beyond immediate display of data. Other viewers of this type could include tasks such as a notification viewer, which contacts the user when a computation approaches a critical point, to a viewer which employs a set of rules to steer a computation. An example of the latter is a program, which watches the number of iterations in a solver.
and adjusts the path following algorithm employed, as well as the size of the step taken along the path based on a set of rules. The author believes that exploring such viewers provides additional areas for future work.

6.3 An Extensible Viewer

While developing the viewers discussed in the previous sections, a pattern began to appear. In general, most viewers follow the basic code in Listing 5. An initial connection must be made with the simulation, the viewer then loops, checking for new frames and acting upon the new data when it arrives. When a GUI becomes involved, an additional call needs to be added to the main loop allowing the GUI to process events. Algorithm 5 presents an abstraction of this basic loop. This algorithm is a minor variation on an event loop, which has been successfully abstracted into a library in many different application areas.
Algorithm 5 The Basic Viewer Loop

Connect to Simulation

time = 0

while time != -1 do

    newtime = Get Next Frame

    if newtime != time then

        time = newtime

        Process Actions on Parameters

    end if

    Process GUI Events

end while

Many systems which successfully deploy such an event loop include the concept of registered callbacks. A callback is a routine which is called whenever an associated event occurs. In the case of a parameter viewer, there is only one event, the arrival of a new data frame from CUMULVS; however, there is the potential for multiple callbacks associated with that event. Consider Figure 58, a GUI interface for the LIFE simulation. In this interface there are a number of parameters being watched, and a number of different actions occur with these parameters. Callbacks are required to update the totals displayed in the upper left hand portion of the screen, and to update the list of values displayed in the upper right hand corner. The buttons across the top provide steering for the application while the bottom half of the screen provides control over checkpoints.
Building upon these observations, an extensible viewer library has been constructed [15]. This library consists of routines to register callbacks along with the parameters needed in these callbacks, to connect to an application, to steer parameters, and to perform a main loop similar to that outlined in Algorithm 5. It relies on the pyCUMULVS library and as such, supports connection to a single application.

There are two forms of callbacks supported by the extensible viewer library. A normal callback occurs whenever a new frame is received. Such a callback may update the value displayed in the label of a GUI or compute the ratio of two parameters. An idle callback occurs each time the application executes a main loop, and is designed to allow tasks such as GUI event handling. Both forms of callbacks can request a list of parameters to be passed in when the callback is called. Listing 9 demonstrates a complete application written using this library. In this Listing, three different callbacks are registered, two normal callbacks and one idle callback. The normal callbacks request different parameters
def call1(appname, paramList, myList=()):
    print appname, " returned values to call1"
    for i in paramList:
        print "Parameter: ", i[0], " = ", i[1][0]
def idleCall(myList =()):
    print "In idle call"
exServer.addCall("LIFE", ("Generation Number", "Max Age",
    "Cells Alive", "Average Age"), call1, ()
exServer.addCall("LIFE", ("Generation Number", "Max Age"), call1, ()
exServer.addIdleCall(idleCall, ()
exServer.mainloop()

Listing 9: A Simple Client Built with the Extensible Viewer Library

be passed to the routine call1, with no additional arguments. The idle callback requests no additional arguments.

The library maintains a list of callbacks, and the associated parameters. Each time a callback is registered, the library checks for a connection to the simulation, and if none exists, connects. When a new frame is received, the library builds a list of parameter names and values using each callback's associated parameters, and calls the callback.

Two routines are provided to execute the event loop. The application in Listing 9 calls the mainLoop() routine which takes control of the application. For applications which must retain control of the main loop, checkForSTVUpdates() will perform a single iteration of the loop, calling normal callbacks if necessary, but omits a call to the idle callback.

A number of viewers, including the LIFE viewer pictured in Figure 58 have been produced using this framework. In the process, this framework has proven equally useful for creating GUI and non-GUI viewers. It can be employed in the construction of viewers
which steer parameters as well as those which only provide parameter visualization. The library is complete and functional.

6.4 Python CUMULVS GUI Objects

In order to simplify the development, provide a consistent API, and ease the task of learning to use a GUI toolkit such as FLTK, developers have turned to object oriented methods. A base class, often with no screen presence, is developed containing the elements common to all objects in the library. From this base class, specialized sub classes are developed, which inherit properties and interfaces from the base class. This process continues as more and more specialized objects are created to perform specific tasks.

An example of such a hierarchy in the context of an STV environment is pictured in Figure 59. This figure shows how a Labeled Parameter Viewer is derived from a Parameter Viewer which is in turn derived from a Base Class. The Base Class contains methods for receiving data. The Parameter Viewer adds a screen presence, plus the ability to display the value of the parameter associated with this object on the screen each time the parameter is updated. The Labeled Parameter Viewer adds the display of an associated label.

![Figure 59: A STV-GUI Object Hierarchy](image)

The objects in an STV environment have two different and distinct sets of properties.
They behave both as GUI objects, with a screen presence and interaction with input devices, and also as STV objects, which receive and possibly return parameter values. Such objects are often derived from two different base classes and are said to have multiple inheritance. This relationship is pictured in Figure 60. In this figure, the Parameter Viewer inherits properties from both the STV Base Class and the GUI Base Class, and has the properties of both. By providing a GUI Base Class, the system can easily be adapted to use different GUI toolkits.

```
+----------------+    +----------------+
| GUI Base Class |---| STV Base Class |
+----------------+    +----------------+
        |                      |
        | Parameter Viewer     |
```

Figure 60: A STV-GUI Object Hierarchy with Multiple Inheritance

Simple objects are combined to create more complex objects. The Labeled Parameter Viewer above takes a single parameter, and updates the display as new values are received from the simulation. An equivalent Parameter Steerer presents the user with an input box. When the user enters a new value into the input box, that value is sent to the simulation to update an associated parameter. By combining these two objects into a new object, ShowAndSetParameter the library is extended to produce an element, pictured in Figure 61 which can be used to display and steer a single parameter.

```
Set To
```

Figure 61: A GUI Object to Display and Steer a Single Parameter

The ShowAndSetParameter object is generic in that it can be used with any parameter of any type. The interface pictured in Figure 62 is constructed of such objects. These
objects are part of a prototype library, called *pyGUIObjects*, which is being developed as part of this project. The aim of this library is to allow easy creation of graphical viewers.

The *pyGUIObjects* library is composed of two basic elements, clients and connections. A client can be either a viewer or steer a parameter as described above. Clients receive parameter data from a connection and return steering information to the connection. Clients have no knowledge of CUMULVS, but rely on the connection for this service. Code to create the plot display in the lower left hand corner of Figure 62 is given in Listing 10. This employs a *PZ_XYScatterPlot* object, which plots the values of two parameters. The second and third line in this listing are used to place the object on the screen. The last line connects the object to a connection object, in this case an instance of *plotFilter*, which will be discussed later.

Connection objects are the second basic type of object in the *pyGUIObjects* library.
MainPlot = PZ_XYScatterPlot("NewPoint ak0","NewPoint eigenvalue",10,10,60)
MainPlot.SetPosition(width - MainPlot.w() - 20, 300)
MainPlot.GUIElement()
MainPlot.SetConnection(plotFilter)

Listing 10: pyGUIObjects Code to Create a Scatter Plot

collection = Connection.Connection("pathprog")
# code to Establish GUI
# code to build client objects
while 1:
    Fl.check()
    connection.DoUpdate()

Listing 11: pyGUIObjects Connection Code

These objects are responsible for interaction with CUMULVS, via pyCUMULVS, and with client objects. Code to deploy a base connection object is demonstrated in Listing 11. In this code, a connection object is instantiated and employed. Code to create the windows and to create the client objects has been deleted. The object is instantiated with the name of an application, in this case pathprog, to which it should connect. As a part of this process, the connection object will call pyCUMULVS routines to connect to the application and begin steering. When the doUpdate() method is called, the connection object checks to see if a new frame is available from CUMULVS. If such a frame is available, the connection object delivers registered parameters to any connected client objects.

A filter object inherits properties from both the client object class and the connection object class. Filters are responsible for receiving data from a connection object, applying a filter to this data, and sending the data to a client object when appropriate. The filter
plotFilter = StaticFilter("Global State", PlotFilterFunction);
plotFilter.SetConnection(connection)

Listing 12: pyGUIObjects Filter Code

will also receive steering data from a client and pass it along to the connection object.

The filter object in listing 10 is an instance of a StaticFilter object, which watches a
single parameter, and passes the data to the clients when the parameter value causes an
associated function to evaluate to true. Code establishing this filter is given in Listing
12. The interaction of these objects is displayed in Figure 63.

Figure 63: The Interaction Among pyGUIObjects

Using the pyGUIObjects library simplifies the task of creating GUI steering interfaces.
Pictured in Figure 64 is an interface created with these objects. This viewer is hard coded
to connect to the LIFE simulation and watch the parameter Restart Board, but could
be made completely generic by allowing the user to specify these values via command
line arguments. The code needed to produce this viewer is given in Listing 13. With the
underlying pyGUIObjects, this listing demonstrates how a fully functional CUMULVS
GUI can be constructed in relatively few lines of code. This code follows coding pattern
commonly deployed for the construction of a GUI, and programmers familiar with such
an environment should be able to rapidly deploy the pyGUIObject library.
from fltk import *
from ShowAndSetParam import *
import Connection

window = Fl.Window(440,30)
connection = Connection.Connection("LIFE")
client = ShowAndSetParam("Restart Board", "Restart")
client.GUIElement()
client.SetConnection(connection)
window.end()
window.show()

while 1:
    Fl.check()
    connection.DoUpdate()

Listing 13: The Full Code for Life Reset

6.5 Conclusions

This chapter has presented a number of tools which have been developed to assist programmers in creating parameter viewers for CUMULVS. These tools range from generic tools, which simplify deploying the CUMULVS API, to specialized tools for constructing GUI viewers. By employing these tools, the task of creating parameter viewers for CUMULVS has significantly been simplified. These tools will be employed in the next chapter to build an interface to allow interaction with mpipath, especially to assist in the location of singular points and steering this computation through such points.
CHAPTER 7

Steering Through Singular Points

The ultimate goal of this project is to provide the tools to steer a parallel computation through a singular point. This chapter describes how the tools and techniques discussed in the previous chapters have been combined to accomplish this goal. To illustrate how the task of steering a computation through a singular point can be achieved, *mpipath* has been instrumented, and as needed, modified. This simulation is known to encounter singular points [6], but has no mechanism for detecting or managing such points. The creators of *mpipath* had planned to incorporate steering at singular points; however, time constraints and programming difficulties left this task unaccomplished [64].

7.1 Locating a Singular Point in *mpipath*

A number of modifications have been made to this simulation. Beginning with an implementation of Algorithm 3, the code was instrumented with CUMULVS and the eigenfinder PARPACK was employed to create a product which closely resembles Algorithm 4. These modifications required little change to the existing code and demonstrated that further work was required.

Figure 65 shows a scaled temperature value ($AK_0$) plotted against the minimum eigenvalue of the Jacobian ($\lambda_{min}$). This data is from a $22 \times 22 \times 22$ cube with radial boundary conditions. This simulation begins with an initial value of $AK_0 \approx 1.0$ and $\lambda_{min} \approx 0.6$. For convenience, points on the path, such as the starting point ($AK_0 \approx 1.0$, $\lambda_{min} \approx 0.6$) will be
represented as an ordered pair \((AK0, \lambda_{min})\), or, in this case \((1.0, 0.6)\). The simulation ends when \(\lambda_{min}\) becomes negative, which occurs just below \(AK0 = 0.47\). From this experiment, it can be determined that a singular point is somewhere between \((4.95 \times 10^{-1}, 3.91 \times 10^{-2})\) and \((4.68 \times 10^{-1}, -1.90 \times 10^{-2})\). The last point, displayed in red, has a negative minimum eigenvalue, which caused the simulation to exit. Data in this plot was extracted from the simulation with CUMULVS and displayed using a pyGUIObject viewer. No steering was performed in this experiment.

![AK0 vs Minimum Eigenvalue](image)

Figure 65: The Temperature \((AK0)\) vs Minimum Eigenvalue of the Jacobian \((\lambda_{min})\)

Many numerical algorithms have additional parameters, or include tolerances, which must be adjusted by the user for a given situation. The step size in pseudo-arc continuation is one such parameter. The mechanism to adjust the size of this step in *mpipath* is programmatic and is based on iteration counts, which must be adjusted by the user.
before the program is run. By providing the ability to steer these parameters, a user can make adjustments as the program is running to achieve better results.

