The Language And Platform Independent Steering (LAPIS) System

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Harrison B. Smith, B.E., M.S.

Graduate Program in Electrical and Computer Engineering

The Ohio State University

2012

Dissertation Committee:

Dr. Ashok Krishnamurthy, Advisor

Dr. Bradley Clymer

Dr. Giorgio Rizzoni

© Copyright by

Harrison B. Smith

2012

Abstract

Computational steering technology was introduced in the late 1980s as a combination of real-time visualization and real-time manipulation of program state. Despite several decades of progress, computational steering has not seen wide-spread adoption due to a number of inherent problems with the technology. The paper presents a Language And Platform Independent Steering (LAPIS) system that addresses many of the these problems, and significantly lowers the technical barrier for entry into the use of computational steering, making it a viable technology for not just computer scientists, but for the scientific computing community at large. LAPIS is a language independent API standard that allows for interoperability between systems, languages, and models. Presented in this paper are the advantages of steering systems, as well as the problems. Additionally, the design considerations that led to the basic structure of the LAPIS system, a discussion of the architecture of the system, and a number of example applications that have been successfully implemented with the LAPIS structure are also presented.

To my wife, family, and friends who have all been very patient and supportive of me though this process

Acknowledgments

I would like to thank the my Advisor, Dr. Ashok Krishnamurthy, and my former advisor, Dr. Stan Ahalt for their continued instruction throughout my graduate education. I would like to thank Dr. Betty Lisa Anderson, and Dr. Bradley Clymer for their advise and encouragement. Also, thanks go out to the National Science Foundation, the Ohio Supercomputer Center and the Center for Automotive Research for their support of this research effort.

Vita

July 2, 1981	Born - Napbles, Florida, USA
2004	B.S.E. Electrical and Computer Engineering, The Ohio State University
2007	M.S.E. Electrical and Computer Engineering, The Ohio State University
2012	Ph.D. Electrical and Computer Engineering, The Ohio State University
2004-present	Graduate Research Associate, The Ohio State University.
2009	Graduate Teaching Associate, The Ohio State University.

Publications

Research Publications

H. Smith, A. Krishnamurthy, S. Ahalt "The Language And Platform Independent Steering (LAPIS) System". *IEEE Transactions on Parallel and Distributed Systems*, Submitted April 2012

H. Smith "LAPIS: A Unified Computational Steering Environment for Scientific Simulation and Modeling". *The Ohio State University Department of Electrical and Computer Engineering Poser Contest*, September 2010

J. Nehrbass, B. Guilfoos, H. Smith "A Graphical SAR Processing Suite" *WPAFB ATR Center Workshop, 2008*

J. Nehrbass, B. Guilfoos, H. Smith "The GOTCHA GUI Summary" *WPAFB ATR Center Kickoff Meeting proceedings, 2007*

H. Smith "Development of the SMART Coprocessor for Fuzzy Matching in Bioinformatics Applications" *Cray User Group 2006 proceedings*

H. Smith "A SAGE Analysis Coprocessor For Fuzzy Matching Using Reconfigurable Computing On the Cray XD1" *OCCBIO 2006 proceedings*

H. Smith "The SMART Reconfigurable Processor for Fuzzy Sequence Matching" *RSSI 2006 proceedings*

Fields of Study

Major Field: Electrical and Computer Engineering

Studies in:

High Performance Computing:

Software Techniques	Prof. Ashok Krishnamurthy
Hardware Techniques	Prof. Stanley C. Ahalt, UNC

Software Engineering
Computer Graphics

Table of Contents

Page

Abstract ii

Dedication iii

Acknowledgments iv

Vita v

List of Tables viii

List of Figures ix

1. Introduction to the Field of Computational Steering 1

 1.1 Computational Steering Defined 2
  1.1.1 Open- vs. Closed- Loop Simulation 2
  1.1.2 Components of Computational Steering Systems 3
  1.1.3 Organization of Computational Steering Systems 10
  1.1.4 A More Generalized View of Computational Steering Systems 10
 1.2 Motivations for and Advantages of Computational Steering 13
  1.2.1 Novel Problem Solutions With Computational Steering . . . 13
  1.2.2 Enhanced Efficiency and Speed With Decreased Effort With Computational Steering 14
 1.3 Contribution of LAPIS to the Field of Computational Steering . . . 15
 1.4 Definition of Terms Used Within This Work 16
 1.5 Organization of This Dissertation 17

2. Background Review of Literature on Computational Steering 19

 2.1 The Birth of Computational Steering 19
 2.2 The Evolution of Computational Steering 20

2.2.1 Forms of Computational Steering 21
2.2.2 Computational Steering Fields of Use 25
2.2.3 User Interfaces for Steering Systems 26
2.2.4 Steering Systems Extensions and Advanced Techniques . . . 29
2.2.5 Computational Steering System Scaling 33
2.3 Observations on the Field of Computational Steering 34
2.3.1 Modern Computational Steering 34
2.3.2 Commonalities and Base Structures Implicit in Computational Steering Systems and Techniques 34

3. Design of a Computational Steering System for Mainstream Acceptance . 36

3.1 Historical Trends in High Performance Computing Libraries 37
3.2 Consideration and Leveraging of Commonalities Within Computational Steering 38
3.2.1 High Level Design 38
3.2.2 Node-Level Design 42
3.3 Identifying and Addressing Problems in Existing Computational Steering Systems 45
3.3.1 Difficulty in Development of Computational Steering Applications 46
3.3.2 Maintainability Challenges With Steered Applications . . . 49
3.3.3 Portability Issues with Steered Applications 51
3.3.4 Structural Constraints Imposed By Steering Tools 53
3.3.5 Computational Steering and Data Issues 55
3.3.6 Performance Issues with Computational Steering Systems . 56
3.3.7 Supporting Software and Hardware by Steering Tools & Environments 57
3.4 Practical Design Considerations 58

4. The Development and Use of the LAPIS System 61

4.1 Structure and Behavior of the Overall System 61
4.2 Development of the LAPIS Software Layers 66
4.2.1 A Summary of LAPIS Control Messages 67
4.2.2 The LAPIS Daemon Layer Implementation 67
4.2.3 The LAPIS API Layer Implementation, Java Version 72
4.2.4 The LAPIS API Layer Implementation, MATLAB Version . 75
4.2.5 The LAPIS COM Layer Implementation 75
4.3 Developing With the LAPIS System 77
4.3.1 Prerequisites of Use 78
4.3.2 The LAPIS API Overview 78

4.3.3 Initialization of the API and Steering Network 86
4.3.4 Example Software Structures for Use with the LAPIS Steering System 87
4.4 Developing Extensions for the LAPIS System 94
4.4.1 LAPIS Control Messages 95
4.4.2 LAPIS API Layer Development: The LAPIS API Standard 102
4.4.3 LAPIS COM Layer Development: The LAPIS COM Standard 105
4.5 Summary of Key System Features, Advantages, and Abilities of the LAPIS System 107

5. Implementations of Example Simulations with LAPIS 109

5.1 Steering Conway's Game of Life 109
5.1.1 Technical Description of the Demo 110
5.1.2 Lessons Learned from this Demo 112
5.2 Managing Hill Climbing Clients with a Manager Interface Using LAPIS 112
5.2.1 The Hill-Climber Manager GUI 113
5.2.2 The Java and Matlab Hill Climbers 114
5.2.3 Lessons Learned from this Demo 116
5.3 A Simple Multi-Model Simulation of a Sailboat, Wind & Current Connected Via LAPIS 116
5.3.1 Wind and Water Models 117
5.3.2 Sailboat Models 118
5.4 Implementation of a Feed-Back Control System of a Driver & an Engine With LAPIS 120
5.5 The Use of LAPIS in Implementation of the CDI Multi-Model Simulation 123
5.5.1 The First Generation CDI Multi-Model: The Utility Provider Simulation Website 125
5.5.2 The Website 127
5.5.3 The Back End Manager 128
5.5.4 The PDM and POE Models 128
5.5.5 System Operation 129
5.5.6 The Second Generation CDI Multi-Model: The Continuously Executing Power Demand And Supply Model 131
5.5.7 The Third Generation CDI Multi-Model: The CDI Simulation System 133

6. Conclusion 135

6.1 LAPIS' Contribution to Computational Steering 135
6.2 Future Work 137

Appendices 138

A. Implementation Example Code Bodies 138

A.1 Conways Game of Life, Steered Version Code 138
A.1.1 Game of Life Code 138
A.1.2 Steering GUI Code 142
A.2 Hill Climbing Code 145
A.2.1 Hill Climbing Manager 145
A.2.2 Matlab Hill Climber 149
A.2.3 Java Hill Climber 152
A.3 Sailboat Model Code 158
A.3.1 Sailboat Code 158
A.3.2 Water Feild Code 160
A.3.3 Wind Feild Code 162
A.4 Driver-Engine Model Code 163
A.4.1 Driver Code 163
A.4.2 Engine Code 165

List of Tables

Table **Page**

4.1 A summary of LAPIS control messages and their usage. Details of message format and the LAPIS Command Message Standard can be found later in this chapter in sections **??**. 68

4.2 The standard methods the LAPIS API provides, as required by the LAPIS API standard (see section **??** for more detail on the standard. 79

List of Figures

Figure **Page**

1.1 Traditional vs. Steered Simulations: Traditional simulation is a very stochastic process of steps, each taking place in isolation, with the researcher at times playing no more roll than a process scheduler. Steering allows the researcher to act as an active participant in the ongoing simulation, allowing human insight and intuition to play a part in scientific computing 3

1.2 A Generalized Client/Server Steering System Architecture. Note the roll of actuators and sensors is dependent on the perspective of the user interface 5

1.3 An output interface showing currents and turbulence from a simulation of lake Eire circa 1990 [?]. Outputs of this system also included volume and contour renderings of temperature. 7

1.4 An early computational steering graphical user interface used in the simulation of turbulence in lake Eire circa 1990 [?]. Controls include buttons for controlling the simulation and slider bars for controlling various input parameters. 9

1.5 Despite significant differences in implementation, steering systems that run on a single system, a distributed system, or the grid are all identical when the specifics of the connecting structure are abstracted. 11

1.6 A Generalized Steering System Architecture. In this case, nodes can be user interfaces, simulations, data stores, or any other computational resource involved in the steering system. No distinction is made between nodes or between sensors and actuators 12

2.1 An example Model Exploration interface from [?]. The figure shows the deformation of a lattice of atoms during a molecular dynamics nanoindentation simulation. 22

2.2 An example of performance steering from [?]. The left hand figure shows a more sophisticated partitioning achieved though steering. The chart on the right shows the performance improvement though use of the technique. 23

2.3 An early computational steering implementation for use in medicine, specifically placement of electrodes for implanted defibrillator [?]. . . 26

2.4 An example of Model Exploration using a novel user interface, the CAVE virtual environment from [?]. The figure shows a user interacting with a running microwave radiation simulation via a virtual CAVE environment. 28

2.5 Checkpointing, a technique in which a running simulation periodically saves current state to disk, is a technique frequently used in high performance computing and simulation to ensure that in the event of a crash, computation can later be resumed with minimal loss of effort. With the addition of some sort of state management, as shown, a checkpointing system can be expanded into a simulation system that allows the end user to stop, rewind, replay, and explore alternative execution paths. Computational steering systems easily lend themselves to such techniques. 30

3.1 The high level architecture that will be used by the LAPIS system. Note that each application may use a different supported language and thus a different implementation of the API layer. Also, the LAPIS steering network it not explicitly implemented but rather is effectively created via a coordination of the Daemon layers, communicating though the COM layers. 40

3.2 Proposed Software Stack 42

4.1 Step 1: A new node, (A), starts and connects to the network via a known member of the network, (B). 63

4.2 Step 2: (A) sends a complete copy of its node list to (B), (B) sends a complete copy of its node list to (A) 63

4.3 Step 3: (A) compares the list received from (B), and notes all new nodes. (B) likewise compares list of received nodes from (A) and notes all new nodes . . . 64

4.4 Step 4: For all new nodes detected, both (A) and (B) send a new update message, including a list of all known nodes. Each of these nodes, (C) and (D), return a list of known nodes to (A). . . . 64

4.5 Step 5: All nodes compare received lists to their internal list of nodes. If, at this point, no new nodes are found, the update process ends. . . 65

4.6 Architectural structure of the LAPIS Daemon. The overall structure of the LAPIS Daemon follows the producer/consumer design pattern, where the producers are the Listeners and the consumer is the Worker. LAPIS Steering Network structure is stored via the Node and Data tables. . . . 69

4.7 Organization of computational threads involved in the execution of the LAPIS Daemon. The main execution thread first starts the API and COM Listener threads before entering the Worker routine. . . . 70

4.8 Roll of the Monitor thread in relation to the main user simulation thread. Nominally, while core simulation code is running, the Monitor thread is I/O blocking, using no resources. This only changes for two reasons. First, if the simulation leaves the computational kernel and calls a LAPIS API method, it will block on the method until the Monitor thread receives a responds and wakes the main thread. Second, when the monitor thread receives a request from the Daemon, the Monitor thread will process this request before resuming its I/O blocked state. . . . 73

4.9 The TCP/IP Com structure: A Copy of the Data Transfer Thread Object is spawned every time that a GET or SET command is issued and exists for only the duration of the data transfer. . . . 76

4.10 Pseudo-code example of use of the Constructor . . . 79

4.11 Pseudo-code example of use of the Initialize . . . 80

4.12 Pseudo-code example of use of the Connect . . . 80

4.13 Pseudo-code example of use of the Disconnect 81

4.14 Pseudo-code example of use of the Nodes 82

4.15 Pseudo-code example of use of the Close 83

4.16 Pseudo-code example of use of the Publish 83

4.17 Pseudo-code example of use of the Redact 84

4.18 Pseudo-code example of use of the Lock 85

4.19 Pseudo-code example of use of the Get 86

4.20 Pseudo-code example of use of the Set 86

4.21 Pseudo-code for implementation of simple simulation state sharing. Note that no changes are required within the main simulation loop at all. 89

4.22 Pseudo-code for implementation of simulation pausing. The only overhead when the simulation is not paused is a single 'if' statement once per simulation outer loop iteration. 90

4.23 Pseudo-code for implementation of a check-pointing node. An outer loop after the initialization and connection to the LAPIS network would allow for a repeated collection of checkpoints. 91

4.24 Pseudo-code for implementation of a rewinding node. A user interface would better allow for selection of previous states and handling cases of multiple rewinds per execution. 92

4.25 Pseudo-code for publishing data vectors that grow wile the execution of the simulation proceeds. Note that even though output is reinitialized at each execution of the loop, each copy is preserved within the LAPIS framework. 94

5.1 The Stock Game of Life Interface 111

5.2 Initialization of the Game Of Life, additions included to initialize the LAPIS API and publish data to the steering network. 112

5.3 Start of the Game Of Life simulation loop with added 'pause' code and X replaced with $X.d$ 113

5.4 The Steered Game of Life User Interface: Changes made in the steering interface (foreground) are reflected in the simulation (background) when the simulation is resumed. 114

5.5 The Hill-Climbing Director 115

5.6 The vector field generated at one particular time-step of the Water Models execution 119

5.7 The vector field generated at one particular time-step of the Wind Models execution 120

5.8 The Boat simulation is a simple multi-model simulation. The boat reads a position vector from both other simulations to determine it's next travel vector. 121

5.9 Driver/Engine Model is arranged in an asynchronous feedback loop where the instantaneous state of each model directly influences the next state of each model 122

5.10 Example output of the Driver/Engine model. In this case, the models result in an emergent behavior of oscillation around the speed limit. . 124

5.11 The structure of the 1st generation CDI Multi-Model. Note that the BEM and models may all be running on separate physical hardware. . 126

5.12 User interface web page for the power generation/consumption simulation 127

5.13 The structure of the 2nd generation CDI Multi-Model. Note the lack of static user interface as well as the lack of any central controller. . . 131

5.14 The structure of the 2nd generation CDI Multi-Model. Note the lack of static user interface as well as the lack of any central controller. . . 134

Chapter 1: Introduction to the Field of Computational Steering

The field of scientific high performance computing is populated by diverse problems each spanning a wide variety of technical and scientific areas. Solving these problems has resulted in many techniques for simulation, processing, and generally leveraging the computational power available today exist. Amongst these techniques there are high successful, well adopted paradigms, such as parallel distributed computing with MPI, OpenMP, and other standard libraries, or rapid prototyping with tools like MATLAB. At the same time, there are other techniques that, despite immense promise, have failed to become mainstream practices. Computational Steering is one such field with an enormous promise of benefit to the researcher, allowing them to act in concert with their simulations rather than as just a instigator and mortician to them. The obvious questions this raises are therefore "What factors have kept computational steering for being widely adopted?" and "What steps can be taken to address this problem and speed the adoption of this useful and highly valuable technique?" The body of work presented in this dissertation seeks to identify and resolve the reasons for this lack of adoption though the creation of a new and novel computational steering tool, the LAPIS System.

1.1 Computational Steering Defined

Despite more than two decades of research and development, significant theoretical as well as realized advantages (for example, [?]), and a wide variety of available tools [?,?,?,?,?,?], computational steering has failed to become a mainstream computational paradigm [?]. Computational steering (or simply 'steering') can be defined a number of different ways. Most generally, steering can be thought of as any computer application that both: 1) presents its current internal state and, 2) allows manipulation of that state to effect its output. However, this definition is overly board, as nearly all interactive applications, video games, word processors, development environments, fall into this category. In practice, the concept of computational steering is further refined by requiring that, without user input, the program can, nominally, do something [?], and additionally, that the program in question is a simulation or other scientific, technical, or high performance application. This group of applications will nominally be refer to as 'simulations'.