Instrumentation was added to mpipath to allow the user to modify some parameters to the numerical algorithms, and a user interface was created to permit interactive steering of these parameters. This interface, pictured in Figure 66, allows the user to control parameters including Qdelta. Created using the pyGUIObject library from the previous chapter, the interface also allows the user to control the operation of the computation (step, pause, exit), and to save points along the path, both of which will be discussed later.

![Figure 66: An Interface to Steer Path following Parameters.](image)

Using the interface in Figure 66 to adjust the value of Qdelta the user is able to approach the singular point much more closely. Figure 67 displays the results of this process with the inset graph depicting the points on the path closest to the singular point. The simulation was restarted and allowed to run to until $\lambda_{\text{min}} < 0.1$. At this point, the value of Qdelta was adjusted manually to approach the singular point. The results of steering, pictured in the figure, are obtained by setting Qdelta to values of 0.25, 0.1, 0.025, 0.005 and 0.001. Through this process, the location of the singular point
was determined to be between $(4.7465 \times 10^{-1}, 2.1 \times 10^{-5})$ and $(4.7462 \times 10^{-1}, -4.1 \times 10^{-5})$.

While the ability to modify the parameter $Q_{\text{delta}}$, and view the values of $AK_0$ and $\lambda_{\text{min}}$ are sufficient to approach a critical point, the ability to do this is greatly enhanced by adding control over $mpipath$’s operation.

![Graph](image)

Figure 67: The Temperature ($AK_0$) vs Minimum Eigenvalue of the Jacobian ($\lambda_{\text{min}}$) as a Result of User Steering

7.1.1 Increasing Program Control in $mpipath$

Without code modification, the normal flow of an instrumented program is given in Algorithm 6. As described previously, all data is transferred between viewers and simulation in the call to `stv_SendReadyData()`. In this situation, the user must provide steering data to CUMULVS before the update occurs; however, the user may require
data from the update to determine what steering data is necessary. For example, to
determine the new value for $Q_{\text{delta}}$ it is necessary to observe the change in the minimal
eigenvalue, which is not available until after the data update has occurred. In the default
configuration, steering is always one step behind the computation.

Algorithm 6 The Basic STV Loop

\begin{verbatim}
while Not Finished do
    Perform Main Loop
    stv_SendReadyData()
end while
\end{verbatim}

To correct this problem, the code has been instrumented with a set of program control
parameters. These parameters are used to alter the flow of the simulation in a number
of ways to give the user better control over steering and provide access to the tools that
are part of the simulation. Algorithm 7 presents the changes needed to supply simple
program control such as pausing the main loop and exiting the program. Default values
can be set to allow the program to run without steering, i.e. always continue and never
exit, so that the original behavior of the program is preserved when no viewer is attached.

Algorithm 7 The STV Loop Modified for User Control

\begin{verbatim}
while Program Control Parameters Do Not Indicate an Exit do
    Perform Main Loop
    repeat
        stv_SendReadyData()
    until Program Control Parameters Signal A Continue
end while
\end{verbatim}
By adding the changes in Algorithm 7, a user can observe the state of a computation, pause when necessary, and provide steering information before continuing. This change is relatively minor, and has very little impact on the performance of the simulation until the user desires to pause the computation. The buttons at the bottom of the viewer in Figure 66 provide a means to control \textit{mpipath}. The \textit{Exit} button allows the user to exit the path following loop, the \textit{Step} button allows the program to step through the loop once, \textit{Run} allows continuous operation of the loop and \textit{Pause} interrupts continuous operation. These modifications, along with the steering viewers mentioned above, provide an easy to apply method for approaching a critical point.

7.1.2 Additional Tools for Locating A Singular Point

Two additional tools have been implemented to assist in locating singular points. A system for storing path points in an array, with associated control parameters to provide access to this array, supplies users with a short term working storage much like that on a scientific calculator. This memory is implemented as an array of pointers to a structure which contains all information relating to a point on the path. Both the array and the structure are additions to the simulation, along with all code required to store data into and retrieve data from the memory. The impact of these structures on a simulation which is running without user intervention is minimal, as the only structure allocated is the array of pointers.

Access to the memory is provided by two viewer elements. The interface, pictured in Figure 66, contains elements to store the current path point to and load a point from the memory. These elements are located in the upper right hand side of the interface.
and consist of two instances of the pyGUIObjects StaticListParameterSteerer. This object is instantiated with a parameter and a list of labels and parameter values. When the user selects a new label from the drop down menu, the parameter is set to the associated value. In this case, the user has selected the label Pos 1, so the associated parameter, Save Position is set to the value of 1. To support this action, the simulation has added code which checks the variable associated with Save Position after the call to \texttt{stv\_SendReadyData()}. If the value has changed the simulation stores the current path point into the appropriate memory location. Loading data from the memory is accomplished in an identical manner.

To increase the usefulness of this memory, a second viewer element presents information related to data stored at each location. Pictured in Figure 68, this interface shows the ten memory locations. Data has been stored in a number of locations and the user is able to examine the associated values of $\mathbf{AK}0$ and $\lambda_{\text{min}}$. The field on the right allows the user to make notes related to the contents of individual memory locations. This interface is constructed from pyGUIObjects.

The memory, and associated interface elements, improve the users ability to explore the impact of parameter settings on the simulation. By saving a path point to memory, the user establishes a safe checkpoint. Applying a set of parameter values at this checkpoint, running the simulation and observing the results provides insight into the operation of the simulation. Restoring the checkpoint by loading the point from memory, the user can return to the previous state and apply a different set of values to the parameters of interest.
This memory could be further enhanced by adding the ability to transfer data in a memory location to and from a file on disk. While the code to complete this modification is relatively simple, to be most useful this would require an effective way to transfer filenames and thus strings, through CUMULVS, and has been deferred until this enhancement to CUMULVS has been completed.

The second tool for locating singular points involves the function given in Equation (19). This function takes the solution to Equation (8) and returns the minimum eigenvalue of the Jacobian ($\lambda_{\min}$). Applying numerical methods for locating a root of an equation, such as the secant method or the bisection method, allows a user to approach the point where $\lambda_{\min} \approx 0$, thus locating a singular point.

$$F(G(Q, T)) = \lambda_{\min}$$  

(19)

The bisection method requires two points on a curve, $x_p$ and $x_n$, one on each side of a root. To employ this method, compute $x_c = \frac{1}{2}(x_p + x_n)$ and evaluate $y = F(x_c)$. If $y < 0$
then let $x_n = x_c$, otherwise let $x_p = x_c$. This method will eventually find a value for $x_c$ such that $F(x_c)$ is arbitrarily close to 0. The initial values for $x_p$ and $x_n$ are obtained from the path following routine.

Figure 69 depicts the results of applying the bisection method to $mpipath$. The simulation was restarted and allowed to run until a negative value for $\lambda_{\min}$ was encountered. After the simulation exited the path following routine, with results identical to those pictured in Figure 65, the bisection method was applied. After 14 iterations, the singular point was identified to be between $(4.746376 \times 10^{-1}, 3.6 \times 10^{-6})$ and $(4.746358 \times 10^{-1}, -3.2 \times 10^{-7})$.

![Figure 69: Applying the Bisection Method to $mpipath$](image)

The secant method, represented by the recurrence relation given in Equation (20) attempts to locate a root of a function by approximating the function using a line through
two known points on the curve. These two points, $x_{n-1}$ and $x_{n-2}$, can be obtained from the results of the path following routine. This algorithm can converge much more quickly than the bisection algorithm [68]. The interface pictured in Figure 70 shows that starting from the same initial points as above, the simulation has found the singular point to be located at $AK0 \approx 4.746360 \times 10^{-1}$ in 3 iterations.

\[
x_n = x_{n-1} - \frac{f(x_{n-1})}{f(x_{n-1}) - f(x_{n-2})} (x_{n-1} - x_{n-2}) \tag{20}
\]

Figure 70: Data From Applying the Secant Method to a Run of mpipath to Find a Singular Point

Implementing these algorithms as part of mpipath, or any simulation under consideration for singular point detection, is relatively straightforward. The tools required to complete these computations are most likely present in the existing code and can be reused for these routines. Employing these algorithms, however, may present more of a challenge.

The mpipath program at the beginning of this project was designed to simply follow the state of the liquid crystal slab as temperature changed. The data to represent a
point on the path was spread over a number of seemingly unrelated data structures, with no convenient means for storing or manipulating multiple path points as required in the above algorithms. In this case, identifying all critical data values for aggregation into a single data structure was not a trivial task. Once the task of identifying and aggregating the data was complete, the task of restructuring the code began.

7.1.3 Restructuring *mpipath*

As stated in Chapter Two, the core algorithm for *mpipath* is given in Algorithm 2, and modified to match Algorithm 3. This algorithm was implemented as a routine called `pathFollowing()`. The overall structure of this version of the program is given in Algorithm 8. This control structure is sufficient for following a path which encounters no singular points, but must be modified to respond properly when such a point is encountered.

**Algorithm 8** The Original *mpipath* Control Algorithm

1. Initialize System
2. Perform one Newton Iteration
3. Call `pathFollowing()`
4. Print Statistics

To test components, and in an attempt to preserve the batch processing environment, a control program matching Algorithm 9 was created. Each of the lower level routines produces return values which indicate the exit status for the routine. No routine is executed unless the previous routine indicated that the conditions are correct for such execution. Finally, all routines have been instrumented with control variables to allow
interactive steering, but also have reasonable default values to allow for batch operation. For some initial conditions, this algorithm can, without outside intervention, locate and successfully follow a path out of a singular point.

Algorithm 9 The Modified mpipath Control Algorithm

Initialize System

while $T < T_{end}$ do

Perform one Newton Iteration

Call pathFollowing()

Find the singular point

Find a path out of the singular point

end while

Print Statistics

In order to allow the most flexibility, a final version of the mpipath control program was created. As with the previous version, the new routine relies upon existing functions, thus maintaining the compatibility with the batch program. This version, represented in Algorithm 10, relies upon the variable USER_ACTION, which has been instrumented and is steerable. This variable has an associated set of predefined constants which are used to determine what action the user wishes to take. Using the interface pictured in Figure 71, the user specifies points to be used and the portion of the system to be executed.
Algorithm 10 The Final Interactive Version of the \textit{mpipath} Control Algorithm

Initialize System

\textbf{while} USER\_ACTION is not EXIT \textbf{do}

Communicate with CUMULVS

\textbf{if} USER\_ACTION is NEWTON \textbf{then}

Perform one Newton Iteration

\textbf{else if} USER\_ACTION is PATHFOLLOWING \textbf{then}

Call pathFollowing()

\textbf{else if} USER\_ACTION is ROOTFINDER \textbf{then}

Find the singular point

\textbf{else if} USER\_ACTION is PATH\_OUT \textbf{then}

Find a path out of the singular point

\textbf{end if}

\textbf{end while}

Print Statistics

\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
Main & Path & Root & Secant & Elsction & Qdel & Brs & Singular & RCIRBEKS & Path Out \\
\hline
Aki & Lambda\_min & Gdelta & Save Position & Load Position & \\
\hline
Current Point & 5.00523e-01 & 5.331600e-02 & 1.000000e+00 & \downarrow & \downarrow \\
\hline
Negative Point & -5.000000e-01 & -5.331600e-02 & 1.000000e+00 & \downarrow & \downarrow \\
\hline
\end{tabular}

Figure 71: The Main Steering Interface for \textit{mpipath}

Using this interface, a user can employ path following until $\lambda_{min}$ becomes negative, then employ a root finding routine, such as the secant method. Occasionally, such a root
finding routine will produce a point on a new path, which the user may wish to explore. The user can exit the root finding routine and start path following from this new point, or save the point in memory for later exploration and continue the process of locating the singular point.

Using the method described in Algorithm 10, interactive routines for each of the major components, path following, root finding, and singular points, have been created. Each routine relies on the variable `USER_ACTION` to select the next action the user wishes to perform, provides the user with the ability to load or store points to the memory, and provides access to sub-tasks within the main task. A typical interface, the root finding interface, is pictured in Figure 72.

In this interface, the user is able to load and store one of two points used in the root finders. A menu of choices, including three root finding methods, allows the user to select the sub-task they wish to perform. The interface also contains a dialog box instructing the user how the points will be used as input and output to the sub-tasks.

![Figure 72: The Interface for the Root Finding Segment of mpipath](image)

Sub-tasks generally are provided with a user interface as well. The interface to the secant method is pictured in Figure 73. This interface displays the points involved in
the computation, provides the user with control of relevant steering parameters, and allows the user to control the flow of execution. This interface is typical of the sub-task interfaces, which are built from a common pyGUIObjects.

![Figure 73: The Interface for the Secant Method Segment of mpipath](image)

All control elements are combined into a single element of the user interface, with individual elements accessed through a tab bar. Pictured in Figure 74, this interface indicates the active element with a green tab. In this case, mpipath is executing the path following routine. Other tabs, marked in yellow, are available for viewing. In the figure, the user has selected the interface to the PCIRBLEIGS package, and steering control for this package is visible.

![Figure 74: The Interface Total mpipath Control](image)
7.2 Steering *mpipath* Through Singular Points

Techniques to classify a singular point, as well as to generate the paths out of such a point are described in Chapter Three. These techniques have been incorporated into *mpipath* and instrumentation has been added to provide access to and control over these algorithms.

In particular PCIRBLEIGS has been employed to allow the user to find the algebraic multiplicity of $\lambda_{\text{min}}$ and thus calculate the dimension of the null space of the Jacobian. The interface to PCIRBLEIGS is shown in Figure 75. This interface provides control over a number of parameters, displays the computed minimum eigenvalues of the Jacobian, and allows the user to set the dimension of the null space. In the figure, the algebraic multiplicity of $\lambda_{\text{min}}$ appears to be one, as only a single eigenvalue close to zero has been returned. Using the interface, the user sets the dimension of the null space to be one and proceeds with the tasks of point classification and locating points on paths out of the singular point.

![Figure 75: Finding the Dimension of the Null Space with PCIRBLEIGS](image)

Applying the point classification techniques from Chapter Three, the point is identified to be a simple bifurcation point. The solutions to the ABEs are pictured in the
scrollable box at the bottom of Figure 76, the interface to finding paths out of a singular point. This interface allows the user to employ solutions to the ABEs in Equation (16) to compute the tangent of the associated path exiting the singular point. Using this newly calculated tangent, a Newton iteration is employed to generate a new point on the path. When a point on a new path is discovered, the user can exit the routine and explore this path, or save the point to memory and attempt to locate additional paths out.

For this experiment, solutions one and two represent the current path, while solutions zero and three represent alternative paths. The results of applying these solutions are pictured in Figure 77. The point indicated in red results from applying the solutions to the ABEs to the vectors of the null space for solutions zero and three.

After paths out of a singular point are determined, the process of path following is repeated. Figure 78 represents following the path out of the bifurcation point and into the next singular point. This process is sensitive to variations in parameter values, which must be carefully adjusted to successfully exit a singular point.

There are a number of interesting behaviors in Figure 78 which bear further investigation. Approaching the first singular point, there appears to be a dramatic change...
Figure 77: The First Step on a Path Out of the Singular Point

Figure 78: The Path to the Next Singular Point
in the nature of the path. The slope of the curve appears to change dramatically and the pseudo-arc takes a larger than expected step along the path. This phenomenon can be explored by stopping the computation when $\lambda_{\text{min}} \approx 0.2$ and adjusting the value of $Q_{\text{delta}}$ appropriately to more closely observe the behavior of the curve.

The second feature occurs as the path exits from the bifurcation point. Somewhere near the point where $\lambda_{\text{min}} = 0.15$, the curve exhibits discontinuous behavior. This behavior is depicted in Figure 79. Basic parameter adjustment (changing $Q_{\text{delta}}$) has been insufficient to produce a satisfactory explanation for this behavior. A more detailed study, where extremely small step sizes are employed may yield better results.

![Figure 79: A Discontinuous Break in the Path Out of a Singular Point](image)

### 7.3 Conclusion

The example in this chapter demonstrates that singular points can be located, identified, and the paths out computed using the tools produced in this project. Deploying the tools in a simulation not designed to perform these tasks requires a major restructuring
of the top level control code. Successfully employing the tools requires instrumentation, and a steering interface. This chapter has presented solutions to both of these problems.
CHAPTER 8

Conclusions and Future Directions

The previous chapter demonstrated the ability to steer a parallel simulation through a singular point. This process involved extracting data from the simulation, implementing routines to add functionality to the simulation, and modifying the simulation to allow for interactive steering. While these tasks were performed on a specific simulation, \textit{mpipath}, they can be adapted to any simulation which employs path following.

Instrumenting code using a STV package like CUMULVS is reasonably straightforward. A number of tutorials demonstrate this process and the documentation for the interface is complete. Performing this task provides the simulation with interactive data extraction and manipulation capabilities and enhances the ability to observe, and control the operation of the simulation.

Providing the additional routines required for the location and classification of singular points most likely relies on existing functionality within the simulation. Both ARPACK and PCIRBLEIGS require a vector matrix multiply routine, which is likely to exist in any path following application. Other routines, such as root finding and applying the ABEs to find a path out, employ a solver, which again is likely to be be part of any simulation employing path following. Finally, tasks such as finding the solutions to the ABEs or classifying the singular point rely on very low level routines such as \textit{axpy}, a routine to compute $Y = aX + Y$ where $a$ is a constant and $X$ and $Y$ are vectors, which
are the building blocks for most matrix related programs.

The final task, modifying the simulation for interactive steering, should follow the pattern described in Chapter Seven. Steerable control variables added to the code provide a mechanism to select the task to perform. These control variables provide access to modules to perform tasks such as root finding, identification and classification of singular points, and finding paths out of a singular point. Building a memory to store points on the path completes this task and allows the user to fully explore the singular point.

The task of creating an interface which allows users to interact with the modified simulation described above can be greatly simplified by employing tools such as pyGUIObjects described in Chapter Six. Recognizing the importance of providing powerful tools which are easy to use, this library permits rapid development of CUMULVS viewers for steering any computation. The author believes that the tools developed represent a major improvement in the usefulness and usability of the CUMULVS package as these tools, while developed specifically for this project, are both generic and extensible. As a result, these tools can be used to rapidly develop parameter viewers for any parallel computation.

8.1 Future Directions

As with many research level projects, the work described in this document serves as a foundation for future development. A number of follow on projects, both theoretical and applied, are natural extensions of this work.

The most natural continuation of this project is to continue to explore the behavior of mpipath by employing the tools developed. Further exploration should not only provide insight into the underlying mathematics, it will also provide guidance for refining existing
tools and inform the development of new tools.

A second direction is to apply the results of this work to a different simulation which employs path following. This project is likely to suggest improvements and extensions to the current work. Additionally such a project will expose patterns in deploying these tools and thereby enable the development of more generic techniques and interfaces to the existing tools.

A more applied project is to continue to develop PCIRBLEIGS into a product ready for full release. This work includes providing full documentation, extending the example programs, providing full functional equivalence with IRBLEIGS and developing interfaces for other languages. Further work could include comparison of performance results with existing packages. This project will potentially provide a valuable asset to the scientific computing community.

Chapter Four concluded with a number of suggestions for improving the functionality of CUMULVS. Implementing these changes, completing the documentation of the viewer interface, and providing full, well documented examples is a worthwhile project. Such a project will improve CUMULVS’s usefulness to the parallel computing community in general and allow for improved steering and better functionality when locating singular points.

Development of the user interface construction tools described in Chapter Six will be continued. As stated in that chapter, it is the author’s opinion that the availability of these tools greatly impact the usability of CUMULVS. A significant contribution can be made by developing a production quality set of tools which are well documented and
have adequate example applications.

These packages, the CUMULVS toolkit and pyGUIObjects, are currently research quality software. The naming conventions and interfaces are inconsistent, documentation is mostly non-existent, and there are few example programs. The work of preparing these products for public release is the next project priority. These products will be employed to steer additional parallel computations to expose missing components, documentation and example programs will be written.

Finally, a prototype GUI based drag and drop interface builder has been developed. This prototype employs pyGUIObjects and allows the user to construct interfaces composed of these objects graphically, at run time. When developed, this tool will allow users to develop a user interface without requiring programming ability.

The suggestions from the previous paragraphs illustrate that there are excellent opportunities to continue the work carried out in this project. These extensions represent work in the area of computer science, numerical analysis, and cross disciplinary projects. This work can be performed at all levels, from undergraduate projects through original research.
BIBLIOGRAPHY


APPENDIX A

PCIRBLEIGS

A.1 Overview

IRBLEIGS is a Matlab program for computing a few eigenvalues and associated eigenvectors of a large sparse Hermitian matrix. This program was developed by J. Baglama, D. Calvetti and L. Reichel [7] and is available from Baglama’s website at http://www.math.uri.edu/~jbaglama/software/irbleigs.m. CIRBLEIGS is a port of this program to C++. PCIRBLEIGS is a parallel implementation of CIRBLEIGS using MPI, the Message Passing Interface, for parallelization. This port was performed by Dan Bennett who currently maintains this package. A copy of this package is available at http://mirkwood.cs.edinboro.edu/software/CIRBLEIGS.

This program implements the implicitly restarted block Lanczos method. This method can compute the algebraic multiplicity of an eigenvalue up to a given block size $b$, and find the associated eigenvectors. For full details, please see [7].

A.2 Copyright

This document is part of PCIRBLEIGS/CIRBLEIGS.

PCIRBLEIGS/CIRBLEIGS is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

PCIRBLEIGS/CIRBLEIGS is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
A.3 Requirements

Both packages employ the LAPACK package.

Both PCIRBLEIGS and CIRBLEIGS assume the existence of a matrix multiplication function which must be supplied by the user. This function must take a skinny matrix $X$, where $X$ is a $n \times m$ matrix where $n >> m$, and return the result of $A \times X$, where $A$ is the matrix for which the eigenvalues will be found. This requirement permits the software to operate when no explicit representation of the matrix $A$ is given.

PCIRBLEIGS assumes that MPINIT() has been called. It further assumes that the random number generator has been initialized with srand48() and each process has employed a unique seed.

A.4 Software Status

The current release is version 0.1. This release is the result of a research project and should be considered an alpha release. Functionality is limited to finding minimum and maximum eigenvalues. Future releases will improve the documentation, example sections, and will expand the functionality of the package.

A.5 Interface

P/CIRBLEIGS maintains compatibility with IRBLEIGS whenever possible. Descriptions of most of the parameters is taken directly from the IRBLEIGS documentation, and interested readers are directed to that documentation.
P/CIRBLEIGS were ported by replacing low level OCTAVE functions with C++ functions. The process was repeated by working up the call tree until the main routine was replaced with a C++ function. Due to this process, some parameters to PCIRBLEIGS seem to have been retained which serve no purpose. These parameters should be removed in future versions of this library. Other parameters, such as those related to finding eigenvalues about an arbitrary value are currently not used as this functionality has not yet been implemented. This interface is subject to change in future releases.

To find a few eigenvalues of the matrix $A$, invoke the function, `cirbleigs` with the following arguments:

```c
matrix (*mply) (matrix &)
```

A pointer to the matrix multiply routine. This routine takes a `skinny` matrix, either `myMatrix` for the sequential program or `pmatrix` for a parallel program $X$, and returns the result of $AX$.

```c
int maxit
```

The maximum number of iterations or block Lanczos restarts to perform.

```c
int nbls
```

Number of blocks in the Lanczos tridiagonal matrix. The program may increase NBLS to ensure certain requirements in [1] are satisfied. A warning message will be displayed if NBLS increases.

```c
myMatrix & ninpeigvec,
```

An initial set of eigenvectors is to be used, this matrix contains the
number of initial eigenvectors given. This parameter may change in future versions of the code.

vector <double >& eigval
A matrix of converged eigenvectors.

pmatrix & eigvec
A matrix of converged eigenvectors.

int blsz
Block size of the Lanczos tridiagonal matrix.

int K
Number of desired eigenvalues.

pmatrix & V,
the matrix V contains the corresponding eigenvectors

myMatrix & permM
Permutation vector for the Cholesky factorization of M(PERMM,PERMM). When the input matrix M is replaced with R where M(PERMM,PERMM)=R'*R then the vector PERMM is the permutation vector.

string funpar
Unused. This parameter may be eliminated in future versions of the code.

int n
double tol

Tolerance used for convergence. Convergence is determined when \( \|Ax - \lambda x\|_2 \leq TOL \|A\|_2 \|x\|_2 \) is approximated by largest absolute Ritz value.

string computvec

Should the eigenvectors be computed. 'T' if the vectors are requested.

int dispr

Indicates if K Ritz values and residuals are to be displayed on each iteration. Set positive to display the Ritz values and residuals on each iteration.

vector <double >& nval

Not used.

string sigma

Two letter string or numeric value specifying the location of the desired eigenvalues. 'SE' Smallest Real eigenvalues. 'LE' Largest Real eigenvalues.

string zertyp

Two letter string to indicate which type of zeros to apply. 'WL' - Weighted fast Leja points. The weight functions are used to help increase convergence. 'ML' - Mapped fast Leja points. Fast Leja points are computed on \([-2,2]\) and mapped to the dampening interval. This option is not available when sigma is a numeric value NVAL.

int sizint

size of the dampening interval. Value of 1 indicates consecutive Ritz values are
used to determine the endpoints of the dampening interval. Value of 2 indicates endpoints are chosen from Ritz values that are separated by a single Ritz value. A value of 3 indicates endpoints are chosen from Ritz values that are separated by two Ritz values. Etc. The minimum value is 1 and the maximum value is \((\text{NBLS}-1)\ast\text{BLSZ-K}\). The program may modify \text{SIZINT} without notification to ensure certain requirements in [1] are satisfied.