1.1.1 Open- vs. Closed- Loop Simulation

Traditional computational simulation (see Figure 1.1) is an iterative loop of three primary steps: running a simulation, analyzing results, and refinement of parameters. Each step in the loop must finish before the next step can begin, and each step typically requires user interaction in order to start execution, making the process "open-loop". Computational steering developed as a merging of traditional computational simulation and real time visualization as a means of "closing the loop" [?] by allowing all three steps of the simulation process to happen simultaneously. Computational steering has also been defined as the interactive control over an executing

computational process [?] which describes any of a large number of computational systems wherein the user is able to direct the flow of computation in real time via textual, graphical and other interfaces.

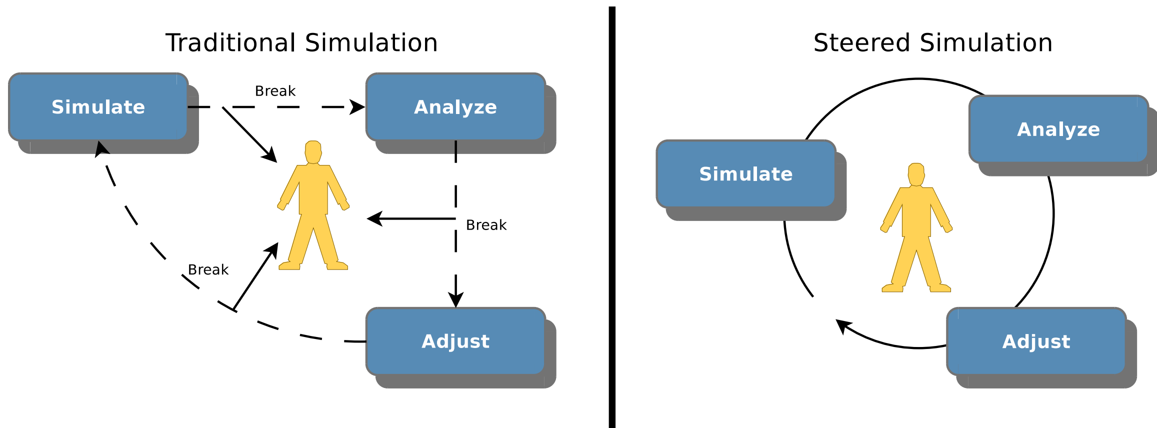

Figure 1.1: Traditional vs. Steered Simulations: Traditional simulation is a very stochastic process of steps, each taking place in isolation, with the researcher at times playing no more roll than a process scheduler. Steering allows the researcher to act as an active participant in the ongoing simulation, allowing human insight and intuition to play a part in scientific computing

1.1.2 Components of Computational Steering Systems

There are two important functional aspects to a "closed loop" computational steering system [?]. First the system must allow the researcher to visualize the running simulation in real time. In doing so, the researcher can directly observe the inner workings of their simulation through the "window" granted to them via this real time visualization. This changes the researchers simulation from an abstract computation to something more like a virtual laboratory experiment. Such a window allows problems with the simulation, such as erroneous or uninteresting behavior, to be detected

early, allowing for early termination of faulty experiments [?, ?, ?]. Additionally, this "window" allows the researcher to watch the evolution of the system state as the simulation progresses which may well illuminate behaviors that would not be present in the output of the simulation.

The second functional aspect computational steering systems must posses is the ability of the user to manipulate data within the simulation in real time. While the ability to manipulate the internal state of a running simulation is useful in its own right, the addition of the "window" described above makes such manipulations far more powerful. Such systems change the very nature of computational simulation from that of an iterative "test, check, retest" tool to something much more powerful, intuitive, and immersive. Simulation changes from an inherently iterative tool to an interactive one, allowing researchers to gain additional insight into the principles underlying their simulations [?, ?]. In fact, a number of one-off system implementations demonstrate the variety of uses this paradigm makes possible. For instance, engineers can use simulation to help design and optimize new components or products [?]. Steered simulations can be used as learning environments for students [?] as well as experimental tools [?]. Finally, steering reduces wasted computation by allowing a more immediate response to poorly running simulations [?], the early termination of experiments that are erroneous or have reached a steady state [?, ?], and the reuse of previously performed computation [?, ?, ?].

In typical computational steering implementations there are three distinct portions of code that must be considered [?] (see Figure 1.2). The most obvious component of the steered simulation is the simulation itself, weather preexisting and modified for steering, or developed within a steering framework, or otherwise. The second

component, nearly as obvious, is a user interface through which the end user can steer the simulation. The last component, referred to as middleware, sometimes seen as a collection of sensors and actuators [?], allows the end user visualize the current state of the simulation, and affect that same state by connecting the user interface to the simulation. Each of these components is important in it's own right and each are discussed separately below.

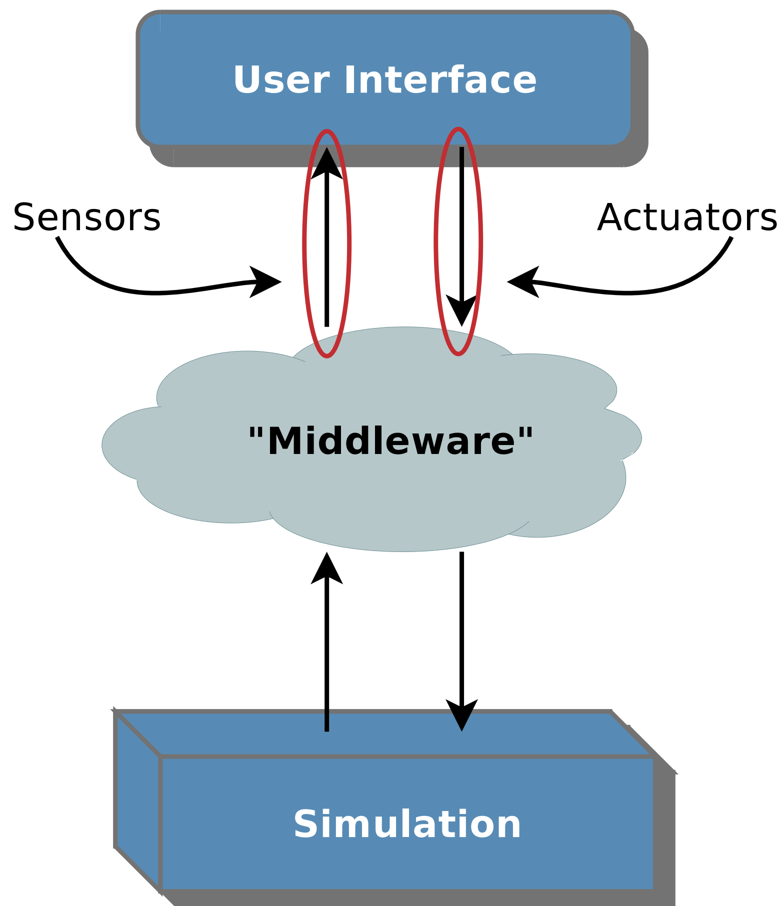

Figure 1.2: A Generalized Client/Server Steering System Architecture. Note the roll of actuators and sensors is dependent on the perspective of the user interface

The User Simulation

In order for a simulation to make use of steering, the simulation must either have been developed as a steered application from the beginning, or alternatively been an existing code that was fully developed prior to any intention of using computational steering. Implementing steering in the latter of these cases requires additions to the simulation source code. These additions allow external code to read and write to key variables within the simulation, and through the manipulation of these variables steering is enabled. The process of editing the source code of an existing simulation to add steering functionality is referred to either as "tooling" or "instrumentation" [?].

In theory, computational steering is applicable to nearly all areas of simulation and with the right tools any simulation could be tooled appropriately to allow steering. However, in order to see significant benefit from computational steering the simulation must be sufficiently complex and long running so that the user has time to connect, observe, and adjust parameters in real time. Simulations with extremely large parameter spaces, long running iterative simulations, and simulations who's internal state is easily representable graphically tend to be the most suitable targets for computational steering implementations. There are numerous examples of steered simulations that have been quite successful, including applications in medicine [?], Computational Fluid Dynamics [?], Molecular Dynamics [?], as well as many others. A more thorough discussion of computational steering applications can be found in section 2.2.2.

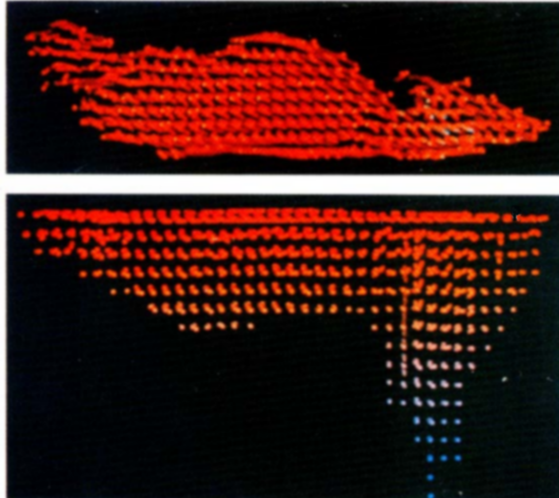

Figure 1.3: An output interface showing currents and turbulence from a simulation of lake Eire circa 1990 [?]. Outputs of this system also included volume and contour renderings of temperature.

The User Interface

The second portion of code critical to steering systems is the user interface which is composed of all the mechanisms that allow the user to view and manipulate the simulation state. Early computational steering systems, being derived from real-time visualization systems of the time, used simple 2-D graphs [?] (see figure 1.3). Today, in addition to such 2-D output, mechanisms for the display of simulation state include text output of numerical values, or full 3-D renderings [?, ?, ?], or even three dimensional displays shown through specialty hardware [?, ?]. Mechanisms for the manipulation of simulation state were again originally derived from real-time simulations, and as such took the form of 2-D GUIs [?] (see figure 1.4). Inputs can additionally come in the form of a scripting language or console-style text input [?], 3-D widgets embedded in the graphical display [?], or even monitoring programs [?].

Through a combination of both display and manipulation mechanisms a user interface for computational steering is created. In the case of multi-model simulations, there are many cases where the output of one model acts as the input to another model, in which case there need not be any human user interface between these two models. Instead, such systems are built around programming interfaces of various types and structure, and a user interface for steering may interact with one or only a few, or every model involved in the multi-model simulation. In all cases, such input and output interfaces can be seen an opening though which data may be added to or extracted from the simulation.

The Middleware, Sensors & Actuators

The third component involved in steering systems is the interconnection between the user interface and the simulation, often referred to as "middleware." This middleware is responsible for forwarding information from the simulation to the user interface for rendering and receiving input from the user interface and applying that input to the simulation in order to affect change. The requirements for middleware can change dramatically from system to system and simulation to simulation. This results in middleware that is most often custom coded for each situation, which in turn results in poor portability and reusability of the middleware. Additionally, the middleware must deliver high bandwidth, low latency communication between the user interface and the simulation in order to make the computational steering tool as effective as possible [**?**]. These additional requirements make the development of middleware highly difficult but also critical to the overall performance of steering systems. The functional components of middleware are sometimes collectively referred to as sensors and actuators. A sensor is any mechanism that shows information to the

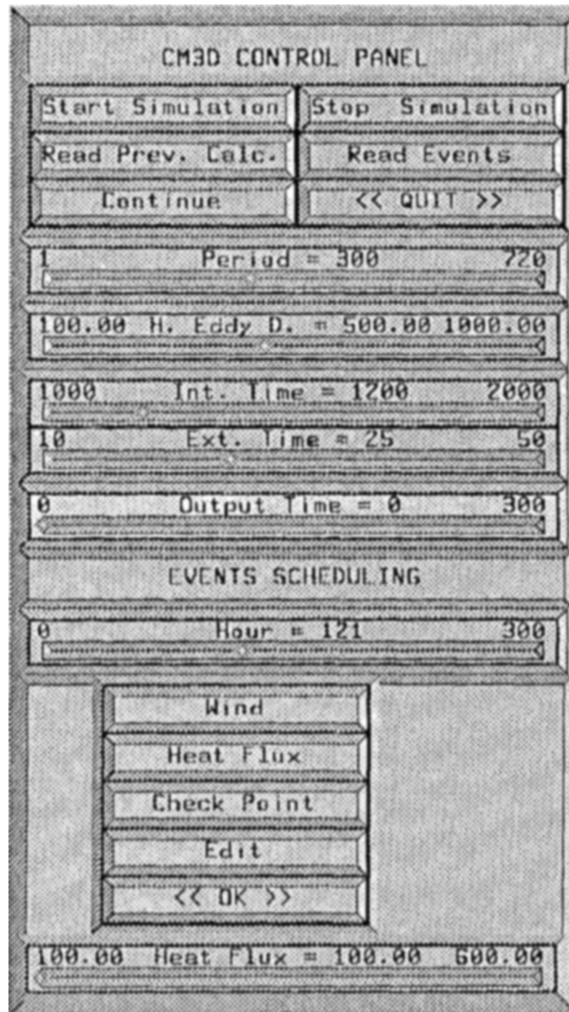

Figure 1.4: An early computational steering graphical user interface used in the simulation of turbulence in lake Eire circa 1990 [?]. Controls include buttons for controlling the simulation and slider bars for controlling various input parameters.

user and, conversely, an actuator is any mechanism that manipulates data within the simulation. Interestingly, from the perspective of the simulation, the rolls of sensors and actuators are reversed.

1.1.3 Organization of Computational Steering Systems

Steering systems found throughout the literature make use of the components discussed above in a wide range of organizational paradigms. The most common of these is a client-server model wherein the simulation code acts as a data server to a steering front end, often in the form of a GUI (as seen in [?], for example). Another common model is a client-distributed worker model, where the user interface plays the roll of the client to the system, much the same as the client-server model. However in this case, the back end 'server' of the system is a collection of computers processing in parallel, often as part of a larger supercomputer cluster (see [?] for example). Other systems exist as hybrids of these forms [?, ?], or in other configurations such as found in [?], where several layers of management applications separate the end user from the graphical and numerical applications they are steering. Some systems are even more complicated, involving specialty hardware for certain tasks, such as [?]. Many of these models can also be found with a discrete and separately running data repository system of some sort [?]. While the organization of these systems varies quite a bit, the one commonality between them all is a communication medium, usually network base, over which all the data exchange, coordination, and other communication occurs.

1.1.4 A More Generalized View of Computational Steering Systems

When considering the components (section 1.1.2) and organization (section 1.1.3) of computational steering systems, a number of important observations can be made. Most importantly, a totally generic model of steering can be constructed. Specifically, steering systems can all be modeled as a number of pieces of code running in separate

memory spaces, connected via a middleware by way of actuators and sensors. These separate memory spaces may be different locations in the physical memory of a single machine, different physical computers in a parallel system, or even different computer centers located on a computational grid (see figure 1.5).

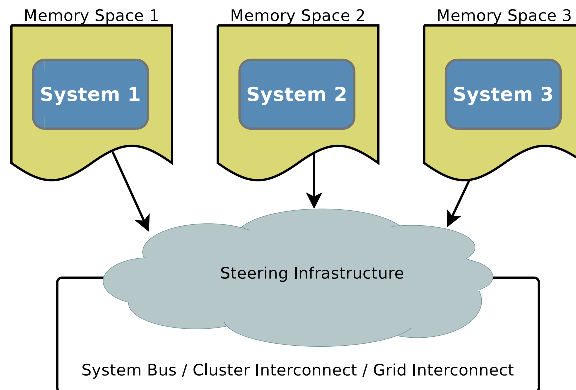

Figure 1.5: Despite significant differences in implementation, steering systems that run on a single system, a distributed system, or the grid are all identical when the specifics of the connecting structure are abstracted.

It should also be noted that, when creating a steering tool, there is no reason for the tool to distinguish between steered code and interface code. Weather the end user wishes to create a client-server organized steered simulation, or use any other organization, a tool that makes no such distinction can be used and intentionally limited in it's use to create such a system. Conversely, a steering tool that requires a client-server model, for instance, would be more difficult to use in modeling a complex coordination of components.

Along a similar line, when creating a steering tool, there is no compelling reason to distinguish between actuators and sensors. As noted in section 1.1.2, what constitutes a sensor or an actuator is purely a matter of perspective. Rather than

Sensors and Actuators, a more general “published data” model can be used. In the a published data model, each component of the steering system (be it simulation, interface, data repository, etc.) simply makes the data available, or “publishes” it. Any other component of the system can act upon that published data when and how they like.

From all these observations, it can be concluded that computational steering is a communication problem, albeit with some special requirements. Specifically, communication should be initialized unilaterally, without a priori knowledge of when such communications will occur on the part of the other components in the system. Also, it can be seen that most, if not all steering systems could be implemented on a peer-to-peer network of separately running components (see figure 1.6), all utilizing the same communication mechanism to share data between each other arbitrarily.

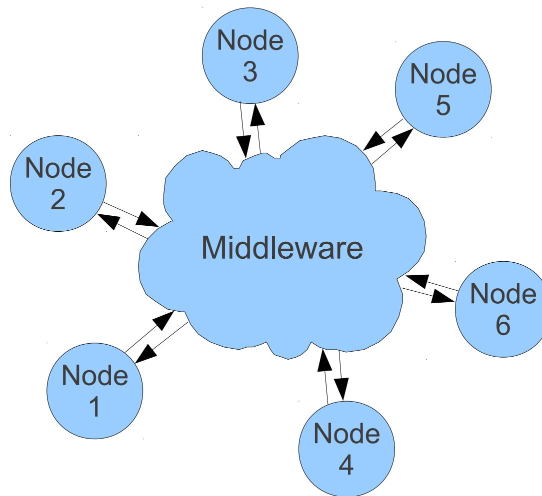

Figure 1.6: A Generalized Steering System Architecture. In this case, nodes can be user interfaces, simulations, data stores, or any other computational resource involved in the steering system. No distinction is made between nodes or between sensors and actuators

1.2 Motivations for and Advantages of Computational Steering

Computational steering provides a number of advantages that are difficult or impossible to obtain without the use of steering. While there are a number of distinct advantages, they generally fall into either novel ways to approach computational problems or advantages in computation time, effort and resource usage.