\textbf{myMatrix \& M}

Not used

\textbf{string endpt}

Three letter string specifying the location of the interior endpoints for the dampening interval(s). 'FLT' - Let the interior endpoints float. 'MON' - Interior endpoints are chosen so that the size of the dampening interval is increasing. This creates a nested sequence of intervals. The interior endpoint will approach the closest Ritz value in the undampened part of the spectrum to the dampening interval.

\textbf{int maxdpol}

Numeric value indicating the maximum degree of the dampening polynomial allowed.

\textbf{int \& iter}

The number of iterations taken.

\textbf{int \& Tsz}

Used internally by the routines. This parameter may change or be eliminated in
future versions.

\texttt{vector<double>& ritz}

The computed Ritz values.

\texttt{myMatrix & ritzvec}

The computed Ritz vectors.

A.6 Skinny Matrix Class

To facilitate the development of this library, a \textit{skinny} matrix class has been implemented. This class relies on routines from LAPACK (http://www.netlib.org/lapack/). The class is implemented as \texttt{myMatrix} and represents a $n \times m$ matrix where $n >> m$.

The member functions of the matrix class are not necessarily consistent with respect to indexing. Some routines use FORTRAN style indexing (1, .. n) others C style (0, ..., n-1). This was done to simplify the port from MATLAB/Octave, which uses FORTRAN style indexing.

Information hiding is not employed in this class. This decision was made to reduce function call overhead when desired.

The following methods are supported by the class \texttt{myMatrix}:

\texttt{myMatrix()}

Create an instance of \texttt{myMatrix}

\texttt{myMatrix(const myMatrix & src)}

Create a new instance of \texttt{myMatrix} by copying an existing instance.
myMatrix(int row, int col)

Create a new instance of myMatrix of a given dimension.

myMatrix()

Destroy an instance of myMatrix.

void reDim(int, int)

Change the size of an instance of myMatrix. All data contained in the existing instance is destroyed.

void print()

Print the data stored in the matrix to the standard output stream.

void tell()

Print out information about the dimensions of the matrix to the standard output stream.

void set(int, int, double)


myMatrix & operator=(const myMatrix & src)

A=B, copy the matrix B to the matrix A.

double operator()(int, int)

C style indexing deference operator.

myMatrix operator *(myMatrix & B)

Compute A*B.
myMatrix operator *(double mply)

    Compute A*x.

myMatrix operator -(myMatrix & B)

    Compute A-B.

myMatrix operator +(myMatrix & B)

    Compute A+B.

myMatrix transpose()

    Compute A transpose.

myMatrix extractSub(int rn, int rx, int cn, int cx)

    Extract a sub-matrix from A. This operator employs FORTRAN style indexing.

void insertSub(myMatrix & A, int rn, int rx, int cn, int cx)

    Insert a sub-matrix into A. This operatory employs FORTRAN style indexing.

double norm()

    Compute the L2 norm of the matrix.

myMatrix orth()

    Returns the orthonormal basis for the range of A.

void qr(myMatrix & Q, myMatrix & R)

    The qr function performs the orthogonal-triangular decomposition of a matrix.

int rank(double tol)
Compute an estimate of the number of linearly independent rows or columns of a full matrix.

```cpp
void eig(myMatrix & vectors, myMatrix & values)

Find the eigenvalues and eigenvectors of the matrix.
```

```cpp
void svd(myMatrix & U, myMatrix & D, myMatrix & VT)

Perform a single value decomposition of the matrix.
```

```cpp
void ssvd(myMatrix & A, myMatrix & S, myMatrix & R)

Special skinny matrix SVD
```

```cpp
double snorm()

Compute the norm of the matrix.
```

```cpp
void sqr(myMatrix & q, myMatrix & r, vector<int> & p)

Special skinny matrix QR decomposition.
```

A.7 C++ Example

Listing 14 is an example program using CIRBLEIGS to locate the minimum eigenvalues of a matrix. These results are compared to the values found by LAPACK.

```cpp
#include <iostream>
#include <iomanip>
#include "matrix.h"
#include "util.h"
#include <string>
#include <vector>
#include "cirbleigs.h"
#include <cmath>
```
#include <time.h>

extern "C" double dlamch_(char *);

using namespace std;

bool RandomStart = false;
myMatrix A;

void setUpA(myMatrix & A, int n, int width) {
    A.reDim(n,n);
    double tmp ;
    int i,j;

    for(i=0; i<n; i++) {
        for(j=0; j<n; j++) {
            A.set(i,j,0);
        }
    }

    // set the diagonal
    for(i=0; i<n; i++) {
        if (RandomStart) {
            tmp = gasdev();
        } else {
            tmp = -2;
        }
        A.set(i,i,tmp);
    }

    // set the band about the diagonal
    // right now, just one above and below

    for(j=1; j<= width; j++) {
        for(i=j; i<n; i++) {
            if (RandomStart) {
                tmp = gasdev();
            } else {
                tmp = 1;
            }
            A.set(i,i-j, tmp);
            A.set(i-j,i,tmp);
        }
    }
}
myMatrix mply(myMatrix & b) {
    myMatrix rv;
    rv = A * b;
    return (rv);
}

int main(int argc, char * argv[]) {
    int n = 100;
    int width = 1;
    int i, j;
    int K = 10;

    srand48(time(NULL));

    for (i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-n") == 0) {
            i++;
            n = atoi(argv[i]);
        } else if (strcmp(argv[i], "-b") == 0) {
            i++;
            width = atoi(argv[i]);
        } else if (strcmp(argv[i], "-c") == 0) {
            i++;
            K = atoi(argv[i]);
        } else if (strcmp(argv[i], "-R") == 0) {
            RandomStart = true;
        } else {
            cout << "Usage " << endl;
            cout << "  -n #, size of the array " << endl;
            cout << "  -c #, number of eigenvalues to find " << endl;
            cout << "  -b #, size of the block " << endl;
            cout << "  -R, random start " << endl;
        }
    }
}
setUpA(A, n, width);

int cholM = 0;
int dispr = 0;

string endpt = "MON";
int maxdpol = 200;
int maxit = 10000;
int nbls = 3;
int blsz = 3;
vector<double> nval;
string zertyp = "ML";
int sizint = 1;
double tol = 1e-6;
myMatrix permM;

if (blsz * nbls >= n) {
    nbls = n/blsz;
}

myMatrix V(n, bsz);
for (i=0;i<n;i++){
    for (j=0;j<blsz;j++) {
        V.set(i,j, gasdev());
    }
}

if (blsz * nbls -K - blsz - sizint < 0) {
    nbls = int(ceil(double(K+sizint+blsz)/double(blsz)+0.1));
    cout << "Warning Increasing nbls to " << nbls << endl;
}

if (blsz*nbls > n) {
    cout << "n is too small, it is " << n << " and should be at least " << blsz*nbls+1 << endl;
}

char var = 'e';
double MTOL;
MTOL = dlamch_(&var);
if (tol < MTOL) {
    tol = MTOL;
}

string computvec = "F";
myMatrix ninpeigvec(1,1);
ninpeigvec.set(0,0,0);
vector<double> eigval;

int iter = 0;
myMatrix eigvec;
string funpar;
string sigma = "LE";
int Tsz = 0;
vector<double> ritz;
myMatrix ritzvec;

cirbleigs(mply, maxit, nbls, ninpeigvec, eigval, eigvec, blsz, K, V, permM, funpar, n, tol, computvec, 
dispr, nval, sigma, zertyp, sizint,M, endpt, maxdpol, 
iter, Tsz, 
ritz, ritzvec);

myMatrix Val, Vec;

cout << "Calling the lapack routine to find eigen values " << endl;
A.eig(Vec,Val);

sort(eigval.begin(), eigval.end());

cout << endl;
cout << "
                        ******************************************************" << endl;
cout << "\tMatrix size " << n << endl;
if (RandomStart) {
    cout << "\tThe matrix is random" << endl;
} else {

cout << "The matrix has typical row . . . 1 −2 1 . . . " << endl;
}
cout << "The number of eigenvalues to find is " << K << endl;
cout << "Total Iterations: " << iter << endl;
cout << endl << endl;

int tmp;
tmp = Val.rows−1;
cout << scientific;
cout << " Lapack Value Cirbleigs Value Difference " << endl;
for (i=0;i<eigval.size();i++) {
    cout << setw(15) << Val(tmp−i,0) << " − " << setw(15)
         << eigval[eigval.size()−i−1] << " = " << setw(15)
         << Val(tmp−i,0) −eigval[eigval.size()−i−1] << endl;
}
cout << endl;

return 0;
}

Listing 14: A C++ CIRBLEIGS Example

A.8 Parallel Skinny Matrix Class

The parallel skinny matrix class, pmatrix, is used in CIRBLEIGS. A typical matrix
is $n \times m$, where $n >> m$. This matrix distributes data across the processes by assuming
that a process contains at least $m$ rows. With the exception of process 0, all processes
can access the $m$ rows above by communicating with the previous process. The $m$ rows
below a process, with the exception of the last process, are available from the following
process.

The class has a large number of methods and only those required for interfacing with
PCIRBLEIGS will be presented here. As with the myMatrix class, some of the member
functions use FORTRAN style indexing while others employ C++ style. The class does
not employ data hiding to allow for optimized access to data.

**pmatrix(int lrows,int lcols, char)**

Create an instance of a matrix, with lrows local rows and lcols local columns.

The character is for debugging purposes and will be removed in future releases.

**pmatrix(int lcols, pmatrix & A)**

Create an instance of a matrix with lcols local columns, local rows will be based on the local rows of the matrix A.

**pmatrix (const pmatrix & src )**

Create an instance of a matrix which is a copy of the matrix src.

**void print()**

Print the matrix.

**void tell()**

Print details about the matrix.

**void zero()**

Initialize the matrix to zero.

**void urandomFill()**

Fill the matrix using the uniform distribution, range 0 to 1.

**void grandomFill()**

Fill the matrix using the normal distribution, u=0, s=1.
void patternFill()

    Fill the matrix with a pattern.

pmatrix extractSubP(int, int )

    Extract a set of columns from the matrix. This uses FORTRAN style indexing for
the columns to extract.

pmatrix insertSub(pmatrix src, int, int)

    Insert the columns from src into the matrix. This uses FORTRAN style indexing
for the columns to insert.

double index(int row,int col)

    Return the value of an element of the matrix. This requires a collective communi-
cation so that all processes contain the same value. This uses C style indexing.

void set(int row, int col, double value)

    Set a value of the matrix.

A.9 Parallel C++ Example

    Example 15 is taken from the PARPACK example set. The matrix used is the two
dimensional discrete Laplacian on unit square with zero Dirichlet boundary condition.

    In this example, the matrix is not explicitly stored.

    Matrix size is read from a file to permit batch testing.

#include <mpi.h>
#include <iostream>
#include <iomanip>
#include <fstream>
#include <string>
```cpp
#include <vector>
#include <cmath>
#include <time.h>
#include <cstdlib>
#include "pmatrix.h"
#include "matrix.h"
#include "util.h"
#include "peirbleigs.h"

using namespace std;

extern "C" double dlamch_(char *);

int MatrixMultiply;

struct junk{
    double value;
    int pos;
    bool operator<(const junk that) const {
        return (value < that.value);
    }
};

int nx;
int n;

double Middle = 100;
double Side = -25;

pmatrix doMult(pmatrix & a) {

    int size, rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    double * above;
    double * below;

    above = new double[nx];
    below = new double[nx];
```
for (int i = 0; i < nx; i++) {
    above[i] = 0;
    below[i] = 0;
}

int tag = 11;
MPI_Status status;

if (rank != size-1) {
    MPI_Send(&(a.array[a.lrows-nx]), nx, MPI_DOUBLE, rank+1, tag, a.comm_group);
}
if (rank > 0) {
    MPI_Recv(above, nx, MPI_DOUBLE, rank-1, tag, a.comm_group, &status);
}
if (rank != 0) {
    MPI_Send(&(a.array[0]), nx, MPI_DOUBLE, rank-1, tag, a.comm_group);
}
if (rank != size-1) {
    MPI_Recv(below, nx, MPI_DOUBLE, rank+1, tag, a.comm_group, &status);
}

int i, j;

pmatrix b(1, a);
b.zero();

double tmp;

for (i=0; i<a.lrows; i++) {
    tmp = 0;
    if (i < nx) {
        tmp = Side * above[i];
    } else {
        tmp += Side * a.array[i-nx];
    }
if ((i + a.srows) % nx != 0) {
    if (i == 0) {
        tmp += Side * above[nx-1];
    } else {
        tmp += Side * a.array[i-1];
    }
}

else {
    tmp += Middle * a.array[i];
}

if ((i + a.srows) % nx != (nx - 1)) {
    if (i == a.lrows - 1) {
        tmp += Side * below[0];
    } else {
        tmp += Side * a.array[i+1];
    }
}

else {
    tmp += Side * below[(i - a.lrows + nx) % nx];
}

else {
    tmp += Side * a.array[i + nx];
}

b.array[i] = tmp;

delete[] above;
delete[] below;
return b;
}

pmatrix mply(pmatrix & b){

    int size, rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    MatrixMultiply ++;
    pmatrix rv(b.gcols, b);

rv.zero();

pmatrix a, c;

int i;

for (i=1; i<= b.gcols; i++) {
    a = b.extractSubP(i, i);
    c = doMult(a);
    rv = rv.insertSub(c, i, i);
}

return rv;

int main(int argc, char *argv[]) {

    int rank;
    int size;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    ifstream infile;

    infile.open("RUNSIZE");
    infile >> nx;
    infile.close();
    n = nx*nx;

    if (nx < size) {
        if (rank == 0) {
            cout << "You must change nx or use less than " << nx
                 << " processors " << endl;
        }
        MPI_Finalize();
        exit(-1);
    }

    long dtmp;
    int i, j;
    int K = 10;
if (rank == 0) {
  srand48(time(NULL));
  for (i=0; i<size; i++) {
    dtmp = lrand48()+1;
    MPI_Send(&dtmp, 1, MPI_INT, i, 1, MPI_COMM_WORLD);
  }
} else {
  MPI_Recv(&dtmp, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
  srand48(dtmp);
}

double start, end;

start = MPI_Wtime();
myMatrix M;
int cholM = 0;
int disp = 0;

string endpt = "MONT";
int maxdpol = 600;
int maxit = 200000;
int nbls = 5;
int bksz = 5;
vector<double> nval;
string zertyp = "ML";
int sizint = 1;
double tol = 1e-8;
myMatrix permM;

if (bksz * nbls >= n) {
  nbls = n/bksz;
}

int localRows;
if (rank < (n%size)) {
  localRows = int(ceil((float)n/(float)size));
} else {
  localRows = int(floor((float)n/(float)size));
}
pmatrix V(localRows, blsz, 'a');
V.grandomFill();

if (blsz * nbls - K - blsz - sizint < 0) {
    nbls = int(ceil(double(K+sizint+blsz)/double(blsz)+0.1));
    if (rank == 0) {
        cout << "Warning Increasing nbls to " << nbls << endl;
    }
}

if (blsz*nbls > n) {
    if (rank == 0) {
        cout << "n is too small, it is " << n << " and should be at least " << blsz*nbls+1 << endl;
    }
}

char var = 'e';
double MTOL = dlamch_(&var);
double MTOL = 0;
if (tol < MTOL) {
    tol = MTOL;
}

string computvec = "F";
myMatrix ninpeigvec(1,1);
ninpeigvec.set(0,0,0);
vector<double> eival;

int iter = 0;
pmatrix eigvec;
string funpar;
string sigma = "SE";
int Tsz = 0;
vector<double> ritz;
myMatrix ritzvec;

// count total multiplies
MatrixMultiply = 0;
cirbleigs(mply, maxit, nbls, ninpeigvec, eival, eigvec, blsz, K, V, permM, funpar, n, tol, computvec,
MPI_Barrier(MPI_COMM_WORLD);
end = MPI_Wtime();

if (rank == 0) {
    cout << endl;
    cout << "Total Time " << (float)(end - start) << endl;
    cout << endl << endl;

    vector<junk> jj;
    junk jtmp;
    for (i = 0; i < eigval.size(); i++) {
        jtmp.pos = i;
        jtmp.value = eigval[i];
        jj.push_back(jtmp);
    }

    sort(jj.begin(), jj.end());
    int tmp;
    cout << scientific;
    cout << "Ritz values and direct residuals" << endl;
    cout << "---------------------------" << endl;
    cout << "Col 1 Col 2" << endl;
    for (i = 0; i < jj.size(); i++) {
        cout << "Row" << setw(4) << i << " : ";
        cout << setw(15) << setprecision(13) << jj[i].value << "\t";
        printError(jj[i].pos);
        cout << endl;
    }
    cout << endl;
    cout << "PCIRBLEIGS" << endl;
    cout << "= = = = = = = = = =" << endl;
    cout << "Size of the matrix is " << n << endl;
    cout << "Number of the processors is " << size << endl;
    cout << "Number of Ritz values requested is " << K << endl;
}
cout << " The number of Arnoldi vectors generated is " << 0 << endl;
cout << " What portion of the spectrum: " << sigma << endl;
cout << " The number of converged Ritz values is " << jj.size() << endl;
cout << " The number of Implicit Arnoldi update iterations taken is " << iter << endl;
cout << " The number of OP*x is " << MatrixMultiply << endl;
cout << " The convergence criterion is " << tol << endl;
cout << endl;

MPI_Finalize();

Listing 15: A C++/MPI PCIRBLEIGS Example
APPENDIX B

Pathview User Manual

Pathview is a special purpose CUMULVS field viewer designed for use with mpipath. This program is designed to graphically display data generated by mpipath on workstations and visualization walls. This software incorporates a number of alternative input devices which allow the user to control the program.

This document is derived from the Pathview documentation posted on the CUMULVS wiki located at http://mirkwood.cs.edinboro.edu/∼bennett/research/pmwiki/.

B.1 Status

This software is special purpose, and is most likely only usable with mpipath. The author is happy to share this software, upon request. Please contact Dan Bennett, danbennett360@gmail.com for access.

B.2 Pathview Command Line Arguments

Pathview uses the following command line arguments

-nd

No display. This is a debugging option. If this flag is present, the display routine is replaced with a no-op.

-stv
Attempt to contact pathprog via CUMULVS. This flag is off by default. This flag is invalid if the program is not compiled with the STV libraries.

-file filename startNumber endNumber

Take input from a named file. Pathview assumes that the data files have been dumped by pathprog and are in sequential order numbered from startNumber to endNumber. There is no provision for an extension. The user is warned that when running under CHROMIUM the entire path name must be specified.

This command line argument is only valid if all three parts are given.

The command

```
pathview -file outs/out 0 20
```

Will attempt to read out0 through out20 in the directory outs

No validation is performed.

-noWii

Do not attempt to connect to the wiimote. This is a debugging option to speed the testing cycle.

-BigFont

Use the large font set. This option enables the larger font for the GUI system required by the wall. By default this option is disabled.

-LineWidth n
Change the default line width used to draw the HUD and the annotation elements.

By default the line width is 1. An appropriate line width for the wall appears to be 3.

B.3 Space Navigator in Pathview

The space navigator is a low cost three dimensional input device which provides 6 degrees of freedom. This device provides data which can be used to produce both rotation and translation along all three axis. Equipped with two buttons and a three dimensional joy stick, this device provides a natural way to manipulate the orientation of the data displayed.

In pathview the space navigator is used to perform rotations and translations of the data set. By default rotation is enabled, however this can be toggled on and off via the right button. Likewise, translation, which is disabled by default, can be enabled via the left button.

In general, the only difficulty encountered when using the space navigator is physical orientation of the device. The device should be positioned with the logo pointing towards the user, as pictured, and the cable extending out of the back.

B.4 Wiimote

The wiimote is a three dimensional input device which supplies a variety of orientation information. This device is equipped with an accelerometer, which can provide information about rotation and vertical angle, as well as an infra-red camera which can provide horizontal angle information. The device is further equipped with 11 buttons.
Additional features include a set of 4 led lights, a speaker and a vibration device.

The device can be enhanced with a number of extensions including a nunchuk. The nunchuk provides an additional accelerometer, a two dimensional joystick and two additional buttons.

To use a wiimote with pathview, the user must attach the device during the initialization phase of the program. At startup, user will be prompted with the following message:

************************************************
Attempting to connect to the wiimote
If you wish to use the wiimote,
you should press both 1 and 2 on the wiimote now
************************************************

wiiuse v0.12 loaded.
   By: Michael Laforest <thepara[at]gmail{dot}com>

The program will then spend 5 seconds searching for any available wiimotes. The wiimote must be placed in discovery mode by simultaneously pressing buttons 1 and 2. If the wiimote is successfully placed in discovery mode, all four leds will flash until a connection is made or the discovery period on the wiimote times out.

Upon successful connection the wiimote will rumble (vibrate) for approximately half a second, and the first led light will be enabled. In addition, the following message will appear.

[INFO] Found 1 bluetooth device(s).
[INFO] Found wiimote (00:1E:35:7E:C8:D2) [id 1].
[INFO] Connected to wiimote [id 1].
Connected to 1 wiimotes (of 1 found).

Within pathview, the wiimote serves a number of purposes.
B.4.1 Cursor Control

The wiimote accelerometer, rocker pad and B button are used to control the position of the cursor. The cursor is the box used to highlight individual elements of the liquid crystal structure. This box can be placed on any element within the data or on the boundary. Movement is accomplished via the following controls:

- The left and right rocker buttons move the cursor to the left and right.
- The up and down rocker buttons move the cursor forward and backwards (into the screen).
- Vertical movement is accomplished by pointing the wiimote up or down and pressing the B button.
- The 2 button toggles cursor visibility.

B.4.2 Display Control

The wiimote A button, along with the + (plus) and - (minus) buttons are used to control a number of display aspects. Currently the A button is used to cycle between color, shape, scale, and advance options, while the plus and minus buttons are used to cycle forward and backward through different values of each option. The current option is displayed in the HUD on the wiimote line.

B.4.3 Clip Box Control

The wiimote can be placed into clip box control mode. In this mode, the wiimote can be employed to adjust the clipbox. This mode is entered and exited by pressing the
1 button. When in this mode, the wiimote line on the HUD will display Wiimote: Clip Box, and the external clipbox will be displayed. Control is only available for the external clip box, and is accomplished as follows

- The +/- buttons are used to cycle through the clip box faces.
- The left/right buttons on the rocker pad are used to move a face in and out. If the selected face is aligned with the y-z plane, the movement corresponds to rocker pad movement, ie left moves the face to the left and right moves the face to the right.
- The up/down buttons on the rocker pad are used to move the clip box. If the selected face is aligned with the x-z plane, the movement corresponds to the rocker pad movement, ie up moves the box up and down moves the box down.

B.4.4 Nunchuk

The nunchuk is used to control the on screen pointer. This is a two dimensional selection object used to select items in the GUI. In many respects, the nunchuk behaves much like a mouse. The joystick is used to move the pointer about the screen. The Z button is used in place of the left mouse button to select GUI elements. The C button serves as the right mouse button, which is only currently used to move pop-up menus, such as color bin selection. In general, pointer control and movement is highly susceptible to lag when a large number of simulation elements are displayed.