1.2.1 Novel Problem Solutions With Computational Steering

As discussed later in section 2.2.1, there are four notable forms of computational steering, model exploration, performance steering, algorithm exploration, and multi-model simulations. All four of these forms of simulation are difficult, if not impossible to achieve using traditional simulation techniques. Model exploration represents a separate paradigm from traditional simulation, where, by definition, computational steering is involved. Model exploration allows the researcher to apply domain knowledge, intuition, insight, and hard-to-imitate human thought processes to the act of simulation and problem solving. Such systems present opportunities for investigation that are unparalleled in traditional simulation. While performance analysis and optimization can certainly be performed without steering in many cases, in some cases, the ability to apply human thought processes to the task has yielded significant results where programmatic optimization fails. Algorithm development can certainly take place without computational steering, and indeed even without computation to some extent. However, the highly interactive nature of algorithm steering systems provide an unprecedented level of flexibility, allowing the user to experiment in ways never before possible. While there are certainly examples of multi-model simulations

that do not use computational steering systems, such structures fit very naturally into the computational steering paradigm, as interactions between models fall naturally into the definition of steering, where the input-generating model acts as the steering agent for the input-accepting model. Implementation of multi-model simulations with a steering system should, in theory, be nearly a trivial amount of increased effort over that of developing the models themselves, and indeed, this proves to be true with LAPIS, as discussed later in section 5.5.6. All four of these examples represent new ways to think about problem solving that have, in the past and without steering, been impossible or rejected due to the high level of difficulty in implementation.

1.2.2 Enhanced Efficiency and Speed With Decreased Effort With Computational Steering

In general, in open-loop scientific simulation (see figure 1.1), a researcher will first create a set of parameters which determines the output of the simulation at run time. As each run of the simulation produces only one set of results, the researcher must use their best judgment in choosing starting parameter sets. The researcher can apply extensive experimental observation, theoretical knowledge, experience, and/or results of prior simulation runs, once prior to the new simulation run in order to chose a parameter set that will result in the most interesting, useful, or illustrative simulation run. Once all of these factors are considered by the researcher, the parameter set is created and the simulation is run. After the simulation finishes execution, the researcher analyzes its output and determines whether or not the parameter set needs further refinement.

Open loop simulation is inherently inefficient for several reasons. First, the researcher cannot determine whether the choices made during the creation of the parameter set were well-considered until post-simulation analysis of results. Only after a complete run of the simulation will the scientist be able to analyze its output and determine the quality of the chosen parameter set. Only after this analysis can new insight gained from the simulation run be incorporated into subsequent simulations [?, ?]. Secondly, the linear nature of the open-loop methodology prevents efficient exploratory analysis of the simulation parameter space [?, ?] as exploring any single point in the simulation space requires the simulation to be run to completion. Finally, if during a given run the simulation enters an uninteresting state, or an erroneous state, the computation time used by the that run of the simulation is wasted [?, ?]. Computational steering is capable of addressing all of these inefficiencies by "closing the loop," allowing researchers to visualize and interact with their simulations as they are running, mitigating these problems [?, ?].

1.3 Contribution of LAPIS to the Field of Computational Steering

This dissertation presents an important and novel approach to computational steering. Specifically, the development of the Language And Platform Independent Steering (LAPIS) system was novel in it's consideration of the full field of computational steering, its history, its benefits and limitations, the tools targeting end-users, and with a historical understanding of similar developments, like that of OpenMP and the MPI standard library. The implementation of LAPIS is likewise novel in its platform and programing language agnosticism, and its very high level of flexibility coupled with a minimal amount of duplicated implementation effort between target

languages, platforms, and scales. Finally, the use of the LAPIS system in implementing steering systems has been kept as simple as possible, opening up the usage of the LAPIS system to a wide range of end users, most importantly including those who are not intrinsically a part of the computer engineering/computer science community already. At the same time, the LAPIS system is complete enough in implementation to allow for highly sophisticated tools to be constructed.

1.4 Definition of Terms Used Within This Work

Before proceeding to the remainder of this work a number of key terms should be defined and understood by the reader, as they will be used extensively for brevity as well as clarity of meaning.

middleware: All of the software involved in connecting nodes in the steering network to one another.

node: Any computational process, regardless of function, making use of a computational steering tool to act as part of a steering system. Specifically, with LAPIS, any process that instantiates an instance of the LAPIS API and connects to the LAPIS steering network is defined as a node.

publish: The act of one node making data available to other nodes on the steering network. Past tense form: available for reading or writing.

simulation: Any process within a steering system that produces data that will be used by either the end user or other nodes within the steering system

steering: Used to refer the act of steering. Other forms will be used with similar meaning.

steering network: A collection of nodes all connected via the same steering tool and acting in coordination as a computational steering system. Specifically, with LAPIS, the term steering network refers to the peer-to-peer network formed by the nodes.

steering system: See: steering network

tooling: The act of adding code to an existing program in order to allow it to be used as part of a steering system.

1.5 Organization of This Dissertation

The dissertation has so far defined computational steering, including steerings components, and system level organization. In chapter 2, a summary of the historical development of the field is presented, considering forms of steering, areas of use, development of user interfaces, conceptual and technical developments, and the scale of computational steering systems. In chapter 3, an analysis of computational steering systems and research is presented that explains the failure of wide spread adoption of computational steering as a scientific computing technique. Additionally, a new approach to computational steering systems is presented, along with an analysis that shows how the organization and design of the LAPIS system will overcome the limitations is discussed in chapter 3. Chapter 4 presents the LAPIS standard definitions, a detailed view of the implementation of all components of the LAPIS system, as

well as a detailed explanation of the usage of the LAPIS system. Finally, chapter 5 presents a number of example implementations that make use of the LAPIS system to create a variety of steering simulations and systems.

Chapter 2: Background Review of Literature on Computational Steering

In order to understand both the motivations and importance of computational steering, as well as understand steering's lack of acceptance, it is important to first understand the history and development of the field. This chapter lays out a time line of the developments in computational steering since it's inception. Included below is a discussion of the forms of computational steering, areas of usage, areas of development and experimentation, as well as scaling effort.

2.1 The Birth of Computational Steering

The field of computational steering is entering its third decade of existence since the idea was laid out in the paper "Grasping Reality Through Illusion – Interactive Graphics Serving Science" by Fredrick P. Brooks, Jr., 1988 [**?**]. This work presents the idea of extending real-time visualization by allowing user interaction. The impetus for the development of this idea came from a long history of simulation and visualization.

Over the decades since the ENIAC was used to calculate artillery trajectories for the army during World War II [**?**], computers have played an increasing role in scientific endeavor, particularly in the simulation of experiments to dangerous, costly, or difficult to perform. Since the ENIAC, simulations grew in scale, complexity,

and ambition to the point where simulations generating simple numerical values or graphs were no longer sufficient for meaningful analysis [?,?]. The growing complexity of data generated by simulations led to the development of more specialized and sophisticated graphical display techniques, which are now collectively referred to as visualization [?,?].

With visualization aiding analysis, simulations grew further in scale and complexity to the point where visualization after the fact was no longer sufficient to make effective use of simulation [?]. With this growing complexity came growth of simulation run times to days, weeks, or even months. Consequently, poorly chosen starting parameters could potentially waste large amounts of computation time. Overall, the open-loop design of simulation was, and still is, insufficient for timely analysis of large, long running simulations.

2.2 The Evolution of Computational Steering

The first intellectual step toward computational steering was in 1988 by Brooks [?] when the idea of interacting with a running simulation was introduced. The need for interactive simulations, combined with the research momentum behind modern visualization techniques in the late 1980s led to the development of the first computational steering applications. Subsequently, research in computational steering can be divided into a number of categories, including fields of use, user interfaces, steering techniques, and system scaling. Progress in each of these areas is discussed in further detail below.

2.2.1 Forms of Computational Steering

While development of computational steering applications often requires a different thought process from traditional simulation development, steering opens the door to a number of simulation models that are otherwise unavailable, inefficient, or extremely difficult to implement using traditional thinking and traditional simulation methodologies. Of particular note are model exploration systems, performance optimization systems, algorithm exploration systems [?, ?, ?], and multi-model systems. Each of these classification is discussed in more detail below.

Model Exploration

Model Exploration systems allow the user to interact with a simulation, algorithm, or model in real-time in order to gain increased depth of understanding, insight, and intuition [?, ?, ?, ?]. Examples of such systems include a 3D turbulence model of lake Erie [?] (see figures 1.4 and 1.3 in section 1.1.2), developed at the Ohio Supercomputer Center, computational mechanics simulations [?] from the University of Illinois, molecular dynamics simulations [?, ?] (see figure 2.1 below) from the University of Utah, and microwave radiation interaction simulations [?] (see figure 2.4 from the Navel Research Laboratory, along with many other applications [?]. For several reasons model exploration is the most frequently encountered class of steering in the literature. First, steerings most obvious use is steering the simulation itself, allowing for experimentation with the underlying scientific principles that are being modeled. Second, given the large number of existing scientific simulations, there are many opportunities for the creation of model exploration steering systems. Finally, many of the recognized benefits of steering are particularly applicable to model exploration.

Specifically, the ability to guide a simulation into interesting regions of execution [?], the noted enhancements in comprehension and insight that steering provides [?, ?], and the ability to play "what-if" games [?, ?], among other benefits, have all served as incentives for the development of model exploration steering. In addition to the benefits above, model exploration systems can be used for virtual prototyping [?, ?, ?] and teaching applications [?].

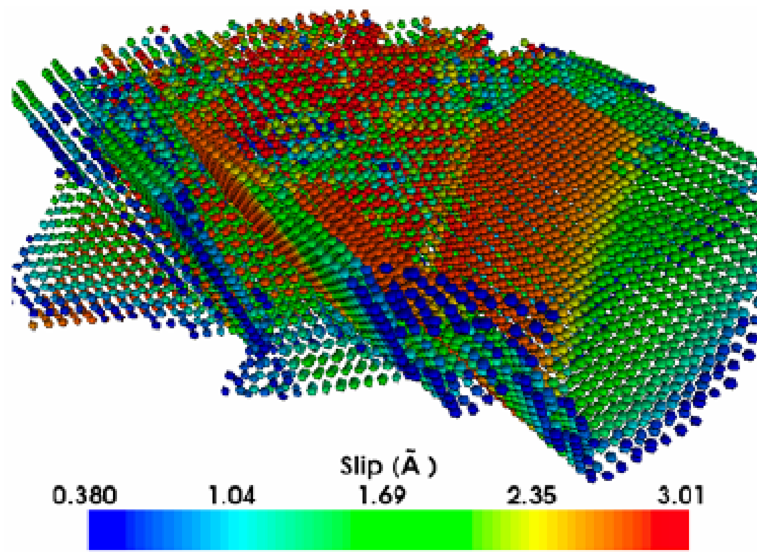

Figure 2.1: An example Model Exploration interface from [?]. The figure shows the deformation of a lattice of atoms during a molecular dynamics nanoindentation simulation.

Performance Optimization

The performance optimization class of steering systems manipulate the running simulation in ways that only affect its performance, not the results of the underlying simulation, for instance, the load balancing of computational workload between

processors in a distributed computation. Performance optimization steering evolved out of interactive debugging tools as well as performance analysis tools. Coupling these ideas with that of interactive monitoring and steering of an application created the performance optimization class of steering application. There are many examples of successful performance optimization applications including a load balancing tool for molecular dynamics simulations [?]. There are also several development environments built specifically for performance steering, including the Pablo Performance Environment [?] from the University of Illinois, the Falcon Steering System [?] (see figure 2.2) from the Georgia Institute of Technology, and a tool for the RealityGRID environment [?]. With these tools, developers can create systems for interactive load balancing of applications, allowing the application of human intuition for effective load balancing. The application of human intuition is useful in situations where information that cannot be captured algorithmically can be used to achieve higher performance and efficiency [?].

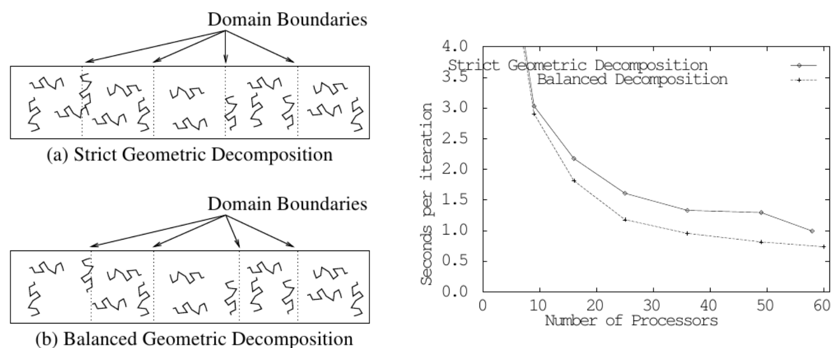

Figure 2.2: An example of performance steering from [?]. The left hand figure shows a more sophisticated partitioning achieved though steering. The chart on the right shows the performance improvement though use of the technique.

Algorithm Exploration

Algorithm exploration is the third class of computational steering applications where users can experiment with computational methodologies within simulation code. This includes trying different algorithms or implementations for common computational tasks, like Fourier transforms, root finding, searching, and others. Examples of the algorithm exploration class are rare, largely because algorithmic steering is of more limited use and requires significantly more program modification and sophistication. While the literature contained few distinct examples of algorithm exploration applications [**?**, **?**], algorithmic steering is perhaps the most interesting use of steering. In effect algorithmic steering acts as an application that helps the scientist develop the simulation itself, not just experiment with a steered simulation.

Multi-Model Simulations

Multi-model simulations are simulations made of collections of independently functioning models. Such simulations are often used to unify initially disparate work from different researchers. Linking independent models into a multi-model simulation is a complicated task involving significant modification to each of the individual models. Modifications including implementing communication protocols between models, synchronization, data transformation and so on. Unlike the other three forms of computational steering, multi-model simulation arose external to the field of computational steering. However, multi-model simulations fit into the computational steering paradigm quite well and through the use of computational steering, it is possible to minimize the required modifications and reduce the complexity of such integration. Steering tools can potentially handle communication and data transformation, as well

as provide mechanisms for model synchronization and the transfer of data between models. While work in this area is sparse, we have demonstrated implementation of such a multi-model simulation via both the Driver/Engine demonstration model and the CDI Web Simulator, both discussed below in sections 5.4 and 5.5.2.

2.2.2 Computational Steering Fields of Use

Given the widespread adoption of simulation as a useful scientific technique, it is surprising that computational steering has not been more frequently used [?, ?, ?], though there have been a number of significant and interesting research efforts into its use. One of the first steering applications was a large scale (for its time) fluid dynamics and weather modeling program developed at the Ohio Supercomputer Center to study turbulence in Lake Erie [?]. Three years later, the VASE steering environment was developed and used to steer computational mechanic simulations [?]. At the same time, steering was first being used for finite element analysis [?], n-body simulations [?] and hydrodynamics [?]. In the years subsequent to this, steering has been leveraged in numerous other areas including molecular dynamics [?] and medicine [?] (1994) (see figure 2.3), atmospheric simulations [?] (1995), genetic algorithms [?] and volume rendering [?] (1996), nuclear research, gas dynamics, and astrophysics [?] (1997), microscopy [?] and seismology [?] (1998). Since 2000, computational steering has found additional use in other fields including, but not limited to, manufacturing [?], microwave simulations [?], civil engineering [?], HVAC design [?] and computational fluid dynamics [?, ?]. While many research areas have examples of steering applications, there are many fields in which the technique has not yet been leveraged. Despite the demonstrated advantages of flexibility, power, insight,

and efficiency provided by steering, widespread acceptance of the technology has been slow.

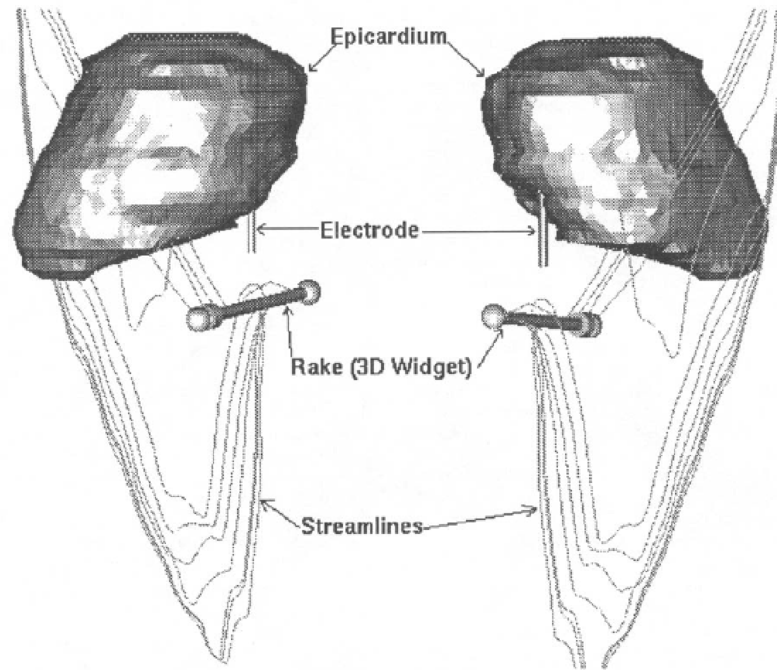

Figure 2.3: An early computational steering implementation for use in medicine, specifically placement of electrodes for implanted defibrillator [**?**].