B.5 GUI Menus

The GUI menu system provides control over a number of features in pathview. Currently these features include the Color Scheme, Shape selection, and the Hidden Element
Scheme. Both color and shape are controlled through drop down menus, while hidden elements are controlled through pop-up menus containing button selections.

All elements are selected through a left mouse button push, which is accomplished either through the mouse, or by using the wiimote nunchuk button Z.

The right mouse button, which can be accessed through the wiimote nunchuk button C is used to move popup menus.

B.6 Keypress Menus

Keypress menus provide access to all pathview options and controls. It is possible to completely operate pathview through the keypress menu system without any additional peripheral devices.

The menu system is a two layer heirarcy, with the Main menu granting access to Cursor, Clip Box, Hidden Element, Stride, and Transform sub menus. In addition, a number of other controls are contained within the Main menu.

B.6.1 Main Menu

• Sub Menu Selection

  – c selects the cursor menu

  – C selects the clip box menu

  – D selects the stride menu

  – h selects the hidden element menu

  – T selects the transform menu
• Frame control

- f steps forward one frame, in file mode this reads the next file, in stv mode this informs cumulvs that the application is ready to receive the next frame.

- F steps backward one frame. In file mode, this reads the previous frame. This options is not available in stv mode.

- g toggle run forward mode. This keypress toggles between running forward and paused. Pathview will read and display the next frame as soon as it becomes available. In file mode, this reads the next file. In stv mode the next frame is displayed as soon is at is delivered and CUMULVS is directed to generate and deliver a new frame as soon as one is available. This action turns off run backward mode.

- G toggle run backward mode. This keypress toggles between running backwards through a file and paused. When enabled, the application will proceed backwards through the data file set until the beginning file is reached. This option is not available in stv mode.

• Other Scene Control

- +/- increase/decrease scale factor by 10

- b/B cycle forward/reverse through visible data scheme. These schemes include: Show only the volume data, show only the boundary data, show both volume and boundary data, show lower half of boundary data, show upper half of boundary data.
– e toggle display of the extracted element.

– k/K cycle forward/backward through color schemes.

– l/L increase/decrease annotation element line width. This is useful when using the wall.

– m Cycle through data display modes. This is highly experimental. The goal is to display how the edges of the data are different from the boundary data.

– s/S cycle forward/backward through shape routines.

– t toggle scale style. There are currently only two options. Scaled by eigenvalue is the intuitive scaling where the size of the element is determined by the size of the eigenvalues. This is the default option. No scaling displays elements without scaling applied. This option is useful when scene elements have effectively become zero, yet one wishes to observe color properties of the item.

• Program Control

– q quit the program.

– r reset most parameters.

– v toggle menu verbosity.

B.6.2 Cursor Menu

The cursor menu provides basic control over the cursor. Through this menu the cursor can be moved about within the data. Other options include hiding and resetting the cursor, and printing information about the item currently selected by the cursor.
- Movement

  - l move the cursor to the left
  - r move the cursor to the right
  - u move the cursor up
  - d move the cursor down
  - b move the cursor back (into the screen)
  - f move the cursor forward (out of the screen)

- Other Options

  - p print information about the item currently highlighted by the cursor.
  - r reset the cursor.
  - q quit the cursor menu and return to the main menu

B.6.3 Clip Box menu

Pathview contains two clip boxes. An inside clip box and an outside clip box. Control of these clip boxes is provided through this menu. Each clip box contains six faces, labeled f,F, g,G, e and E. These faces can be moved in or out of the data to eliminate the display of unwanted data elements. In addition, the clip box can be moved throughout the data as a single unit.

- a toggle the active clip box. The outside clip box is active by default.
- d move the current wall out. This motion is away from the center of the data field.
• D move the current wall in. This motion is towards the center of the data field.

• e to g, E to G select one of the clip planes to be active.

• r reset all clip box elements

• p/P move the active clip box.

• s/S change the clip box visibility scheme. This cycles through none, outside, inside and both.

• q quit and return to the main menu.
APPENDIX C

CUMULVS Toolkit

The CUMULVS Toolkit (CTK/pyCTK) is intended to assist users in creating CUMULVS viewers. This is accomplished by simplifying the CUMULVS viewer API by providing high level routines to accomplish commonly performed tasks. This library is intended for viewers which are intended to attach to a single simulation.

This library was created and is maintained by Dan Bennett, and is available at http://mirkwood.cs.edinboro.edu/software/CTK. Please address correspondences to dan-bennett360@gmail.com.

C.1 Overview

The CUMULS middle-ware software provides a mechanism for adding steering and visualization to a parallel computation. To effectively use this middle-ware a user must create one or more viewers which receive the data from the parallel computation and allow the user to interact with this data. Two broad categories of viewers include field viewers and parameter viewers.

Field viewers allow the user to interact with the large distributed data structure associated with most parallel computations. This is data which has been decomposed across a number of processes. These data sets are likely to be large and the actions performed by special purpose viewers are likely to be highly specialized and related to the problem domain.
Parameters in CUMULVS are classified as scalar parameters; a single variable available on all processes in a computation, or as vector parameters; a collection of scalar parameters, or possibly an array. In either case, all processes have a copy of the variable associated with the parameter, and all instances of the variable contain the same value. An example of a scalar parameter is a loop control variable, while an array containing the coefficients of a polynomial, common to all processes, would form a vector parameter. Parameter viewers allow the user to interact with parameters in a running computation. Each update, CUMULVS extracts parameter values and transfers this data to any attached viewers. Viewers can steer computations by transferring new parameter values to the simulation via CUMULVS. These viewers are more likely to be constructed from generic components in configurations which are specialized to the application.

For both types of viewers, there are a set of common, high level tasks which must be accomplished. These tasks include connecting to the parallel computation, requesting information concerning data transferred, and controlling the flow of execution.

The CUMULVS Toolkit is an attempt to simplify these common tasks by collecting a number of low level routines from the CUMULVS API into high level tasks. To accomplish this, reasonable values are assumed for common parameters, and it is assumed that the user is connecting to a single parallel computation.

This library is written in C++ and relies on the standard template library, especially the standard container classes.

C.2 Copyright

This document is part of CTK/pyCUMULVS.
CTK/pyCUMULVS is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

CTK/pyCUMULVS is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with CTK/pyCUMULVS. If not, see <http://www.gnu.org/licenses/>.

C.3 Requirements

Both CTK and pyCUMULVS require the CUMULVS package. This package can be obtained from http://www.csm.ornl.gov/cs/cumulvs.html.

C.4 Software Status

This software is currently alpha release quality. It was developed to assist in the creation of a number of viewers for the author’s Ph. D. project and tasks tend to be focused on those needed to accomplish this work. Future release versions will clean up the user interface, focus on additional functionality to be added to the library and provide additional example routines and documentation. The package will be further enhanced by employing object oriented techniques. Error checking will be enhanced in future versions.

C.5 C++ Interface

The following documents the CUMULVS Toolkit C++ interface. This interface is aimed at the development of field viewers. Support for parameters is limited to viewing. Future expansion will include routines which are used to support the pyToolkit described
int InitSTV (std::string appname ,bool bolck );

Connect to the application appname. This routine will also reset a connection if requested for a currently connected application. If block is true, the routine will be blocking and not return until a successful connection is made, if block is false, the routine will time out and return regardless of the connection status. The return value will be negative if the routine was unsuccessful.

std::vector<std::string> ParamNames();

Return a list of the parameters available for steering. This is a vector of strings. If the operation fails an error message is printed and an empty vector is returned. Scalar parameters are returned in the form ParamName. Vector parameters are returned in the form MainParamName::SubName.

std::vector<std::string> FieldNames();

Returns a list of regularly decomposed fields available from the application.

std::vector<std::string> ParticleNames();

Returns a list of irregularly decomposed fields (or particles) available from the application.

int ViewFieldConnect(string fieldName)

Attempt to form an initial connection to a given field. This connection is needed to acquire information about the named field which is necessary to form a final
connection to the field. This routine is called by `GetFieldUB` and `GetFieldLB` if necessary.

```cpp
std::vector<int> GetFieldUB(std::string);
```

Return the upper bounds of a named field. This is a vector, with a value for each dimension of the field.

```cpp
std::vector<int> GetFieldLB(std::string);
```

Return the lower bounds of a named field. This is a vector, with a value for each dimension of the field.

```cpp
void stvConnectToFields(std::vector<int> glb,
                        std::vector<int> gub,
                        std::vector<int> cellSize);
```

Connect to the fields. This routine is required to form a final connection to a field and begin receiving field data. This routine takes a list of field upper and lower bounds, and cell sizes.

```cpp
int stvNextFrame();
```

Attempt to receive the next frame from CUMULVS. If successful, the routine returns a 1, a zero is returned if the frame is not yet ready, a negative number is returned upon error.

```cpp
std::vector<int> stvFieldDims(std::string);
```

Return the dimensions of a named field.

```cpp
void * stvDataField(std::string);
```
Return a pointer to a data field. This routine is used to retrieve field data.

```cpp
void stvContinue();
```

Allow the computation to receive the next frame when it has been produced. Calling this routine indicates that all data from the current frame has been extracted and data values can be overwritten by the next frame. This call only needs to be made after the viewer has connected to a field.

```cpp
void stvHalt();
```

Disconnect from the application

```cpp
int stvGetIntParam(std::string);
```

Return the value associated with the named parameter as an integer. This routine is invalid for vector parameters.

```cpp
std::string ParamValue(std::string);
```

Return the named parameter value rendered as a string.

```cpp
double GetDoubleValue(std::string);
```

Return the named parameter value rendered as a double.

### C.6 C++ Example

Listing 16 contains example code using the CUMULVS toolkit to view parameters. This example is written in C++ and designed to work with the LIFE application.

```cpp
#include <iostream>
#include <cmath>
#include <string>
```
```cpp
#include <vector>
#include <iomanip>

// must be defined
#define STV_MP_PVM
#include <CTK.h>

using namespace std;

void TellAboutThing(string Name, vector<string> Thing) {
    int i;

    cout << endl << endl;
    if (Thing.size() > 0) {
        cout << Name << endl;
        for (i = 0; i < Thing.size(); i++) {
            cout << setw(5) << i+1 << setw(50) << Thing[i] << endl;
        }
    } else {
        cout << "No " << Name << " Found" << endl;
    }
}

void TellAboutBounds(string title, vector<int> bounds, string fieldName) {
    int i;

    cout << endl;
    cout << "The field " << fieldName << ":" << endl;
    cout << "\t" << title << " ";
    for (i = 0; i < bounds.size(); i++) {
        cout << bounds[i] ;
        if (i < bounds.size() - 1) {
            cout << " x ";
        }
    }
}

int main () {
    // a place to store the names of everything.
    vector<string> Params;
    vector<string> Fields;
}
vector<string> Parts;

int rc, i;

// connect to the simulation
rc = stvInit(“LIFE”, false);

if (rc < 0) {
    cout << “Unable to connect to LIFE” << endl;
    return -1;
}

cout << “Connected to LIFE” << endl;

// Gather the names of all of the data instrumented
Params = stvParamNames();
TellAboutThing(“Paramters”, Params);
Fields = stvFieldName();
TellAboutThing(“Fields”, Fields);
Parts = stvParticleNames();
TellAboutThing(“Particles”, Parts);

int frameTime=0;

cout << endl << endl;
cout << ”Blocking Test” << endl;
cout << ”Watching parameter ” << Params[10] << ” for 3 frames” << endl;
cout << ” Frame Number Parameter Value” << endl;

for (i=0; i < 3; i++) {
    frameTime = stvNextFrame(true);
    if (frameTime > 0) {
        cout << ”\t” << frameTime << ”\t” << stvParamValue( Params[10]) << endl;
    }
}

// test of non-blocking next frame
cout << endl << endl;
int oldFrameTime, count = 0;
cout << ”Non-blocking test” << endl;

cout << "Watching parameter " << Params[10] << " for 3 frames" << endl;
cout << " Frame Number Parameter Value" << endl;

oldFrameTime = frameTime;
i = 0;

while (i < 3) {
    frameTime = stvNextFrame(false);
    if (frameTime > oldFrameTime) {
        cout << "\t" << frameTime << "\t" << stvParamValue(Params[10]) << " ";
        cout << count << " checks between frames were performed" << endl;
        oldFrameTime = frameTime;
        count = 0;
        i++;
    } else {
        count ++;
        usleep (5000);
    }
}

stvHalt();
}

Listing 16: A C++ CTK Example for Parameters

Listing 17 contains example code to receive field data using the CUMULVS toolkit.

This example is written in C++ and is designed to work with the LIFE application.

/*
This is a simple field extraction example using CTK, the CUMULVS Tool Kit.
*/
#include <iostream>
#include <cmath>

#include <string>
#include <vector>
#include <iomanip>
/ must be defined
#define STV_MP_PVM
#include <CTK.h>

using namespace std;

void PrintSquare(int * field, int size) {
    int j, k;

    for(j=0; j<5; j++) {
        for(k=0; k<5; k++) {
            cout << field[j*size+k] << " ";
        }
        cout << endl;
    }
    cout << endl;
    return;
}

void TellAboutBounds(string title, vector<int> bounds, string fieldName){
    int i;

    cout << endl;
    cout << "The field " << fieldName << ":" << endl;
    cout << "\t" << title << " ";
    for(i=0; i<bounds.size(); i++) {
        cout << bounds[i] ;
        if (i < bounds.size()-1) {
            cout << " x ";
        }
    }
}

int main () {
    vector<string> Fields;

    vector<int> cellSizes;
    vector<int> UpperBounds, LowerBounds;
    vector<int> CellSize;
```cpp
int * fieldData;
int frameTime=0;
int rc ,i ;