2.2.3 User Interfaces for Steering Systems

While Brooks envisions many interesting interfaces for computational steering systems, few have been realized [**?**]. The first generation of steering user interfaces used basic, two dimensional application windows for input, much like those of common desktop applications (see figure 1.4 [**?**]). However, even the earliest steering applications made use of three dimensional renderings for output of simulation results to the user (see figure 1.3). Some more recent work has included efforts to create

systems supporting dynamic user interfaces based on XML mark-ups or other such information exchanges [?].

Given the multidimensional data that computational steering applications handle, the limitations of two dimensional interfaces were quickly realized and user interfaces were adapted to accommodate this high dimensional data. Within four years, there were already example systems where the simulation state was displayed as a three dimensional rendering and through manipulation of this rendering, the user provided input to the system [?, ?] (see figure 2.3 for an example of the SCIRun interface). Shortly thereafter, an effort was made to control a steered simulation through scripting languages and console like interfaces [?]. Another alternative development was the use of an algorithm to steer simulations [?, ?].

All of the above steering methods have advantages and disadvantages. Simple two dimensional interfaces are easy to develop and are easy to use, however they are limited in their flexibility, intuitiveness and ability to handle complex data. Three dimensional interfaces are far more powerful and intuitive and can handle complex, high dimensional data, but require significantly more development effort and can be inflexible. A scripting language provides both flexibility and power, but requires a highly trained user, perhaps even the developer of the steered simulation. Finally, having an algorithm steer a simulation allows mundane and repetitive steering tasks to be handled by the computer, freeing the user to focus on more important issues during computation. However, algorithms such as this cannot handle all steering decisions and can be very inflexible. Today, examples of all of these methodologies are present in the literature.

In addition to these developments, a number of other novel ideas and approaches to interacting with simulation state have been developed. Steering environments have been developed that include parameterized cameras, allowing the user to link the cameras settings, location, and operation to internal simulation data [?]. Other systems have developed methods for providing haptic feedback to the user based on simulation state and the users interaction with said state [?]. Other systems now allow multiple users to interact with simulation state simultaneously [?], while still others have been created to leverage much more immersive virtual environments [?, ?, ?] (see figure 2.4 for example). Finally, in our ever-more connected and networked world, steering interfaces have been developed for use via the web [?, ?] and though web-connected PDAs [?].

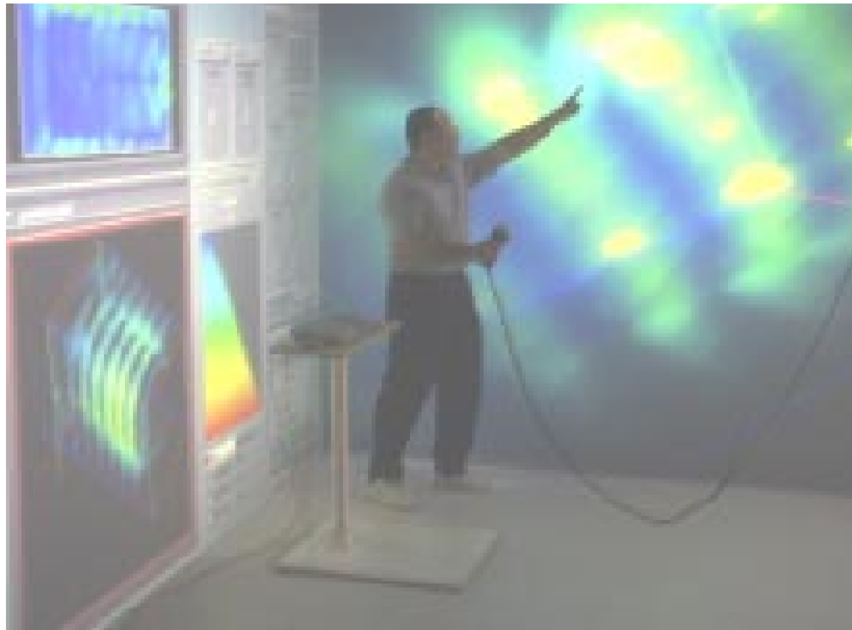

Figure 2.4: An example of Model Exploration using a novel user interface, the CAVE virtual environment from [?]. The figure shows a user interacting with a running microwave radiation simulation via a virtual CAVE environment.

2.2.4 Steering Systems Extensions and Advanced Techniques

Over the last two decades of development, considerable effort has been expended in developing the three computational steering techniques discussed in section 2.2.2 specifically related to computational steering, namely model exploration, performance steering, and algorithmic steering. Research in these areas have created numerous discrete instances of steering applications, as well as a number of steering development environments. These efforts have helped to refine the field in general, and have led to research in other aspects of steering. The extensions and techniques developed to expand the function and enhance development, debugging, and performance of steering systems are discusses topically below.

Check-pointing

Check-pointing, first demonstrated in a steering system in [?], is the process of periodically saving the internal state of the steering application as it runs. These saved states can be used to implement "Stop and Rewind" techniques [?] (see figure 2.5), allowing the simulation to be rewound and rerun, either to re-observe the simulation run or to start a new branch of execution by changing a key variable. Using a check-pointing system to explore various simulation paths is often referred to as 'Parameter Searching' as it allows the user to search for optimal parameter sets given a fixed staring point. Parameter searching allows for efficient experimentation as shown in [?]. There have also been research efforts in using check-pointing for fault tolerance and computational relocation [?]. While check-pointing is not exclusive to computational steering, steering combined with check-pointing provides a distinct advantages that traditional check-pointing simulations cannot emulate.

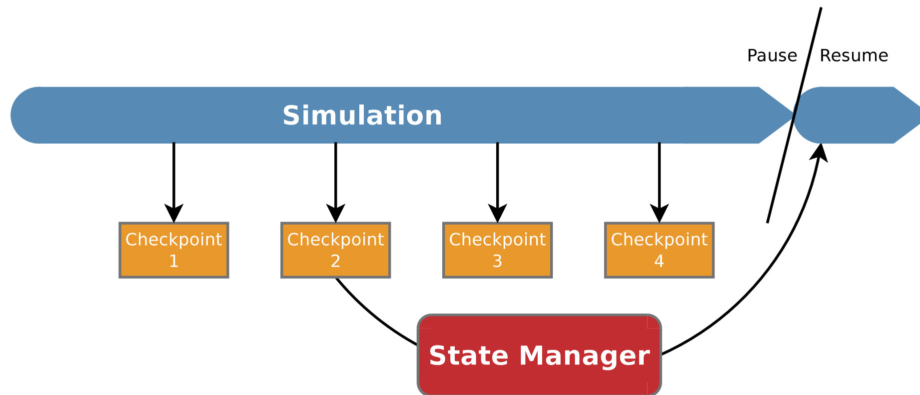

Figure 2.5: Checkpointing, a technique in which a running simulation periodically saves current state to disk, is a technique frequently used in high performance computing and simulation to ensure that in the event of a crash, computation can later be resumed with minimal loss of effort. With the addition of some sort of state management, as shown, a check-pointing system can be expanded into a simulation system that allows the end user to stop, rewind, replay, and explore alternative execution paths. Computational steering systems easily lend themselves to such techniques.

Consistency

Another direction of research is consistency considerations. Inconsistencies between the state of the simulation and the state of the visualization presented to the user can cause considerable problems, including decreased efficiency and incorrect steering [**?**]. There have been a number of research efforts aimed at developing a visualization that is consistent with the internal state of the simulation [**?**]. Work in this area is generally applicable to the entire field of computational steering in theory, but due to the fragmented nature of the development tools, implementation of these concepts is difficult.

Collaboration & Algorithmic Steering

With the ability to expose the internal state of a simulation, computational steering provides a unique opportunity for researchers to make observations and work together on simulations, both in development, and at run-time. A number of efforts have been made to enable such collaboration [?, ?, ?, ?]. These systems allow multiple researchers to access, observe, and manipulate a remote simulation from geographically separate computers.

Rather than having multiple researchers work together, another branch of research has focused on allowing a researcher to work in coordination with code designed to steer a simulation algorithmically. First described by Vetter and Schwan in [?] and [?] and later implemented by Swan et. al. in [?], algorithmic steering systems are controlled by a user defined algorithm rather than user input. Algorithmic steering allows the user to focus on system behavior rather than system control. If a situation arises that the algorithm is incapable handling, the user can take manual control until the algorithm can resume control. Algorithmic steering has been used to in a variety of areas including computational fluid dynamics, product design, and microwave radiation simulation [?, ?, ?].

Development and Design

Significant research efforts have also been made in the advancement of development techniques. Most notable amongst this research are the efforts to develop visual programming models and languages [?, ?] in which a developer uses structural blocks to build simulations. The intent behind such systems was to ease the traditionally complex task of simulation development, particularly when steering is involved.

Though the use of structural blocks which already contain the steering elements, the task of tooling a simulation was largely removed from the developer of the simulation. Along very similar lines, there has been work in developing a data-flow model for steered simulations [?] where the main organization of the code centers on how data moves though the simulation.

There has also been work regarding some design aspects of computational steering systems, such as the creation of a steering system such that when not in use has minimal performance impact on the tooled simulation [?]. More recent efforts have also focused on expanding support for steering systems used with other high performance computing libraries, such as CUDA [?, ?]. Other recent efforts include investigations of better architectural structures and better methodologies for data transmission [?].

Other Research Efforts

In addition to those discussed above, there have been a few other research efforts for furthering the development of computational steering. In [?], the idea of performance steering was implemented. Later, performance steering was likewise implemented for the RealityGRID system [?]. Along a different line, there have been research efforts specifically on logging in a computational steering environment [?]. Computational steering research has also investigated issues of debugging code via "time-travel" [?]. Other efforts have looked at mitigating high complexity simulation by first creating a 'path' in a low resolution mode [?] before performing a full simulation along that path.

2.2.5 Computational Steering System Scaling

Simulation, visualization and computational steering are highly computationally intensive tasks. Since the first computational steering experiments and systems, supercomputers and specialty workstations have been involved to perform simulation and visualization [?]. As simulations have grown in scale and complexity, their computational requirements have also grown. Additionally, there is an increasing requirement for visualization in order to comprehend higher complexity simulations, and efforts have been made to enable such visualizations [?]. This increasing computational demand is countered by technological developments that today offer enough power on the average desktop to perform simulations and visualizations at scales inconceivable even 10 years ago.

Given the increasing demands of simulations and the increasing power of desktop systems, there has been significant effort in moving computational steering to both large scale parallel platforms as well as desktops systems. Efforts to move computational steering to large scale parallel platforms include Falcon [?] and others [?], the efforts to parallelize the SCIRun Environment [?], and the development of a computational steering API for the RealityGRID grid environment [?] as well a other GRID systems [?]. Research in moving steering to the desktop have included the creation of steering environments designed to run entirely on a single system as well as environments that make use of graphics hardware found in most modern desktop to speed up the steered simulation and visualization [?]. These developments make computational steering available to more potential users of the technique. The technology and

demand are both in place for the development of a single tool that is capable of developing computational steering applications for grids, clusters, individual desktops, and all architectures in between.

2.3 Observations on the Field of Computational Steering

Considering all of the above, a number of insightful observations can be made with regard to the field of computational steering. First, the modern state of computational steering is discussed. Second, the commonalities present within computational steering systems are enumerated and discussed.

2.3.1 Modern Computational Steering

Computational steering environments and applications can now be found on a variety of platforms, from single processor workstations, to massively multiprocessor clusters, to grid based systems. Steering systems have developed to take advantage of various new hardware and software technologies and today, most research focuses on extending steering environments to take advantage of new technology or the creation of steering solutions for specific problems. However, while a wide variety of steering systems exist, there is no single, simple, hardware independent, and unified solution.

2.3.2 Commonalities and Base Structures Implicit in Computational Steering Systems and Techniques

There are a number of themes and trends found in the computational steering literature. Most notable amongst them is that, at their base, all computational steering systems are fundamentally about two things. First and foremost, steering is about the manipulation of the internal state of a running computation from outside that

computation. This computation can be a simulation, code driving a display mechanism of some sort, a user interface, or any other piece of code. Second, computational steering is about how the moving of data between computational elements to affect said changes to internal state. In many cases, this communication takes place over the network, in other cases, via exchange of files though a networked file system. In all cases, however, this communication takes place. While there are many other commonalities, such as the presence of a client interface, or the presence of a central coordination point between compute elements and the steering elements, none are nearly as universal as the two discussed above. These two commonalities were key in the decision making processes involved in the creation of a new computational steering interface.

Chapter 3: Design of a Computational Steering System for Mainstream Acceptance

Today both the required hardware and software tools exists to make a fully featured computational steering environment very successful. Further, it has been shown that steering is a very useful, powerful and efficient computational methodology [**?**]. Additionally, computational steering shows clear advantages of putting a researcher within the computational loop, where they can apply abstract domain knowledge and intuition to their simulation. Given the proven and potential benefits of steering, the expectation would be for its widespread usage. It is therefore both surprising and disappointing that steering has not gained wider acceptance in computational science.

Key to this lack of acceptance are a number of serious and common problems with existing computational steering designs. In order to resolve these problems, a solution is proposed which considers not only the problems but additionally considers commonalities found throughout the field of steering discussed in sections 1.1.4 and 2.3, as well as accepted and well tested steering system structures discussed in section 1.1.2. Also considered are historical matters of high performance computing tool history, design, and adoption specifically with regard to the MPI and OpenMP parallel computing libraries. Finally, all of the above considerations are tempered with consideration of practical matters of system design. All of these considerations

are discussed in detail below. Then, in the next chapter, an in-depth discussion of the architecture and implementation of the LAPIS system resulting from all of these considerations is presented.

3.1 Historical Trends in High Performance Computing Libraries

First considered is the parallel that can be drawn between the problems facing computational steering today and those facing parallel computing prior to the adoption of OpenMP and MPI. Like steering, early parallel programs were developed individually, as there were limited tool-sets available, usually all implementing similar behavior, but only available as a vendor specific implementation [?]. Additionally, early parallel programs were tied to the hardware on which they were developed [?]. Until standard tool-sets became available, the same parallel functionality had to be repeatedly developed by different researchers working on different projects or platforms. This resulted in a high development cost for parallel programming, and the adoption of parallelization was slower than expected. Computational steering has followed a similar development path, and its adoption has also been slower than expected. In the case of parallel programming, the creation of standard and open source libraries, such as MPI and OpenMP, has been critical to its wide spread adoption [?, ?]. Likewise, an open standard library for computational steering would aid in its adoption. Therefore, the LAPIS system has been developed around a proposed standard for steering tools. Additionally, LAPIS will be licensed as free, open source software, but will allow for third parties to commercialize the software, so long as the standards set forth are not violated.

3.2 Consideration and Leveraging of Commonalities Within Computational Steering

As observed in section 1.1.4, there are a several important commonalities found throughout the field of computational steering that seem to be nearly universal. One of the key conclusions that can be drawn from these commonalities is that all steering systems could be modeled by a peer-to-peer network of computational nodes and that such a modeling would not limit the desired architecture of a simulation. Toward that end, LAPIS has been designed to use a high level model of a fully-connected peer-to-peer network. Additionally, observation shows that there is no intrinsic reason to vary the function of specific nodes and toward this end each node within the network uses a three layer software stack. This creates greater consistency in design and structure and additionally addresses many of the problems found within steering systems (discussed in section 3.3) by creating important isolations and abstractions. Both the high level design and the node level design are discussed in detail below.

3.2.1 High Level Design

The high level structure of the LAPIS system is that of an interconnected network of separately running nodes. Specifically, the steering network (see figure 3.1) is a completely connected peer-to-peer network, meaning that every node attached to the network is aware of all other nodes attached to the network and communication between any two nodes is dependent only on the sending and receiving nodes. This view includes systems that run completely on a single system, where the medium is the systems memory, as well as systems running on grid systems, where the Internet

is the medium and all others in between. In addition to being a valid model at various system scales, this view accommodates all four classes of computational steering discussed in section 2.2.1. Additionally LAPIS supports all steering systems, regardless of whether component processes communicate directly or through an arbitrator process of some sort. From the networks perspective, all the nodes function using the same communication protocol, though communication volume from different nodes may differ. Additionally, the developer is welcome to create any artificial limitations on which nodes may communicated with each other. So, while one node may be the simulation itself, and another the steering interface for the simulation, the network handles both the same way. A high level architecture that treats all nodes identically minimizes system complexity while maximizing flexibility. If functionally different nodes were treated non-uniformly, only nodes whose functionality was understood by the network could be used in the steering system.

Given this top level model, consideration must be given to the mechanism for communication between nodes on the steering network. As discussed in section 1.1.2, the roll of sensors and actuators, while seemingly opposite, are in face entirely determined by the point of view of the code interacting with them. An actuator used by a GUI is in effect a sensor from the perspective of the steered simulation. Likewise, a sensor used by a GUI to observe a simulation acts effectively as an actuator, affecting change in behavior on the part of the end user. Toward this end, a more perspective neutral mechanism has been devised. Specifically, all nodes attached to the steering network have the ability to "GET" and "SET" variables published by other nodes. "GET" and "SET" operations are always viewed as such, regardless of perspective.

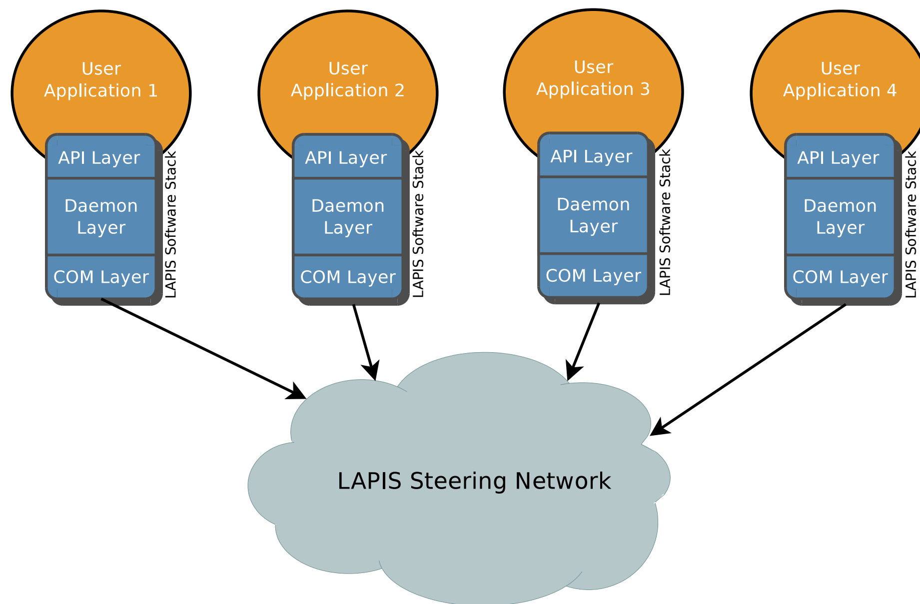

Figure 3.1: The high level architecture that will be used by the LAPIS system. Note that each application may use a different supported language and thus a different implementation of the API layer. Also, the LAPIS steering network it not explicitly implemented but rather is effectively created via a coordination of the Daemon layers, communicating though the COM layers.

When a node requests remote data, that is always a GET, weather from the perspective of the requesting node, or the node that owns the requested data. Likewise, when a node request data be sent to a remotely published variable to overwrite it's value, that is always viewed as a SET, regardless of perspective. Implementation of this mechanism ensures that the intent of an operation is always clear. Additionally, the GET and SET methods are implemented such that their execution can take place more immediately, rather than relying on the node that controls the data to respond explicitly to each request.

Another factor to consider in this design is the mechanism by which new nodes connect to the steering network and, correspondingly how nodes disconnect and how faults are handled. For simplicity of operation, the implementation of these details should be taken as much out of the hands of the simulation developer as possible. To this end, connecting to the steering network should require only the address of one node already attached to the network. From this initial connection, the new node should be able to receive all needed information about the network, and likewise, all the nodes on the network should be informed of the presence of the new node. Likewise, in order to disconnect, the end user should only have to make a simple API call, without any need to provide specific information. Disconnection and resynchronization of the network should be handled seamlessly. Finally, in the event of a node failure, the system should have built in mechanisms that automatically prune failed nodes from the system, while ensuring that all remaining nodes maintain a consistent view of the network. All of this high level functionality in the LAPIS system is as a result of uniform behavior programmed into the individual nodes.

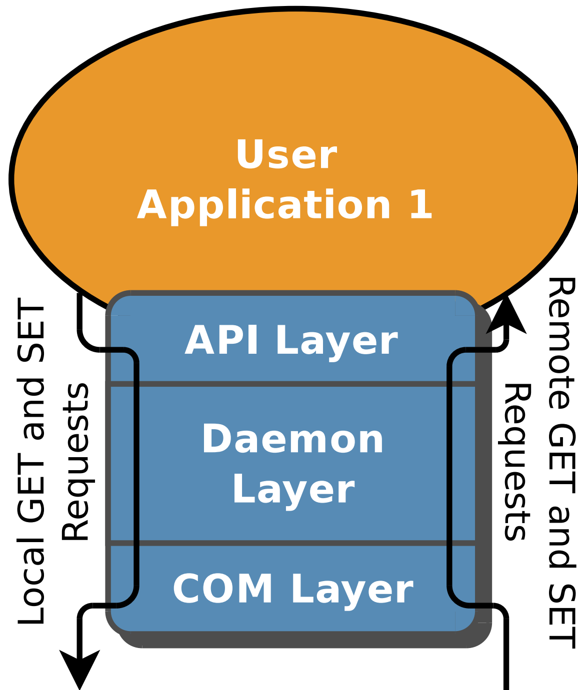

Figure 3.2: Proposed Software Stack

3.2.2 Node-Level Design

Individual nodes will use the three layer software stack shown in figure 3.2 for a wide variety of important reasons, mostly related to addressing the identified problems faced by most steering systems. Generally, the layers of abstraction and isolation provided are very valuable to addressing the known problems of many steering systems. Details of these issues and how the abstraction address them are discussed below in section 3.3. Key to the design of the software stack is that regardless of the functionality of the node, the same software stack will be used to implement steering and attach to the network. Architecture details of the top layer API, bottom layer communication interface, and middleware Daemon are each discussed in detail below.

Top Layer: The LAPIS API

The top level of the software stack focuses on isolating the users choice of development language from the implementation details of the middleware Daemon and communication layer COM. As the API Layer is the component of the LAPIS system that developers will interface with, it is imperative that it remain as constant and easy to use as possible. To that end, a central factor in the development of the API layer is the development of an API layer standard interface. Such a standard interface allows for development of multiple versions of the API for different languages while ensuring that the syntax of the API remains the simple and as constant as possible between languages. Likewise, there interface between the API layer and the middleware layer is also a defined standard, ensuring that development of APIs in a variety of languages is as straightforward as possible.

The primary goal of the API is to allow users to develop simulations and other software components that read data from and write data to other nodes on the network. In order to achieve this goal, the API implements a *Get* and *Set* method, however other functions are required before these methods are useful. The API provides methods that register data within the users code with the middleware, allowing remote nodes to read and write to it, a process referred to as "publishing". Likewise, there are methods that unregister data and control write-access to the published data. Finally, a number of methods provided are focused on creation, maintenance, and connecting to the steering network, as well as the gathering of information about the network. Through these API calls, the users code becomes a node on the steering network, publish data, and read and write to remotely published data. The base API

provides sufficient functionality to implement common computational steering tasks, such as saving and loading computational checkpoints.

Middleware Layer: The LAPIS Daemon

The middle level of the software stack is responsible for four major tasks. First, the middleware uses the communication layer (referred to as 'COM') to establish and maintain the steering network between nodes. Second, the middle layer creates a list of nodes in the network, as well as a list of published data on each node. Third, the middleware is responsible for forwarding *Get* and *Set* related messages to the API so that data transfers can be setup and executed. Finally, the middleware responds to API calls from the users code by requesting data from other nodes via the communication layer. By implementing this layer in Java, all of this complex functionality can be system independent, depending only on an appropriate Java based communication layer. Using the same middleware software regardless of platform and helps greatly to resolves issues of data compatibility between different languages that different nodes may use.

Communication Layer: The LAPIS COM

The bottom of the software stack will implement communication between nodes in the network in Java, based on a standard communication interface that will be used by the middleware. This communication interface allows the middleware layer access to read and modify data on other nodes. Additionally, the middleware responds to control messages send by the communication layer when the communication layer receives requests for data from other nodes. Using such an interface allows for different

implementations of the communication layer to be used to cover different circumstances and communication methods. This effectively removes the last dependency the middleware has to the hardware and software configuration of the underlying system.

3.3 Identifying and Addressing Problems in Existing Computational Steering Systems

A number of central problems exist in computational steering systems that have critically held back its adoption by mainstream scientific computing efforts. Development difficulty is a key problem with new techniques, particularly when presented to researchers who primarily focus is fields other than computational science. Concerns over maintainability and portability of steered simulations will keep many users away, as they value their simulation too much to put it at such risk of obsolescence. Many computational steering systems impose a structure that may not agree with the simulation developer, or may be flatly incompatible. Problems with data security and data compatibility are often too critically important or too complex for developers to overcome. Also, any technology that may negatively effect the performance characteristics of a simulation is likely to be deemed not worth the performance penalty. Finally, a lack of third party support for higher level software tools and more interesting hardware and software applications means that the field lacks some of the range and variety of other computing technologies. All of these issues are discussed in detail below.

In order to create a steering system with wider acceptance, the most important design metric should be in addressing the problems identified previously in existing computational steering systems. Issues of development difficulty can be addressed

with careful creation of a user API, and to some extent by the 3-layer software stack discussed above (see figure 3.2). Issues of portability, maintainability, and 3rd party support can all also be addressed with the 3-layer software stack. A middle layer written in a platform independent language encapsulates the majority of complex code, and links the top and bottom abstraction layers. This structure provides the user with a simple and consistent interface, while at the same time providing platform independence for the tools execution and communication. Structural, Data, and performance issues can all be addresses with careful design considerations. Each of these problems is discusses in detail below, along with a discussion of how they are resolved though the implementation of LAPIS.

3.3.1 Difficulty in Development of Computational Steering Applications

The first shortcoming slowing the widespread acceptance of steering is the difficulty of development associated with computational steering infrastructures. For a research scientist looking to use computational steering, development will take one of two forms: instrumentation of existing code [?], or creation of a new simulation in a specialty steering environment, such as with the SciRun2 environment [?]. In either of these cases the researcher must extend their own programming ability by learning either a new programming library or a new development method, as in many cases, steering is an unfamiliar technology to scientists as first noted in [?]. The average researcher using high performance computing has a very low threshold of difficulty before they deem learning a new technology to not be worth the trouble. Upon hitting any resistance at all, the probability of a researcher continuing to learn a new technology or technique drops staggeringly in many cases.

As first noted by both [?] and [?], computational steering systems available today often require not only domain knowledge relevant to the simulation, but also a non-trivial understanding of computational steering. As many researchers have already pushed themselves in learning programming techniques involving parallel programming, numerical methods, system software and I/O, adding yet another are of required expertise is often quite inhibitive. This is particularly true when the steering systems available to the researcher are complicated to setup as well as use. The difficulty in using steering systems was first noted in [?], and subsequently by [?] and [?], and this trend continues today. In fact, there have been significant research efforts made into tools specifically designed to help integrate steering tools with simulations [?].

Additional difficulties in development tie into issues discussed in sections 3.3.2, 3.3.3 and 3.3.4. Specifically, there has been a trend in computational steering research to develop domain specific steering systems. In addition to being domain specific, many systems are machine, language, or architecture dependent. Finally, different steering systems often impose different organizational structures in order to use them. All of this combined leads to a high likelihood that effort a researcher puts into learning to use a steering system will only pay limited dividends, and will only pay those dividends for a limited time frame. Additionally, the variation between different systems will make updating code to newer systems and setups non-trivial and quite possibly highly costly in terms of development effort.

As an interesting consequence of these critical variations between steering systems, a large amount of duplicate effort has been expended toward porting various steering solutions to various platforms, rather than toward advancing tool sets for computational steering. This duplication of effort is much like the early developmental of

parallel computing before the MPI and OpenMP standards. Custom parallel development systems were provided by machine manufacturers, and had little in common with each other, beyond their objective of enabling parallel computing. Much like steering, this led to a very large amount of reinvention prior to the widespread availability of the MPI and OpenMP libraries. Today, manufacturers still often provide parallel development tools, in the form of custom implementations of the MPI and OpenMP standard libraries.

Addressing Development Difficulty

Development difficulty can be addressed in two ways, first through a concerted effort to keep the tool as simple as possible while at the same time providing all the functionality needed to implement any arbitrary steering system. Provided functions should be simple, yet powerful, unambiguously named and implemented, and should require a minimum of input parameters to accomplish a given task. The second way of addressing development difficulty is to create a tool that is available for use in many languages with as little syntactical difference as possible. This has three primary benefits, first it prevents the potential user from needing to learn a specific language and port their code to make use of the steering tool. Second, existing users of the steering tool need not worry about having to relearn the usage of the tool if they choose to start using a new language. Finally, researchers working in different languages can share ideas on the usage of the steering tool without need for any syntactical or architectural translation.

The LAPIS system was developed with all of the above considerations in mind. LAPIS provides a simple API that is easy to understand, set up, initialize, and use. At the same time, the LAPIS API is complete enough in function to implement

any more complex steering mechanism or structure that may be required. API calls involve a minimum of input parameters and use a simple text based name-space for distinguishing nodes and data within nodes. The proposed LAPIS standard API interface presented in the following chapter will ensure that when new versions of the API become available, they will use nearly identical syntax, and their usage will remain consistent and simple to use.

3.3.2 Maintainability Challenges With Steered Applications

The second shortcoming in current steering applications is that of maintainability. Once a simulation has been built or instrumented with a steering tool, it becomes linked to that steering technology. As many steering tools have requirements in terms of communication mechanisms, availability of hardware and software, and system specific configuration, the steered application inherits these requirements. Changes to the operating environment that violate these requirements results in a failure of not just the steering tool, but the entire simulation. This makes the use of steering technologies difficult, particularly in dynamic environments like a supercomputer center, and even potentially dangerous to the continued development of the simulation.

In addition to the loss of flexibility that can result from the inclusion of a steering tool as part of a simulation, there must also be consideration given to the effects of the simulation development itself. Specifically, inclusion of the steering code has the potential to cause a branching in the simulation development into two distinct efforts, both with and without steering. This will likely happen in any case where inclusion of the steering tool will prevent the simulation from being run without the usage of said tool.

Increasing Maintainability

In a similar vein to portability, creating a steering tool that does not inhibit maintainability is key to a tool gaining wide-spread acceptance. Also similar to issues with portability, many steering tools fail in this regard. Inclusion of many computational steering tools permanently and unavoidably change the underlying simulation code such that is much always be run with the steering tool included. Ideally, a steering tool should be designed such that inclusion of the tool will not require that the tool be used, does not require the user to change update code when the tool itself is updated, and requires the simulation developer to change their simulation code as only minimally.

A 3-layer software stack again helps us meet these design goals. A careful design strategy with regard to the API layer will minimize the required insertion of code into user simulations, and no design compromises need be given to the other two layers. Likewise, additional careful design in the API layer will effectively allow steering implementations that are effectively deactivated when steering is not in use. More complex steering implementations can similarly be designed and implemented with negligible impact, but design details of such system will vary widely. Finally, so long as standard interfaces between the layers are strictly observed, changes to the Daemon or communicator layer should not effect simulations that use the API layer in any way.

Through the use of the three-layer software stack, the LAPIS system maintains a high percentage of the users code original maintainability. Though careful usage and design, simulations can be developed where use of the steering tool is not implicitly required. LAPIS may, however, be explicitly required based on how the user chooses

to use the software tool. That is, if the simulation is dependent on the internal state of the front end, or expects its own internal state to be set by some front end, then it will obviously not function without that front end. This, however, is not a limitation imposed by LAPIS, but rather by the usages of LAPIS by a simulation developer. However, with minimal consideration it is possible to use LAPIS within an existing simulation in order to make it steerable without effecting the simulations ability to run without any steering agent being present.

3.3.3 Portability Issues with Steered Applications

The third shortcoming steering systems currently face is that of portability. While viable steering systems are available for a wide range of hardware, most support only a handful of specific platforms. Given the variety of hardware used in scientific computing, a lack of cross platform compatibility is a critical issue. For any given system, a steering tool must be compatible with not just the systems processor architecture, but also the systems general setup including network hardware, file system structure, batch processing system, and so on. For example, some steering tools require direct socket connections between components [**?**], while others rely on a shared memory architecture [**?**], and still others make use of the message passing interface (MPI) for communication [**?**]. In all these examples, and indeed most steering systems [**?**], the communication method is fixed. Depending on the system, any of these communication methods may or may not be possible. Additionally, should the system configuration change, for example, for security reasons, and a given mechanism become unavailable, any steering system dependent upon that mechanism, and thus

any simulations made with the steering tool, will break. In addition to communication mechanism, there are potentially countless other dependencies a steering system might rely on for proper function. In fact, many systems are tied to a specific setup [?]. Use of such steering tools ties the developer not just to the use of the tool, but to all the dependencies of that tool.

Ensuring Portability

In order better encourage the adoption of a computational steering system by the scientific computing community at large, the steering system in question must be designed with portability in mind. Many existing steering systems fail in this regard, either requiring platform specific compilation, specific hardware, specific operating system, or specific programing language in order to function. Portability is increased when the steering tool can be run on any hardware platform ("hardware agnostic"), on any operating system ("OS agnostic"), and on any communication medium ("communication agnostic") [?]. The software stack presented above in figure 3.2 offers a significant and novel approach to increasing portability, and successfully increases portability in all the respects. Additionally, the software stack presented allows for changes in system behavior via very small configuration changes at run-time.

The API level allows an interface to the core functionality of the tool to be developed in whatever language the simulation developer cares to use, without requiring reimplementation of the whole steering system. The Daemon layer is implemented in a platform independent language (in the case of LAPIS, JAVA was used), therefore most of the core functionality of the steering tool does not require reimplementation in order to support new languages. Finally, the communication layer, also implemented in a platform independent language (again, for LAPIS, this is done in JAVA),

abstracts details of inter-node communication. In this way, a move between systems that use different communication mechanisms only requires a run-time swap of communication layers (or 'COMs'). In a similar fashion, by changing from a communicator that runs entirely within the local system to a distributed communicator, the manner and scale in which the simulation is run can be changed with very little effort.

The LAPIS system isolates as much of the complex functionality as possible within platform independent code bodies, primarily the Daemon, and secondarily the COM module. This ensures that these two layers of the system can run on just about any platform in just about any circumstances. As an end goal of the LAPIS system is to have a wide range of available APIs, and the language of the API chosen will match the language of the code in which it is used, LAPIS will add no constraints to portability beyond that which the users choice of programming language has already added.