// connect to the simulation
rc = stvInit("LIFE",true);

if (rc < 0) {
    cout << "Unable to connect to LIFE" << endl;
    return -1;
}

cout << "Connected to LIFE" << endl;

// Gather the names of all of the data instrumented
Fields = stvFieldNames();

// gather some information about the first field.
stvViewFieldConnect(Fields[1]);
UpperBounds = stvGetFieldUB(Fields[1]);
LowerBounds = stvGetFieldLB(Fields[1]);

TellAboutBounds("Lower Bounds", LowerBounds, Fields[1]);
TellAboutBounds("Upper Bounds", UpperBounds, Fields[1]);
cout << endl << endl;

for (i=0;i<LowerBounds.size();i++) {
    CellSize.push_back(1);
}

stvConnectToFields(LowerBounds, UpperBounds, CellSize);

int count = 0;
cout << "Non Blocking Test" << endl;
for(i=0;i<5;){
    frameTime = stvNextFrame(false);
    if (frameTime > 0){
        cout << "Frame time is " << frameTime << endl;
        fieldData = (int *)(stvDataField(Fields[1]));
        PrintSquare(fieldData, UpperBounds[0]);
        i++;
        stvContinue();
    }
    count++;
}
```
cout << "There were " << count << " tests " << endl;
count = 0;
} else {
count++;
}
}

cout << "Non Blocking Test" << endl;
for (i=0;i<5;i++) {
    frameTime = stvNextFrame(true);
    cout << "Frame time is " << frameTime << endl;
    fieldData = (int*) (stvDataField(Fields[1]));
    PrintSquare(fieldData, UpperBounds[0]);
    stvContinue();
}

return 0;
stvHalt();
}

Listing 17: A C++ CTK Example for Fields

C.7 Python Interface, pyCUMULVS

The python interface to the CTK provides access to CUMULVS for programs written in python. This interface has the advantages of python’s dynamic typing and standard containers to produce a more flexible programming environment. The current version of pyCUMULVS is aimed at producing programs to view and steer parameters. Future versions will include field handling capabilities. pyCUMULVS is designed to attach to a single application.

pySTVInit

Attempt to connect to an application. This routine, when successful returns a list of three lists. The first list contains the names of all parameters. Simple parameter
names are strings, vector parameter names consist of a list of names, main name followed by individual names. The second list contains a list of field names and the third list contains a list of particle field names.

**pySTVNextFrame**

Get the next frame from CUMULVS. This routine returns the frame number of the current frame.

**pySTVHalt**

Disconnect from a computation.

**pySTVParamValue**

This routine takes a parameter name, or a parameter vector list. It returns a list of the values for the parameter or vector parameter named.

**pySTVParamType**

This routine takes a parameter name, or a parameter vector list. It returns a list of the types for the parameter or vector parameter named.

**pySTVAcquireToken**

Acquire a steering token for a named parameter. The return value is non negative if the call is successful.

**pySTVReleaseToken**

Release a steering token for a named parameter.

**pySTVReleaseAllTokens**
Release all steering tokens which have been acquired.

**pySTVNewParamValue**

Set the value of a parameter to be sent to the application.

**pySTVSendParams**

Set the values of all parameters for which steering tokens have been acquired.

### C.8 Python Examples

The python examples in this section illustrate the use of the python version of the toolkit. Program 18 illustrates connecting to an application, retrieving information for parameters, fields and particles, and printing the value of all parameters until execution stops. The example in Program 19 watches the parameter **Generation Number**. When this value exceeds 10, the steering token for the parameter **Reset Board** is acquired and the value is set to 1, which resets the board.

```python
#! /usr/bin/python
import pyCUMULVS
from numpy import *

[vars, fields, particles] = pyCUMULVS.pySTVInit('LIFE')
print 'Parameters:'
print vars
print
print 'Fields'
print fields
print
print 'Particles'
print particles
print