3.3.4 Structural Constraints Imposed By Steering Tools

Another potentially serious issue with computational steering tools is one of code organization, structure, and architecture. Many systems distinguish between elements of the steered simulation, such as distinguishing between client and server portions of code [?]. Such distinctions can add arbitrary structure to the simulations made with the steering tool that would not otherwise be present [?]. In order to use such steering tools, the developer is forced into whatever arbitrary structure the steering tool uses, and often this arbitrary structure can make modification of code for steering quite difficult [?] and may lead to design compromises that result in performance loss. Along

similar lines, the systems upon which computational steering simulations are run may impose their own restrictions, such as batch processing systems. Such systems can make starting and steering a simulation more complicated, and possibly impossible in some cases [?, ?, ?].

Easing Structural Constraints

Throughout the literature available on computational steering, there are numerous examples of steering tools that impose an architectural constraint upon any simulation that makes use of the tool. For example, some steering tools utilize a central data store, against which all the other components of the system synchronize. Such a structure may well be useful in many steered simulation systems, however it can be said with absolute certainty that not all simulations would benefit from such an organization. Thus, it is critical that any steering system we design use an organizational structure that is flexible enough to emulate any needed organization, without incurring a significant performance penalty. A peer-to-peer network as suggested in section 3.2 can be used to model any such structure simply by ignoring certain connections in favor of others. The LAPIS system is designed around such a peer-to-peer network structure and thus avoids imposing any sort of artificial architectural or organizational constraint upon systems that make use of it. Additionally, by making no differentiation between nodes withing the steering network based on function, role, position, or anything else, it additionally imposes no restrictions upon the type, distribution, and demographics of nodes within the steering network.

3.3.5 Computational Steering and Data Issues

Computational steering systems can also cause issues and concerns with regard to simulation and steering data. Most obviously, if different components of a computational steering system are written in different languages, or even are running on different hardware platforms, there may be fundamental differences between data types, such as bit order, bit length, precision, and so on [?]. With traditional simulation, such issues rarely, if ever, arise, with the possible exception of input data sets. With multiple processes, possibly running on multiple machines, such compatibility issues become much more likely. Additionally, with steered parallel simulations, consistently addressing objects that span memory spaces can pose a significant problem [?]. Finally, with internal simulation state being transfered between different processes, an issue may arise with regard to the security of that same data [?].

Managing Data Issues

First, steering systems are often in the interesting position of having different components constructed in different languages, using different tools, systems, and techniques. As such, the data format for a particular type of data my vary between elements of the steered system. By using a three layer software stack as discussed, data format can be normalized by the API layer so that all internal data transfers occur in a single agreed upon format, and that data formats at each end of communication are the native formats of the APIs involved, and thus, the component code as well. In addition to formatting issues, there are some applications for which data security is also of key importance, and a steering system used in those scenarios must provide some mechanism of security for transfers of data between nodes of the system.

Within the LAPIS system, both formatting an security issues are addressed. First, data formatting is unified by the API layer, and all data transfers below the API and between nodes takes place in a unified format, specifically, using data formatting compatible with the JAVA language. When a new API is written for a new language, any differences in data representation must be handled within that new API, and incoming data must be translated to the local languages representation, if needed. Likewise, data sent out form the API layer must be translated into the JAVA representation. Data security, on the other hand, is handled though implementation of a COM layer module that implements whatever inter-node security measures are required.

3.3.6 Performance Issues with Computational Steering Systems

Still another issue that must be considered when looking at the use of steering systems is that of performance and stability. It has been noted in numerous fields that a centralized design creates an obvious single point of failure, and computational steering is no exception [?]. As such, any computational steering system that requires a centralized point of control is introducing potentially serious stability concerns. Additionally, even loss of a periphery component of a computational steering application could result in the total failure of the entire simulation. It has also been noted that in some computational steering tools, required steering function calls must be included within the main computational loop, and as such represent a measurable performance loss [?].

Minimizing Performance Impact

Performance overhead in steering tools is of critical importance as the primary target for the use of steering tools is that of high performance, high complexity simulations and computations. If inclusion of a steering tool into the simulation code causes a significant degradation when in use, or even worse, at all times, any potential benefit of efficiency or performance gained from the involvement of a human operator may quickly be lost. This is especially true for long running simulations that only occasionally require human interaction. In the case of LAPIS, the system has been designed such that when not in use, there is no processor overhead (or nearly none, in the case of MATLAB) for the inclusion of LAPIS in a code. Both the Daemon and COM layer of the LAPIS system are designed to only require CPU time when the system is not in a steady-state. If no node on the LAPIS steering system is performing actions, then no cycles are being devoted to the steering tool.

3.3.7 Supporting Software and Hardware by Steering Tools & Environments

The final shortcoming in current steering applications is a lack of support for a wide variety of languages, hardware, and third party software tools. Scientific simulations use a wide range of programming languages, specialty hardware, and specialty software libraries and tools. In order for researchers to consider steering an an option, steering infrastructures must support these languages and tools. Unfortunately this is usually not the case, with steering systems supporting only one or a few languages each. Additionally, while many of the steering tools available support some of the various specialty software libraries and tools, few steering tools support a wide range.

Providing an Easy Target for Third Party Development

If independent third party software and hardware can be developed for the steering system, it will help greatly in the steering tool garnering wide spread acceptance. The three level software stack, along with the API and COM standard make such development by third parties significantly easier. Either the third party can develop their tool to make use of the API layer calls, or they can implement their tool in place of the API layer itself, so long as the tools function meets the standard set for the APIs function. Along with greater ease of development of third party software tools, third party hardware can, likewise, be built to rely upon software drivers that operate on top of or as the API layer of the software stack.

The LAPIS standard API interface provides sufficient function for the development significantly more complex and powerful tools than the methods within the API itself. Additionally, it provides this with the guarantee that should underlying details of the system change, the API interface will not. This assurance allows third party developers the confidence needed to invest time and effort into the creation of new and more powerful, varied, and application specific tools that has not previously existed in the field of computational steering.

3.4 Practical Design Considerations

The final area of consideration in the design of the LAPIS system is the practical implications of all of the design choices above. All of the topics discussed below are more or less the necessary result of choices made above, rather than choices unto themselves. There are three specific topics that should be understood. First, the choice of development language for the Daemon and COM layers, the division of

functionality between the API, Daemon, and COM, and the programmatic structures that must be present within a language in order to implement a version of the LAPIS API.

First, given the need for as much of the system as possible to be inherently multi-platform with as little reimplementation required as possible, there were two language choices for implementing the Daemon and the COM layers that were seriously considered, Java and Python. While both languages offer much the same in the way of available libraries, function and cross-platform operation, Java is still far more widely available in practice. Most every Unix-based operating system has an available JVM, as does Windows, and in general, they are easily obtainable and installable.

Second, the need to minimize the complexity of both using the steering system to implement steered simulations, as well as implementing new components for the steering system itself, results in the most complex and significant functionality of the system being moved into the Daemon layer. This is also done to minimize the need for reimplementation of critical components and processes. In the end, the implementation of a COM only slightly more sophisticated than simply forwarding messages between the steering network and the Daemon layer. The API, while slightly more complex than the COM, is still very reasonable in terms of difficulty of implementation, and should be approachable by a developer of moderate experience. Finally, design of the API itself has focused on minimizing the number of parameters for each method in order to simplify it's usage.

Finally, all of the above design choices and requirements result in certain requirements of a language before the LAPIS API can be implemented. First and foremost, as all communication between layers occurs via local sockets, the target language

must have a compatible sockets library. Additionally critical is the ability to address data via reference, or achieve the equivalent functionality. Specifically, two threads in the same memory space must be able to access the exact same data object. The chosen language must also support threading in some fashion, with the developer being able to control the function of both threads, and both threads must have access to a common memory space. This requirement can be worked around in some cases, for instance the MATLAB API makes use of the timer object in MATLAB rather than a true thread. Finally, the language must support some sort of blocking mechanism, or at worst a sleep mechanism combined with a polling loop (again, see the MATLAB version of the API for an example of such a work-around).

Chapter 4: The Development and Use of the LAPIS System

Working with the design choices discussed in chapter 3, the various components of the LAPIS system were developed. Prior to discussion of any specific implementation details, a brief discussion of the high level functionality is presented. Implementation details of the LAPIS Daemon, the first TCP/IP LAPIS COM layer, and the LAPIS Java and MATLAB API layers all follow, and are discussed below in detail. Additionally, the details of developing steered simulations and steering interfaces with the LAPIS system are discussed. Also discussed are the details of developing new components (APIs and COMs, most specifically) that interface with the LAPIS system. Finally a summary of the abilities and features of the LAPIS system, in the context of the problems discussed previously, is presented.

4.1 Structure and Behavior of the Overall System

The high level behavior of the LAPIS system and steering network should be discussed prior to any discussion of the actual software components that implement the behavior. The LAPIS system architecture, described previously in chapter 3, is that of a peer-to-peer network of nodes. Each node is a three level software stack comprised of a user simulation making use of the LAPIS API, the middleware Daemon,

and the COM layer. As discussed, there is no explicit implementation of the steering network. Rather, the behavior that establishes the peer-to-peer steering network is emergent behavior as the result of interaction between the Daemon layers of the nodes involved in the steering network. However, in order to understand some of the implementation decisions in the node layers, one must understand the process that is used to create the peer-to-peer network as well as keep it consistent.

As a fully interconnected network of nodes, ensuring each node has a consistent and complete view of the steering network's structure is critical to proper operation of the LAPIS system. A complete and consistent view is achieved by means of an update mechanism that takes place whenever a nodes is added or removed from the steering network. The process starts when a new node, (A), sends an update message to a known member of the steering network, (B). This update message includes all nodes and data elements known to (A). When a node receives such an update message, it follows the following rules:

1. Compare the incoming list of nodes and data to the local list of nodes data and create a list of differences

2. Add the nodes and data on the list of differences to the local list of nodes.

3. If there are any nodes or data on the list of differences, send an update message to that node that includes the updated local list of nodes and data

4. If there are no nodes on the list of differences, take no further action

The following series of figures (4.1 - 4.5) illustrate the process by which this update routine takes place in order to ensure a consistent network view amongst all members of the steering network.

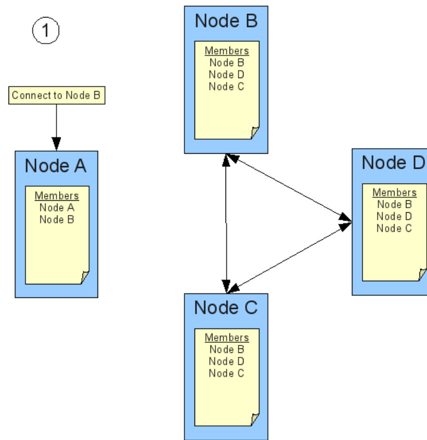

Figure 4.1: Step 1: A new node, (A), starts and connects to the network via a known member of the network, (B).

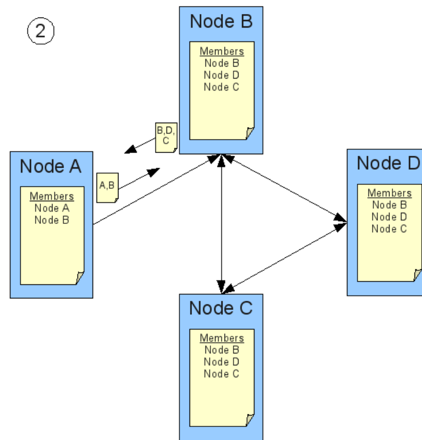

Figure 4.2: Step 2: (A) sends a complete copy of its node list to (B), (B) sends a complete copy of its node list to (A)

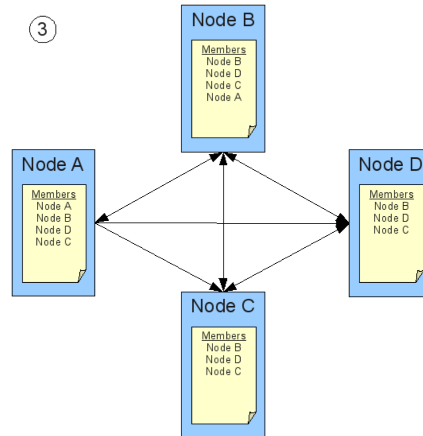

Figure 4.3: Step 3: (A) compares the list received from (B), and notes all new nodes. (B) likewise compares list of received nodes from (A) and notes all new nodes

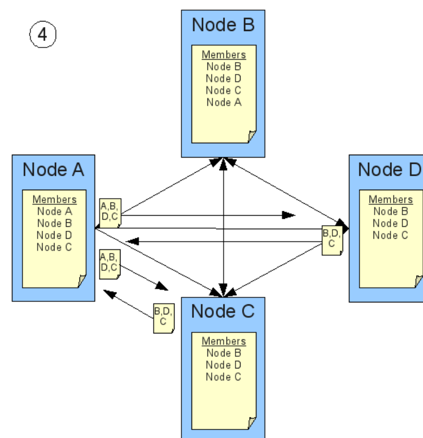

Figure 4.4: Step 4: For all new nodes detected, both (A) and (B) send a new update message, including a list of all known nodes. Each of these nodes, (C) and (D), return a list of known nodes to (A).

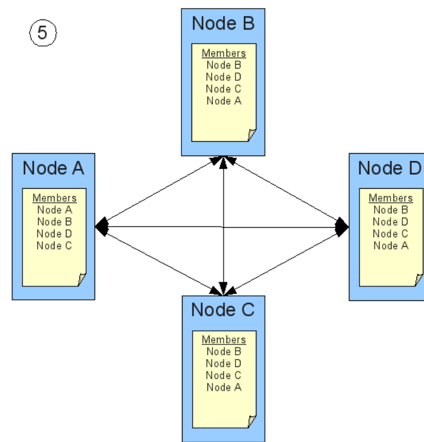

Figure 4.5: Step 5: All nodes compare received lists to their internal list of nodes. If, at this point, no new nodes are found, the update process ends.

4.2 Development of the LAPIS Software Layers

The development of LAPIS was initially focused in three main areas: the Daemon, the Java API and the TCP/IP Communicator. After developing these components, a MATLAB version of the API was developed as well. The details of the development work and implementation of each of components is covered in detail in this section. The first layer of the LAPIS software stack developed was the middleware layer, also referred to as the LAPIS Daemon. Along with the development of the Daemon was the development of a standard list of LAPIS Control Messages. The Daemon connects the API, and thus the user's code, to the COM, and thus to the steering network. Given its central nature, and the fact that the initial TCP/IP COM and API were also going to be written in Java, the Daemon was the logical place to begin. This way, various classes developed for the Daemon would be useful in the development of both the TCP/IP COM and the API. Development of the TCP/IP COM started when further progress of the Daemon was inhibited by an inability to test interoperability between Daemon layers. Likewise, development of the Java API was started when testing of features in the Daemon became impractical to perform without the API. Out of necessity, much of the development of all three components was done in parallel. In the following section, however, discussion of the implementations of these components is limited to one component at a time. Finally, after development of the middleware layer, the TCP/IP COM layer, and the Java API, a Matlab API was developed as well to allow for inter-language testing. The development process and specific implementation details of each component are discussed in full detail below.

4.2.1 A Summary of LAPIS Control Messages

Interactions between the layers of the LAPIS system, as well as in between nodes on the steering network, are all performed via a standard set of LAPIS control messages. Before further discussing the API, Daemon, or COM implementations or standards, a summary of LAPIS control messages is required. The table below lists all the LAPIS message names, along with their use. A full specification of each message, along with a descriptions of the fields that make up the message is given later in this chapter.

4.2.2 The LAPIS Daemon Layer Implementation

Development of the LAPIS Daemon started with the development of the overall structure of the Daemon. The overall architecture was laid out (see figure 4.6) and from this a list was created of the data classes and functional classes needed. Once these and various supporting classes were developed, the LAPIS Control Messages needed for fully implementing communication between the API Daemon, and COM were fully specified (see table 4.1, above). Finally, with both of these in place, the entire functional unit of the Daemon was created and tested to ensure desired functionality.

The Listener Object

The API and the COM both produce LAPIS Control Messages to which the Daemon must respond. These messages can arrive at any time and without any pre-knowledge on the part of the Daemon. The Listener object runs in a separate thread of execution, so at not to interfere with the operation of the other listener or the worker (see figure 4.7), and monitors a local socket acting as a producer in a

Data Control Messages	
PUBLISH	Signals newly published data object
LOCK	Signals a change in read/write status of a data object
REDACT	Removes a data object from the steering network
ERROR	Used for sending error messages
GET & SET Control Messages	
GET	Sent by the API to the Daemon layer to start a GET process
GET_SETUP	Sent by the Daemon to the COM, continues setup of GET process
GET_BEGIN	Sent by the Daemon to the API, starts data transfer of GET process
GET_DONE	Sent by the API to the Daemon, starts clean up of GET process
GET_FINISH	Sent by the Daemon to the COM, completes cleanup of GET process
SET	Sent by the API to the Daemon layer to start a SET process
SET_SETUP	Sent by the Daemon to the COM, continues setup of SET process
SET_BEGIN	Sent by the Daemon to the API, starts data transfer of SET process
SET_DONE	Sent by the API to the Daemon, starts clean up of SET process
SET_FINISH	Sent by the Daemon to the COM, completes cleanup of SET process
Network Maintenance Control Messages	
INITIALIZE	Initializes a new steering network
CONNECT	Connect to a node in the steering network
DISCONNECT	Disconnect from the steering network
UPDATE	Message generated when Connect or Disconnect are used, or anytime an UPDATE message contains node or data lists different from a nodes internal state
CLOSE	Shuts down the LAPIS system entirely. Generates a DISCONNECT message if currently connected
Information Control Messages	
NODE_LIST	Requests or contains a list of all nodes in the steering network
ELEMENT_LIST	Requests or contains a list of all data in the steering network
DEBUG	Used for sending debugging messages
MESSAGE	Used for sending system messages
MISC	Used for any other miscellaneous messages

Table 4.1: A summary of LAPIS control messages and their usage. Details of message format and the LAPIS Command Message Standard can be found later in this chapter in sections 4.4.

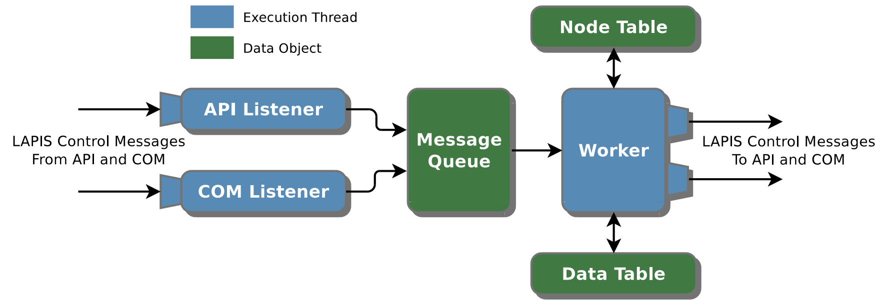

Figure 4.6: Architectural structure of the LAPIS Daemon. The overall structure of the LAPIS Daemon follows the producer/consumer design pattern, where the producers are the Listeners and the consumer is the Worker. LAPIS Steering Network structure is stored via the Node and Data tables.

producer-consumer design pattern the Daemon implements. In the case of the API listener, the other end of this socket connects to the API, and for the COM listener, to the COM. Both listeners I/O block on their socket when incoming data is not available. As soon as data becomes available on that socket the Listener reads the data in, and attempts to parse a LAPIS Control Message. If a valid message is found, the Listener adds the message to the end of the Message queue, then resumes listening to the socket.

Control Message Objects and the Message Queue

In between layers of the LAPIS system LAPIS Control Messages are exchanged via sockets as character strings. Withing the Daemon layer, LAPIS Control Objects are represented and exchanged as a Java Control Message object. When listeners parse incoming data form the sockets they monitor, they produce control message objects. The Control Message object class has a variety of utility methods. These include a method for parsing a string into a control message object, another for writing a control message object to a socket, another for converting a control message object

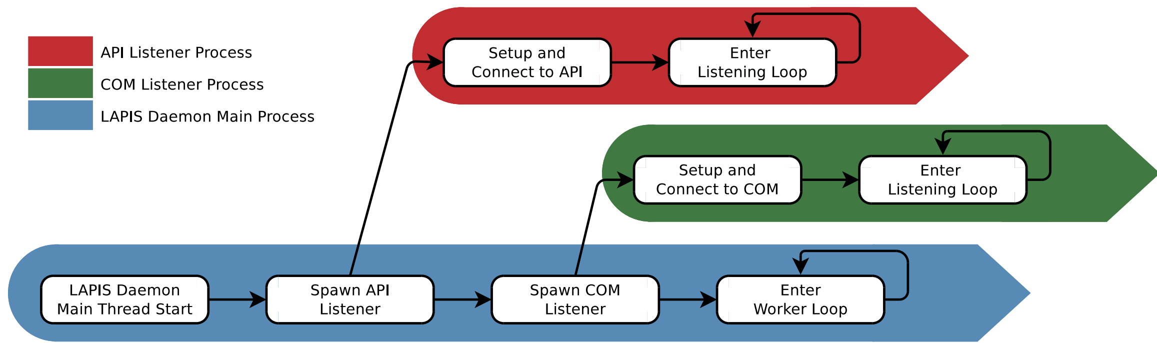

Figure 4.7: Organization of computational threads involved in the execution of the LAPIS Daemon. The main execution thread first starts the API and COM Listener threads before entering the Worker routine.

into a string, and finally a collection of methods that allow retrieving fields of the control message object for use in other processing tasks.

The Message Queue object keeps a collection of all control messages that the listeners have parsed. The message queue provides methods for adding and removing control messages, but beyond that, it does no processing of its own. Rather, the Worker thread (discussed below in section 4.2.2) uses the message queue as a source for LAPIS Control Messages upon which it must act.

The Node & Data Objects and Tables

Information about the steering network is stored within the Daemon as collections of objects, one for each node and published data object. The Node Token object contain all the relevant information about a particular node in the steering network. These tokens are stored within the Node Table object. The Node Table provides methods for translation from node name to node address, and vice versa. The Node Table is also used in generation of the update messages and node list messages that the Daemon produces.

Likewise, the Data Token objects contain all relevant information about a published data object on the steering network, including the relevant Node Token of the publisher. Like the Node Tokens, Data Tokens are stored as a collection in the Data Table object. As with the Node Table, the Data Table is used whenever the Daemon produces an update message or a data list message. Likewise, when information about remote data is needed by the node, the Data Table provides that information.

The Worker

The main workhorse of the Daemon and by far the most complex body of code is the Worker process. The Worker reads LAPIS Control Messages from the Message Queue and processes each one sequentially, performing a wide variety of tasks. LAPIS API calls result in LAPIS Control Messages being sent to the API Listener, which in turn adds the messages to the Message Queue. These messages are processed by the Worker and when appropriate, forwarded to the COM for distribution to the appropriate end points in the Steering Network. Likewise, messages from other Daemons on the Steering Network are received and processed by the Daemon and, when necessary, forwarded to the API layer.

The Worker plays a central roll in the central layer of the LAPIS Software stack. As such the Worker is responsible for a large number of functions. The Worker responds directly to node and data list requests sent from the API layer. Setting up and tear down of the connection frameworks needed to perform GET and SET operations is coordinated by the Worker. The Worker handles the generation and processing of UPDATE messages to ensure that the steering network remains consistent and

operational. Finally, the Worker handles distribution of PUBLISH, REDACT, CONNECT and DISCONNECT messages to the broader network, again to help maintain consistency between nodes on the Steering Network.

The implementation of the Worker is a single while-loop that executes until instructed (by the API sending a CLOSE message) to terminate. At each iteration of the loop, the Daemon removes a LAPIS Control Message from the Message Queue, acting as a consumer in the producer-consumer design pattern. After removing the message, the Worker enters the appropriate section of the worker body to handle that message, executes the handling routine, and then returns to the top of the loop. When no message is available on the Message Queue, the Worker blocks until a message becomes available.

4.2.3 The LAPIS API Layer Implementation, Java Version

In development of the Java version of the LAPIS API, first the functional requirements of the API were defined. Next, the structure of the API was laid out and the classes needed to develop the API were either borrowed from the development of the Daemon, or developed themselves. This entire process mirrored the process used in creating the Daemon. Once implemented, the API was tested along with the Daemon to ensure that the interaction between these components worked as expected. Subsequent of completion of the Java version of the API, a MATLAB version of the API was implemented (see section 4.2.4). Despite some significant limitations in MATLAB environment, the MATLAB versions of the API functions nearly identically to the Java version.

Unlike traditional APIs, the LAPIS steering API implements more than just the methods it provides to the end user. In order to respond to requests from other members of the LAPIS Steering Network, the LAPIS API must first be initialized, and as a part of that initialization, a Monitor thread must be created (see figure 4.8). The implementation of the API and monitor thread are each covered in full detail below.

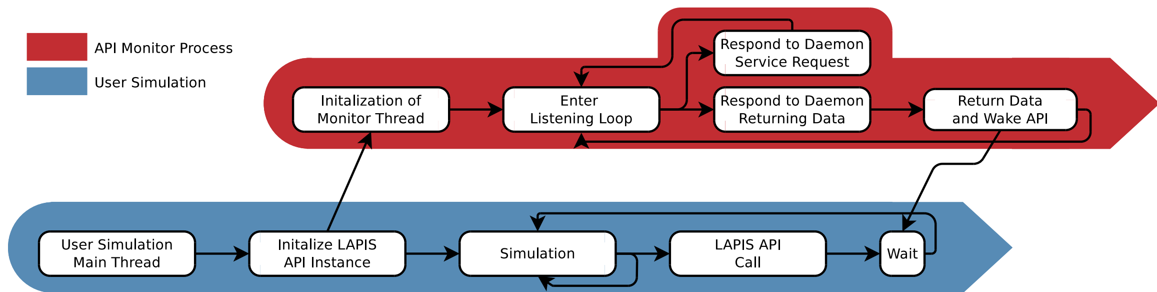

Figure 4.8: Roll of the Monitor thread in relation to the main user simulation thread. Nominally, while core simulation code is running, the Monitor thread is I/O blocking, using no resources. This only changes for two reasons. First, if the simulation leaves the computational kernel and calls a LAPIS API method, it will block on the method until the Monitor thread receives a responds and wakes the main thread. Second, when the monitor thread receives a request from the Daemon, the Monitor thread will process this request before resuming its I/O blocked state.

The API Monitor Thread Implementation

The Monitor thread serves two purposes. First, the Monitor thread returns responses from the Daemon related to to certain API calls made by the user, specifically requests for node lists and data lists. The user simulation generates a NODE_LIST or ELEMENT_LIST message and sends this message to the Daemon and then blocks, waiting for a response from the Daemon. When the monitor thread receives a

NODE_LIST or ELEMENT_LIST message from the Daemon, it wakes the blocked API call, which then returns the list to the user code. The second roll of the monitor thread is to provide a mechanism for GET and SET requests to be serviced without any interaction from the user simulation thread and without any development requirement by the end user of the LAPIS API. When GET and SET related control messages are received by the monitor thread it handles the setup of socket connections to the COM and transmission of data from the user simulation to the socket.

The Monitor thread itself is implemented as a while loop that runs until the LAPIS System is shutdown, much the same as the Worker thread within the Daemon. Within the loop, the Monitor I/O blocks on a local socket connected to the LAPIS Daemon. When messages from the Daemon come in on this socket, they are parsed by the Monitor thread and immediately acted upon. Depending on the type of message the Monitor thread receives, different actions are required by the LAPIS API standard (discussed in section 4.4.2). The Monitor thread takes whatever actions are required before resuming its I/O block on the socket.

The API Method Implementations

The implementations of the LAPIS API methods are done primarily via the creation of LAPIS Control Messages that are immediately forwarded to the LAPIS Daemon. Specifics of what Control Messages must be sent for specific API calls is covered thoroughly below in section 4.4.2. In most cases, the method simply requires a message be formed and sent to the Daemon. However in some cases thread of execution calling the method blocks, as discussed above, until the Monitor thread receives a response form the Daemon and wakes the thread.

4.2.4 The LAPIS API Layer Implementation, MATLAB Version

The structure of the MATLAB version of the LAPIS API is very similar to that of the Java version, with a few subtle but key differences. First and foremost, MATLAB does not support explicit, user controllable threads. As such, the Monitor Thread of the Java implementation is replaced with a MATLAB Timer object whose handle function has similar functionality. Rather than continuously run and block on I/O, the timer polls the socket connection to the Daemon intermittently, at a user-settable rate. While this is a less ideal solution than a I/O block, it is still relatively low overhead and provides the same functionality with only slightly more latency.

Secondly, and more critically, MATLAB support for multitasking is minimal and insufficient to allow for tasks to block on I/O. In any case where an I/O block was used in the Java version, a polling loop with a system wait call has been substituted. While this mechanism is far less ideal than I/O blocking in terms of efficiency and response rate, it is functional and still allows for implementation of a full featured and complete version of the API. Aside from these differences, implementation, development and usage details of the MATLAB API are the same as that of the Java version.

4.2.5 The LAPIS COM Layer Implementation

For the first implementation of the communicator layer, a design based on simple TCP/IP-sockets was chosen. A TCP/IP-socket based design provides the most widely applicable solution, if not the technically best solution in many cases. The COM was developed in the same manner as the API and the Daemon, starting with a definition of the COMs operation, followed by the development of classes, and finally

development of the functional whole using those classes. The organization of the software components that make up the COM can be seen in figure 4.9. Of particular note in this diagram is the Data transfer thread class used by the COM. Data transfer threads are lightweight and are spawned and exist only for the duration of a data transfer. Data transfer threads allow the COM to continue responding to requests even during heavy and sustained data transfers.

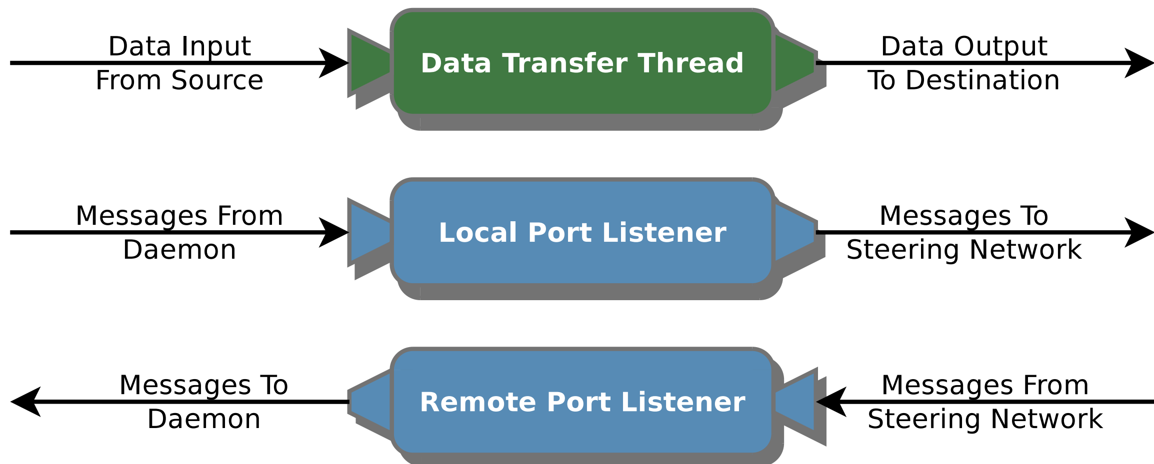

Figure 4.9: The TCP/IP Com structure: A Copy of the Data Transfer Thread Object is spawned every time that a GET or SET command is issued and exists for only the duration of the data transfer.

The TCP/IP COM implementation consists of a main execution thread that creates the Local Port Listener thread and the Remote Port Listener thread before exiting. During initialization, the main execution thread uses specified settings found in the LAPIS.properties file (see section 4.3 for more information on the LAPIS.properties file) to determine if the TCP/IP instance should be created on the systems external or internal interface.

The Local Port Listener and Remote Port Listener are implemented with nearly identical structure, however their response to incoming messages is quite different. For specifics of the COM standard and the way in which LAPIS Control Messages must be handled by the COM, see section 4.4.3. Beyond the specifics of control message handling, both listeners operate in a manner similar to the listeners used in the Daemon. Both attach too a listening port and both attach to a writing port. For the Local Port Listener, the listener reads data from a socket connected to the Daemon and writes data to a socket connected to the LAPIS Steering Network. For the Remote Port Listener, the arrangement is inverted. When no data is present on the listening port, the listener is I/O blocked and creates no computational overhead.

When the COM needs to set up a data transfer between its own node and another node on the steering network, the listener creates an instance of the Data Transfer Thread instantiated with directionality appropriate for the situation. For instance, in the case of a GET initialized by the listeners own node, the Local Port Listener will set up and start a data transfer thread that reads data from a socket connected to another node on the LAPIS Steering network's COM and and writes that data to a local socket connected to the local copy of the API.

4.3 Developing With the LAPIS System

The goal of the LAPIS system is to provide simulation developers a unified, simple, yet powerful set of programming tools for the implementation of computational steering systems and simulations. LAPIS uses a shared data model for the implementation of steering. Each component of a steered system simply 'publishes' data that it wishes to share, making the data available to all other components on the steering

network for reading and (optionally) writing. Key to the operation of LAPIS is that nodes that publish data need take no other action in order to make the data available. No priori knowledge about the timing, number, or nature of data reads and writes are needed as all subsequent details are handled by the LAPIS system.

4.3.1 Prerequisites of Use

In order to program with the LAPIS system successfully, there are 4 components that are required: the LAPIS API library in the language appropriate format, the Daemon JAR file, a COM JAR file, and the LAPIS.properties file. The LAPIS.properties file should be in the running directory of the code that uses the LAPIS API. The location of the Daemon and COM JAR files are specified within the LAPIS.properties file, along with a number of other user configurable settings. There is no additional setup required in order to run code that includes the LAPIS API.

4.3.2 The LAPIS API Overview

The LAPIS API provides a minimal functional set of methods for the application developer. The table below summarizes names of these methods. Each method is discussed in detail below, including a small example piece of pseudo-code.