frameTime = 0
while frameTime != -1:
    newFrameTime = pyCUMULVS.pySTVNextFrame()
```
if newFrameTime != frameTime:
    print 'Iteration Count', newFrameTime
    for aVar in vars:
        tmp = pyCUMULVS.pySTVParamValue(aVar[0])
        tmp2 = pyCUMULVS.pySTVParamType(aVar[0])
        print 'tmp is ', tmp
        print 'tmp2 is ', tmp2
        print '\t', aVar[0],
        if (len(aVar) != 1):
            print ' :'
            i = 1
            for t in tmp:
                try:
                    print '\t', aVar[i], t, tmp2[i-1]
                except IndexError:
                    print t, tmp2[0]
                i = i + 1
        frameTime = newFrameTime
    print '\n'

pyCUMULVS.pySTVHalt()

Listing 18: A Simple Python Example

#!/usr/bin/python

import pyCUMULVS
from numpy import *

[vars, fields, particles] = pyCUMULVS.pySTVInit('LIFE')

paramName = 'Generation Number'
frameTime = 0
while frameTime != -1:
    newFrameTime = pyCUMULVS.pySTVNextFrame()
    if newFrameTime != frameTime:
        frameTime = newFrameTime
        realValue = pyCUMULVS.pySTVParamValue(paramName)[0]

    # the following should reset the board every 10 turns
    if realValue > 10:
        print 'resetting life, Generation Number is ', realValue
        pyCUMULVS.pySTVAcquireToken("Restart Board");
Listing 19: A Python Viewer to Restart an Application
D.1 Overview

pyGUIObjects are an easy to use set of objects which allow users to create a graphical viewer for packages instrumented with CUMULVS [49]. Viewers constructed with these objects provide steering and visualization of program parameters. These objects contain default methods for interacting with a simulation via CUMULVS, as well as methods for providing graphical user interaction.

This package is being developed by Dan Bennett (danbennett360@gmail.com). The status is experimental, however, those interested can obtain a copy of the current version of the software by contacting the author.

D.2 Copyright

This document is part of pyGUIObjects.

pyGUIObjects is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

pyGUIObjects is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with pyGUIObjects. If not, see <http://www.gnu.org/licenses/>.
D.3 Requirements


In addition, the software is currently built upon the pyFLTK interface to the Fast Light Toolkit (FTLK), available at http://pyfttk.sourceforge.net. Future plans include support for additional graphical toolkits.

D.4 Status

The status of this project is experimental. Copies of the software are available from the author by request.

D.5 Background

The process of building a parameter viewer for CUMULVS involves a number of common tasks which are repeated for every viewer. These tasks include forming a connection to the application via CUMULVS, checking for the arrival of new data, receiving the new data, and transmitting changes in parameters to the simulation through CUMULVS. In general, these tasks can be isolated into a single entity, which will be implemented as a connection object.

A connection object is one of the three basic objects in pyGUIObjects and can perform all functions described above. For any application, a single connection object is
instantiated, which is responsible for communications with CUMULVS. Other objects within the viewer must request data from, or send data through this object.

A second class of objects, *client* objects, are provided for user interaction. Client objects register with a connection object and request a list of parameters. As new frames arrive from CUMULVS, the connection object will pass requested parameter values, by way of a function call, to registered client objects. Client objects frequently have a screen presence, which allows for data display and user interaction.

Examples of client objects include objects to display a parameter value as text, to plot the values of a pair of parameter values on a graph, or to allow the user to change a value of a parameter.

A final class of objects are *filter* objects. These objects are placed between connection and client objects and perform functions associated with each. A filter object is responsible for conditional forwarding of parameter data to client objects. As such, filter objects contain the methods of both client and connection objects. A filter object must connect to a connection object, or another filter object. Filter objects are connected to by other filter objects or client objects.

D.6 Data Types

Most data passed between objects is in the form of either a *parameter list* or a *parameter value list*. Parameter lists are python lists containing the names of the parameters for which the object is requesting values. Scalar parameters are passed as a string, while vector parameters are passed as a list of strings, the main parameter name first.

A parameter value list is a list of lists. Each list contains the name of the parameter
followed by the value of the parameter. Vector parameters are passed as a list where the first element is the main parameter name, and each successive element is a list containing the sub-parameter name and the associated value. Figure D.6 displays a parameter list and the corresponding parameter value list.

Figure 80: A Parameter List and a Parameter Value List

\[
\text{["Cell Age", "Generation Number", ["Checkpoint", 
  "First Checkpoint Value", "Second Checkpoint Value"]]
]["Cell Age", 34], ["Generation Number", 123], ["Checkpoint", 
  ["First Checkpoint Value", 0], ["Second Checkpoint Value", 2]]
\]

D.7 Primitive Objects

pyGUIObjects are derived from three base classes. These classes include a \textit{Client-Base} class, a \textit{ConnectionBase} class and a \textit{GUIBase} class. These class support common methods for all classes derived from the base classes. These objects are not intended to be instantiated directly but serve as a common base from which objects are to be built.

D.7.1 ClientBase Class

The \textit{ClientBase} class contains all methods common to clients. Client objects as well as filter objects are derived from this base class.

This class contains two data items:

\textbf{Connection}

The connection object to which this object is connected. Initialized to \texttt{None}.

\textbf{param}
The parameter list this object is to display. Initialized to None. There is some inconsistency in what individual objects store in this variable. Some store a list, while others store a string. This needs to be made consistent before full release.

The methods supported by this class include:

**GetFilterList**

Returns the parameter list the object will use to decide if a data update is required.

This method is normally invoked by a connection object.

**DoParameterUpdate**

Normally called by the connection object, this method will take the parameter value list constructed from the values returned by GetFilterList, and return True if a data update is requested and False if no data update is requested. By default, if param has a value other than None, True is returned, otherwise False is returned.

This method allows an object to perform elementary data filtering and should be overridden when alternative behavior is required.

**GetParameterList**

Returns the parameter list which the object requests for a particular update. This method is normally called by the connection object each update for which DoParameterUpdate has returned True. By default this object returns the list contained in param.

**DoDataUpdate**

Called by the connection object, this routine notifies the client object of new data
from the simulation. Data is passed as a parameter value list. The default behavior for this routine is to return without action. This routine should be overridden in most derived classes.

**SetConnection**

This method takes a single argument, the connection object to which the client will be connected. The method connects to, and registers with the connection object named in the parameter by calling that object’s `RegisterViewer` method. If the client is connected when this routine is called, it deregisters with the current connection by calling the `UnRegisterViewer` method. This must be called once to connect the object to a simulation.

D.7.2 ConnectionBase Class

The ConnectionBase class is the base for all connection objects. Both connection objects and filter objects are derived from this base class.

This class supports two data items:

**clientList**

A list of connected clients. Initialized to the empty list.

**parameterPullList**

A list of clients to poll for parameters before an update. Initialized to the empty list. This is currently unused and available for future development.

The methods supported by this class include:
RegisterViewer

Add a client to the clientList. This method takes a reference to the client to be added.

UnRegisterViewer

Remove a client from the clientList. This method takes a reference to the client to be removed.

GetParameterNames

Return parameter list of all available parameters in the simulation.

DoUpdate

Loops through all of the clients in the clientList. Requests the client filter list, fills this list and requests client update status. If the client wishes to be updated, the parameter list is requested and the client is called with the appropriate parameter value list.

ReceivePush

Receives parameter value list containing new parameter values to be sent to the simulation through CUMULVS. Possible race conditions are handled by last writer wins.

D.7.3 GUIBase Class

The GUIBase class is provided to allow for the deployment of multiple graphical toolkits. This section is most likely to change as the package is adapted to employ other toolkits. This class will eventually provide full support for modifying the properties of
graphical objects, including position, size, visibility. These properties are only partially supported in the current release.

This class currently supports five data items:

xPos

The x position on the screen. Default value is 0.

yPos

The y position on the screen. Default value is 0.

width

The width of the object. Default value is 0.

height

The height of the object. Default value is 0.

outerElement

The outer GUI element for this object. This is a toolkit element specific to the toolkit employed. Default value None.

This class currently supports the following methods:

GetPosition

Return the current x and y position.

SetPosition

Set the current x and y position.
GetSize

Get the current width and height of the object.

SetSize

Set the current width and height of the object.

GUIElement

This method causes the instantiated object to appear (the fltk show method is called for the object). The method also returns the outer GUI element.

activate

Graphically activate the object (allow it to receive events from the GUI). This method will undo the actions of the deactivate method. By default, objects are active when created.

deactivate

Graphically deactivate the object. In most systems the object will no longer receive events, but will still be visible on the screen. Frequently deactivated objects are “grayed-out”.

D.8 Basic Objects

This section describes a set of the basic objects that constitute pyGUIObjects. These are the objects from which most simple viewers will be constructed.
connection = ConnectionTest.ConnectionTest()
client = ParameterViewer.ParameterViewer()

window = Fl_Window(300,60)
window.show()

window.add(client.GUIElement())
client.SetConnection(connection)

Listing 20: A Code Fragment Using the ConnectionTest Object

D.8.1 Connection Objects

Currently two different connection objects exist. The Connection object provides interaction with CUMULVS while the ConnectionTest object simulates interaction with CUMULVS.

ConnectionTest

The connection test object maintains a set of parameters. These include scalar parameters $a$, $b$ and $step$, and a vector parameter $c$ which consists of parameters $d$ and $e$. Each time $DoUpdate$ is called, the value of these parameters is incremented by a constant. Users should add other parameters as needed to match names and performance requirements. This object does not connect to a CUMULVS application and is intended for program testing and requires no arguments at instantiation.

Listing 20 demonstrates the use of the ConnectionTest object along with a ParameterViewer which will be discussed below.
connection = Connection.Connection("LIFE")

client = ParameterViewer.ParameterViewer()

window = Fl.Window(240,60)
window.show()
window.add(client.GUIElement())

client.SetConnection(connection)

while 1:
    Fl.check()
    connection.DoUpdate()

Listing 21: A Code Fragment Using the Connection Object

Connection

The Connection object is required by every program which will communicate with CUMULVS. This object is instantiated with the name of the application to which the application should connect. The DoUpdate method must be called periodically to receive new frames. Clients register with this object by calling its RegisterViewer method. Clients wishing to steer a parameter should call the ReceivePush method with an associated parameter value list. Registered routines will receive data updates through the DoUpdate method described in the base class. This object has no screen presence.

Listing 21 demonstrates the use of a connection object. This program connects to the simulation LIFE and displays a single parameter, selectable by the user at run time. The code in this listing contains code to initialize the window system, as well as code for the main loop. These items will be deleted from future listings. All example code is available in full with the package.
Further Plans for Connection Objects

The connection object set will be extend by adding a *GUIConnection* object. This object will have a screen presence. It will allow the user to enter the application name and perform actions such as connecting, disconnecting and resetting the connection to a simulation. This object will also provide an interface with the capability to pause, multi-step, and continue execution of an application.

D.8.2 Filter Objects

Filter objects are designed to sit between a client and a connection. The filter object receives data from the connection and, based upon some established criteria, decides if the data should be forwarded to any attached clients. Filters forward parameter steering requests to their connection object. There is currently one object in this class, the *StaticFilter*.

*StaticFilter*

The *StaticFilter* object is instantiated with a parameter list and a user supplied boolean function. The object will request the values of the parameters in the parameter list and call the user supplied function with the parameter value list each time a new frame is received. Based upon the results of the user supplied function, all connected clients will be called with an appropriate parameter value list. This object has no screen presence.

Listing 22 demonstrates the use of the *StaticFilter* object. In this case, the object is initialized to watch the paramter *Generation Number* and use the function *myFilterFun*
import StaticFilter
def myFilterFun(pl):
    insideList = pl[0]
    if insideList[1] % 10 == 0:
        return True
    else:
        return False

connection = Connection.Connection("LIFE")
filter = StaticFilter.StaticFilter("Generation Number", myFilterFun)
client = ParameterViewer.ParameterViewer()

Listing 22: A Code Fragment Using the StaticFilter Object

to make forwarding decisions. In this example, data will be forwarded only when
Generation Number is a multiple of 10.

Further Plans for Filter Objects

A DynamicFilter object with a screen presence will be created. This object will
allow the user to select parameter(s) upon which filtering will be performed as well as
graphically build simple rules for filtering.

D.8.3 Client Objects

Client objects are the most diverse class of objects. These objects perform a number of
functions including parameter display and parameter steering. Display object can display
simple text or graphical representation. Some special purpose objects are included in this
class. All objects currently in this class have graphical representations. Static objects
connection = Connection.Connection("LIFE")

clientA = ParameterViewer.ParameterViewer()
clientB = StaticParameterViewer.StaticParameterViewer("Average Age")
clientB.SetPosition(0,60);
clientC = StaticLabeledParameterViewer.StaticLabeledParameterViewer("Cells Alive")
clientC.SetPosition(0,120);

# window code deleted

window.add(clientA.GUIElement())
window.add(clientB.GUIElement())
window.add(clientC.GUIElement())

clientA.SetConnection(connection)
clientB.SetConnection(connection)
clientC.SetConnection(connection)

Listing 23: A Code Fragment Using the ParameterViewer Objects

expect to be given a parameter list when the object is instantiated, non-static objects will present the user with a drop down menu to select an appropriate parameter for display. Static objects do not contain a label unless a StaticLabeled version of the object is created.

Simple Parameter Viewers

The simple parameter viewers consist of the ParameterViewer, the StaticParameterViewer and the StaticLabeledParameterViewer.

Use of the ParameterViewer is demonstrated in Listings 20, 21 and 22. Use of the StaticParameterViewer and the StaticLabeledParameterViewer are demonstrated in Listing 23.
The *StaticVectorViewer* is a compound object for viewing a vector parameter. This viewer is capable of displaying the values of a vector parameters. This display is built, at run time, from *StaticParameterViewers*, which contain the individual elements of the vector parameter. It is likely that this objected needs to be refined and operations expanded in future releases.

**Specialized Parameter Viewers**

Several specialized parameter viewers have been created. These viewers tend to perform very specialized tasks. The *BinaryFlagDisplay* is designed to watch a single integer parameter, and indicate that associated bit fields have been set. For example, a single parameter *State* may be used to track the state of a program. This parameter can indicate what part of the computation is currently in execution as well as the state of execution (paused, waiting for input, ...). The *BinaryFlagDisplay* can be used to display this information.

The *StaticVectorMatrixViewer* is employed to display a two dimensional matrix, which is stored in a vector parameter. This compound object is constructed from *StaticParameterViewer* objects.

**Graphical Parameter Viewers**

*XYScatterPlot* presents a scatter plot of values of two parameters. This object is constructed from objects in the *matplotlib* available at http://matplotlib.sourceforge.net/. The *PZ_XYScatterPlot* object is a compound object constructed from the *XYScatterPlot*. This object has additional controls which allow the user to pan and zoom in the displayed
graph. This class of viewers should be greatly expanded to include more objects from the matplotlib package.

Parameter Steerers

Parameter steerers allow the user to alter the value of a parameter. Types include the \textit{StaticButtonSteerer} which allows the user to send a single value through a parameter, the \textit{ScalarPushSteerer} and \textit{StaticLabeledPushSteerer} which allow the user to enter a value, and the \textit{StaticListParameterSteerer} which presents the user with a list of setting values from which to choose.

The \textit{StaticButtonSteerer} is instantiated with the parameter to steer, and the value to set this parameter to send when the button is pressed. The \textit{ScalarPushSteerer} is instantiated with the name of a parameter to steer. The \textit{StaticLabelParameterSteerer} is a compound object which is built from a \textit{ScalarPushSteerer} and a \textit{StaticParameterViewer}. This object will display the current parameter value, a user supplied label, and provide a text entry field which allows the user to change the parameter value. The \textit{StaticListParameterSteerer} is instantiated with a parameter name, and a list of choices. A drop down menu is constructed from the choice list, and when selected, the associated value is used to steer the parameter.

Code listing 24 shows the code for a viewer which allows the user to press a button and reset the LIFE application. When the button is pressed, the object sets the parameter "Restart Board" to the value of 1. This code is typical of steering objects.

The entire set of client objects will be expanded as additional simulations are instrumented. An object which permits user formatting of the display, including graphical
import StaticButtonSteerer

connection = Connection.Connection("LIFE")
button = StaticButtonSteerer.StaticButtonSteerer("Restart Board", 1);

# window code deleted
window.add(button.GUIElement())

button.SetConnection(connection)

Listing 24: A Code Fragment Using the StaticButtonSteerer Object

control of the type and precision is needed. More complex steering objects are also envisioned. These include standard controls such as sliders and dials.

D.8.4 Other Objects

Several other special purpose objects, which do not fit into the previous categories, have been constructed. These objects provide niche services. The OneTimeTrigger object is designed to set parameter values the first time the DoDataUpdate method is called. After this action is performed, the object calls the connection object’s UnRegisterViewer method and becomes inactive. This object is used to provide initial steering information the the simulation, and has no screen presence.

The ActivationViewer object is used to activate and deactivate elements of the GUI based upon a parameter from the simulation. The object is instantiated with a parameter name, and a list GUI elements and corresponding values. When a new parameter value arrives, the list of objects is checked. Those objects with values matching the parameter value are activated, all others are deactivated.

This class of objects will be expanded as pyGUIObjects are used to build additional
interfaces which require additional actions.

D.9 Derived Objects

One advantage of using an object hierarchy is that by extending existing objects, new objects can be created. Generally extending an object requires much less work than creating a new object from scratch as many of the methods are inherited from the existing class. Listing 25 contains the code for an UpDown object, which is derived from the StaticParameterViewer object. The new object watches a parameter, displaying up if the value of the parameter has increased from the previous iteration, and down if the value has remained constant or decreased. This is achieved by augmenting the instantiation function to set an initial value for the variable lastValue, and overriding the DoDataUpdate method entirely.

D.10 Compound Objects

Compound objects are built to perform special functions and are composed of simpler objects. Listing 26 contains code to build an UpDownLine object. This object contains a StaticLabeledParameterViewer and an UpDown object. The final object is a combination of these two objects, but as can be seen in Listing 27, is treated as a simple object by a viewer program.
from StaticParameterViewer import *
import GUIBase
import ClientBase

class UpDown ( StaticParameterViewer ) :
    def __init__(self, parameter) :
        StaticParameterViewer.__init__(self, parameter)
        self.lastValue = None;

    def DoDataUpdate(self, pv):
        if self.lastValue == None:
            self.lastValue = pv[0][1]
        else:
            if pv[0][1] > self.lastValue:
                word = 'up'
            else:
                word = 'down'
            self.lastValue = pv[0][1]
            self.outerElement.label(word)

Listing 25: Code to Create an UpDown Object
class UpDownLine(GUIBase, GUIBase, ClientBase, ClientBase):
    def __init__(self, parameter):
        GUIBase.GUIBase.__init__(self)
        ClientBase.ClientBase.__init__(self)

        self.outerElement = Fl_Pack(0, 0, 600, 30)
        self.outerElement.type(FL_HORIZONTAL)

        self.box = StaticLabeledParameterViewer(parameter);
        self.box.GUIElement()
        self.updown= UpDown(parameter);
        self.updown.GUIElement()

        self.outerElement.end()
        self.outerElement.hide()

    def SetConnection(self, connection):
        self.box.SetConnection(connection)
        self.updown.SetConnection(connection)

Listing 26: Code to Create an UpDownLine Object

connection = ConnectionTest.ConnectionTest()
client = UpDownLine.UpDownLine("random")

#window details deleted
window.add(client.GUIElement())
client.SetConnection(connection)

Listing 27: A Code Fragment which Employs an UpDownLine Object