The LAPIS Constructor

Prior to using any other functions of the LAPIS system, an instance of the LAPIS API must be created. For object oriented languages, this will be done through an object constructor. For non-object oriented languages, this will be done with a function that spawns another thread of execution and creates some globally accessible data objects. The constructor requires only a single parameter, the string name the

[constructor]	publish
initialize	redact
connect	lock
disconnect	unlock
nodes	get
elements	set
close	

Table 4.2: The standard methods the LAPIS API provides, as required by the LAPIS API standard (see section 4.4.2 for more detail on the standard.

node should be identified with on the steering network. The following pseudo-code example illustrates the use of the constructor. The code creates an instance of the LAPIS API in an object named 'lapis'.

```
main()
{ // The following line is the LAPIS constructor
  LapisApi lapis = new LapisApi('ThisNode')
  lapis.Close()}
```

Figure 4.10: Pseudo-code example of use of the Constructor

The Initialize Method

After creating a copy of the LAPIS API, if no steering network has yet been established by other nodes, the Initialize method is used to start a new steering network. The method takes no parameters and returns nothing. The following code shows the use of the Initialize routine.

```
main()
{ // The following line is the LAPIS constructor
  LapisApi lapis = new LapisApi('ThisNode')
  // create a new steering network
  lapis.Initialize()
  // close the API
  lapis.Close()}
```

Figure 4.11: Pseudo-code example of use of the Initialize

The Connect Method

The Connect method is used by a copy of the API in order to connect the node to the steering network. Connecting to the network with this method automatically results in the other layers of the LAPIS system going through a synchronization routine. The Connect method takes two string variables as arguments. The value of the first string is the name of the node to connect to. The second string is the address of the node to connect to. The code snippet below shows an example of use.

```
main()
{ // The following line is the LAPIS constructor
  LapisApi lapis = new LapisApi('ThisNode')
  // connects to the node named "OtherNode"
  lapis.Connect("OtherNode", "127.0.0.1:12345")
  // close the API
  lapis.Close()}
```

Figure 4.12: Pseudo-code example of use of the Connect

The Disconnect Method

The Disconnect method disconnects the node from the steering network. It takes no parameters and returns nothing. The code snippet below shows the method in use.

```
main()
{ // The following line is the LAPIS constructor
  LapisApi lapis = new LapisApi('ThisNode')
  // connects to the node named "OtherNode"
  lapis.Connect("OtherNode", "127.0.0.1:12345")
  // disconnects to the steering network
  lapis.Disconnect()
  // close the API
  lapis.Close()}
```

Figure 4.13: Pseudo-code example of use of the Disconnect

The Nodes Method

The Nodes method is used to get a listing of all the nodes attached to the steering network. The method takes no parameters and returns a comma separated list of node names, as a string. The following code sample shows the use of the Nodes method.

The Elements Method

The Elements method is much like the Nodes method, but returns a comma separated list of data-node pairs in the form of [data name]@[node name].

```
main()
{ // The following line is the LAPIS constructor
  LapisApi lapis = new LapisApi('ThisNode')
  // connects to the node named "OtherNode"
  lapis.Connect("OtherNode", "127.0.0.1:12345")
  // disconnects to the steering network
  String nodeList = lapis.Nodes()
  // parse the string
  String[] tokens = tokenize(nodeList, ',');
  ...
```

Figure 4.14: Pseudo-code example of use of the Nodes

The Close Method

The Close method is called to close the instance of the API and clean up all the layers and various threads and components that make up the LAPIS system. If the node is not disconnected when this method is called, the API first disconnects from the steering network. This method takes no parameters and returns no value. After the Close method is called, the LAPIS API object should not be used again. The following code shows the use of the Close method.

The Publish Method

The Publish method is used to make a variable available on the steering network. Scalars, vectors, 2D and 3D arrays of Integer, Double, or Character types can be published. The method takes a string name that the variable will be published, a reference to the object itself, and a boolean value that indicates if the variable is read-only. The code below shows the use of the Publish method.

```
main()
{ // The following line is the LAPIS constructor
  LapisApi lapis = new LapisApi('ThisNode')

  // Do simulation/steering stuff

  // close the API instance
  lapis.Close()}
```

Figure 4.15: Pseudo-code example of use of the Close

```
main()
{ // The following line is the LAPIS constructor
  LapisApi lapis = new LapisApi('ThisNode')

  // create and publish a variable
  Integer[] state = new Integer[]
  lapis.Publish("SimulationState", state, 0)

  ...
```

Figure 4.16: Pseudo-code example of use of the Publish

The Redact Method

The Redact method removes availability of a published variable from the steering network. The method takes the string name that the variable is published under. The code below illustrates the function of the Redact method.

```
main()
{ // The following line is the LAPIS constructor
  LapisApi lapis = new LapisApi('ThisNode')

  // create and publish a variable
  Integer[] state = new Integer[]

  lapis.Publish("SimulationState", state, 0)
  // 'state' can be read and written to by other
  // nodes on the steering network

  lapis.Redact("SimulationState")
  // 'state' is no longer avialable on the steering network
```

Figure 4.17: Pseudo-code example of use of the Redact

The Lock Method

The Lock method changes the state of a published variable to read-only. The method only takes a single parameter, the text name of the variable to be locked. The code below illustrates the use of the Lock method (as well as the Unlock method, discussed below).

The Unlock Method

The Unlock method changes the state of a published variable so that it is no longer read-only. As with Lock, it takes a string with the name of the variable to be changed. See the Lock code example for an example

The Get Method

The Get method is used to get a copy of the current state of a remote published variable. It requires two parameters, one string of the name of the node name, and

```
main()
{ // The following line is the LAPIS constructor
  LapisApi lapis = new LapisApi(‘ThisNode’)

  // create and publish a variable
  Integer[] state = new Integer[]

  lapis.Publish("SimulationState", state, 0)
  // ‘state’ can be read and written to by other
  // nodes on the steering network

  lapis.Lock("SimulationState")
  // ‘state’ can now only be read

  lapis.Unlock("SimulationState")
  // ‘stat’ can now be written to again

  lapis.Redact("SimulationState")
  // ‘state’ is no longer avialable on the steering network
```

Figure 4.18: Pseudo-code example of use of the Lock

one string of the variable name. The code example below shows the use of the Get method. The code assumes a remote node named ‘RemoteNode’, has published a variable ‘RemoteData’.

The Set Method

The Set method is used to set the value of a remotely published data. It requires three parameters, one string of the name of the node name, one string of the data name, and a reference to a data variable to use as a source of data to set the remote data. The code below shows the use of the Set method under the same circumstances as the code sample for Get previously.

```
main()
{ // The following line is the LAPIS constructor
  LapisApi lapis = new LapisApi('ThisNode')

  lapis.Connect("RemoteNode")

  Integer[] remoteData = lapis.Get("RemoteData", "RemoteNode")
  ...
```

Figure 4.19: Pseudo-code example of use of the Get

```
main()
{ // The following line is the LAPIS constructor
  LapisApi lapis = new LapisApi('ThisNode')

  Integer[] data = new Integer[]

  lapis.Connect("RemoteNode")

  lapis.Set("RemoteData", "RemoteNode", data)
  ...
```

Figure 4.20: Pseudo-code example of use of the Set

4.3.3 Initialization of the API and Steering Network

Before a body of code can use the LAPIS API, it must first create an instance of the LAPIS API Object (or equivalent, depending on language). The constructor (or other, language appropriate method) that creates the LAPIS API object is responsible for starting the processes involved in the creation and running of the Daemon layer and the COM layer. Once these processes have been started and the API object is

returned to the users code, the user may move on to connecting to the LAPIS steering network, publishing and redacting data, read remotely published data, writing to remotely published data, and in all other ways interact with other nodes on the steering network.

Initialization of the steering network is done by the first node that is executed by the user. This may be a simulation that starts running via a batch scheduler process, it may be a collaborator located in a different lab starting up a collaboration interface, or a management interface that is started prior to scheduling simulations to run on a supercomputer, as just a few possible examples. LAPIS has been left without implicit structure as much as possible, making all of these organizations and more readily and easily available to the end user when they create their computational steering-enabled simulation. Much as network initialization can take many forms, connecting to the network can as well. Multiple distributed processes each connecting to a management interface form a steering network. Two collaborators connecting to a running simulation form a steering network, as do a collection of models all connecting to each other form a steering network. Again, these are just a few examples of the many different forms a LAPIS steering network may take.

4.3.4 Example Software Structures for Use with the LAPIS Steering System

There are a number of different software structures that are useful in the creation of steering systems that are easily created with the LAPIS system. A number of these structures are discussed below, with pseudo-code examples.

Low-Overhead State Sharing

Amongst the most useful features of steering systems is the ability to periodically monitor the internal state of a running simulation. With the LAPIS system, this is exceptionally easy to implement on the simulation side (more effort is, of course, required on the monitoring and visualization side). Additionally, the design of LAPIS ensures that when not actively being monitored the simulation will experience no performance impact. Implementation of simulation state sharing follows the pseudo-code below (figure 4.21). In order to implement monitoring, the visualization node will need to use the GET command to read the simulation state.

```
main()
{ lapis = new LapisApi('ThisNode')
  lapis.Connect('ThatNode')

  Integer[][] SimField = new Integer[][]
  lapis.Publish(SimField, 'ThisVariable')

  // pre-simulation tasks
  while(simulating)
  {  // run simulation core }
  // post-simulation tasks

  lapis.Close()}
```

Figure 4.21: Pseudo-code for implementation of simple simulation state sharing. Note that no changes are required within the main simulation loop at all.

Pausing a Simulation

Another relatively simple but very useful mechanism that can be easily implemented with LAPIS is the simulation-side pausing of execution. Implementation of simulation pausing is slightly more complicated than simple state sharing, but only requires one significant change to the core simulation loop. While not over-head free, implementation in the manner shown below (see figure 4.22) ensures of a minimal impact on simulation performance. When the client side application wishes to pause the running simulation, it uses a Set method to set the pause flag. To resume the simulation, another Set method is used to clear the pause flag will resume execution of the simulation.

```
main()
{ lapis = new LapisApi('ThisNode')
  lapis.Connect('ThatNode')

  Integer[][] SimField = new Integer[][]
  Integer Pause = 0
  lapis.Publish(SimField, 'ThisVariable')
  lapis.Publish(Pause, 'Pause_Flag')

  // pre-simulation tasks
  while(simulating)
  {  if (Pause_Flag != 0)
     { //pause 500 ms }
     else
     { // run simulation core }
  }
  // post-simulation tasks

  lapis.Close()}
```

Figure 4.22: Pseudo-code for implementation of simulation pausing. The only overhead when the simulation is not paused is a single 'if' statement once per simulation outer loop iteration.

Check-Pointing

Once simulation-side pausing is implemented, implementing check-pointing is a simple matter of writing a client-side application that will periodically pause the simulation, read, and then record the simulations state. Below is pseudo-code (figure 4.23) that could be used to implement such a client. In the case of this code, the client will be a single-checkpoint-per-run implementation. This could be extended to a periodic check-pointing client by adding an outer loop and either regular timing controls or a user interface.

```
main()
{ lapis = new LapisApi('Checkpointer')
  lapis.Connect('Simulation')

  Integer[][] state = lapis.Get('SimulationState', 'Simulation');
  Integer timeStep = lapis.Get('SimulationTimeSTep', 'Simulation');
  Double sigma = lapis.Get('SimulationSigma', 'Simulation');
  // collect other relevant state from simulation

  // write all collected state to disk

  lapis.Close()}
```

Figure 4.23: Pseudo-code for implementation of a check-pointing node. An outer loop after the initialization and connection to the LAPIS network would allow for a repeated collection of checkpoints.

Simulation Rewinding

Simulation rewinding is a computational technique where the end user of the simulation can revert the simulation to a previously run time-step with the intension of making some parameter change and watching the simulation re-run and observing the new state. Simulation rewinding can be implemented as an extension of check-pointing systems by creating regular check-points and creating a mechanism to pause a simulation, select a check-point, and load the saved state back into the simulation. The following pseudo-code (see figure 4.24) is for a rewinding interface that will issue commands to a remote steered simulation to rewind that simulation to a previous state. This code assumes that previously recorded states exist, and that the simulation implements a pause loop and publishes all relevant data to the steering network.

```
main(Integer rewindAmount)
{ lapis = new LapisApi('Rewinder')
  lapis.Connect('Simulation')

  lapis.Set('Pause_Flag', 'Simulation', 1)

  Integer timestep = lapis.Get('SimulationTimeSTep', 'Simulation')
  timestep = timestep - rewindAount

  Integer[][] state = file.load('checkpoint_' + timestep)

  lapis.Set('SimulationState', 'Simulation', state);
  lapis.Set('SimulationTimeSTep', 'Simulation', timeStep);
  // load from disk and SET other relevant simulation state

  lapis.Set('Pause_Flag', 'Simulation', 0);

  lapis.Close()}
```

Figure 4.24: Pseudo-code for implementation of a rewinding node. A user interface would better allow for selection of previous states and handling cases of multiple rewinds per execution.

Controlling a State-Diagram-Based Simulation

Much like the implementation of a pause flag within the simulation loop can easily allow a developer to pause their simulation, a more complicated state diagram based simulation could be controlled. As with the pause flag, publishing to the steering network the variable that represents the state of the simulation will allow a remote user interface to modify that state. A slightly more advanced approach might be to create a monitoring application that is also attached to the steering network that monitors some other state variables within the simulation and changes

the state variable appropriately and autonomously before alerting a user interface (again, attached to the steering network) by setting a flag within the user interface code.

Managing Dynamically Growing Data Sets

In many simulations, the output of data occurs continuously as the simulation executes. In cases like these, it is possible to leverage the LAPIS system to publish data to the steering network for access to other bodies of code (User interfaces, monitors, other models, etc) in such a way that the data is still accessible even if the generating simulation looses scope on the data. The snippet of pseudo-code below (figure 4.25) would be within the main simulation loop, but not within the individual time-step loop. The simulation would be setup to run a determined number of time-steps upon each iteration of the outer most loop.

```
Integer week = 0
input = load(initial_conditions)

// main simulation loop
while (simulating)
{ Integer[] output = new Integer[]
  output = SimulateOneWeek(input)

  lapis.Publish(output, 'output_' + week.toString())

  input = output
  week = week + 1}

lapis.Close()}
```

Figure 4.25: Pseudo-code for publishing data vectors that grow wile the execution of the simulation proceeds. Note that even though output is reinitialized at each execution of the loop, each copy is preserved within the LAPIS framework.

4.4 Developing Extensions for the LAPIS System

In addition to providing a simple and intuitive interface for simulation developers, LAPIS provides various layers of abstraction that will allow for third party developers to extend the functionality of LAPIS.These layers provide a mechanism for creating additional APIs and communicator modules, as well as feature rich, higher level specialty systems. Connections between the API, Daemon, and communicator are all done via a small set of well documented messages over local TCP/IP ports. This standard and open method of control message passing allows developers to create new API and communicator versions for their own and community use. Additionally, the system does not require that an API even be present. Any module that responds

appropriately to LAPIS control message from the Daemon can be used instead of a standard API. In this way, developers can create other components that access and manipulate the LAPIS steering network without relying on the default API provided, should more specialized functionality be needed. Finally, given the low level and flexibility, higher level tools can and should be developed on top of the API, providing simulation developers with more specialized tools, APIs, and modules. At the same time, developing these tools on a base standard helps ensure code reuseability and robustness.

4.4.1 LAPIS Control Messages

All communication between layers of the LAPIS system that make up a node, and in between nodes thought the steering network, with the exception of the transfer of data, is handled via LAPIS Control Messages. When working on the creation of a new API, or when working on a software module that takes the place of the API and works directly with the Daemon layer, understanding the format and purpose of all of these messages is critical. Below is a summary of all the LAPIS Control Messages. For each message (or collection of similar messages) a table is given with the message format and a description of what each field in the format represents. Additionally, the function of each message is given.

DEBUG, MESSAGE, MISC, and ERROR Messages

These message types are all included as a means of sending various types of information between nodes that would not otherwise be transmittable with the other control message types.

Field	Description
MESSAGE	The message as plain text that is being sent.

NODE_LIST and ELEMENT_LIST Messages

These messages are requests form the API for a list of nodes or data elements, or a response from the Daemon with the list attached.

Field	Description
CSV List	In the case of NODE_LIST and ELEMENT_LIST messages originating from the Daemon layer, a CSV list of either node names or data names will be included in this field. In the case of ELEMENT_LIST, the each data element will have the form (data name)@(node name).

PUBLISH Message

The PUBLISH messages is sent by the API whenever a new data element is published to the steering network. Additionally, the Daemon forwards the PUBLISH message to all other nodes in order to update them of the PUBLISH event.

Field	Description
ORIGIN_NAME	The name of the node the PUBLISH message originated from
DATA_NAME	The name of the data object being published
LOCK_STATE	Weather or not the data is read-only
DESTINATION_ADDRESS	This field is not filled in by the API when sending the PUBLISH message to the Daemon. The Daemon uses this field in broadcasting the message to all other nodes. This field is the address to which the message should be forwarded.

REDACT Message

The REDACT messages is sent by the API whenever a published data element is redacted from the steering network making it unavailable. Additionally, the Daemon forwards the REDACT message to all other nodes in order to update them of the REDACT event.

Field	Description
ORIGIN_NAME	The name of the node the REDACT message originated from
DATA_NAME	The name of the data object being redacted
DESTINATION_ADDRESS	This field is not filled in by the API when sending the REDACT message to the Daemon. The Daemon uses this field in broadcasting the message to all other nodes. This field is the address to which the message should be forwarded.

LOCK Message

The LOCK messages is sent by the API whenever a the lock state of a published data element is changed. Additionally, the Daemon forwards the LOCK message to all other nodes in order to update them of the LOCK event.

Field	Description
ORIGIN_NAME	The name of the node the LOCK message originated from
DATA_NAME	The name of the data object whose lock state is being changed
LOCK_STATE	Weather or not the data is being locked or unlocked
DESTINATION_ADDRESS	This field is not filled in by the API when sending the LOCK message to the Daemon. The Daemon uses this field in broadcasting the message to all other nodes. This field is the address to which the message should be forwarded.

GET Message

The GET messages is sent by the API to the Daemon in order to initiate a transfer of a remotely published variables state to the requesting node. When the Daemon receives the GET message, it will translate the message into a GET_SETUP message which is forwarded along to the COM.

Field	Description
OWNER_NAME	The name of the node the owns the requested data
DATA_NAME	The name of the data object being requested
XFER_PORT_NUMBER	The port number that should be used for the data transfer between the API and the COM on this node

GET_SETUP Message

The GET_SETUP message is sent by the Daemon to the COM to continue the process of setting up the data transfer channel for the GET request. When the COM gets the GET_SETUP request, it forwards it to the relevant node in the steering network. When the COM on that node receives the GET_SETUP message it forwards along to that nodes Daemon. That Daemon will translate the message into a GET_BEGIN message before sending it along to the API of the node that owns the requested data.

Field	Description
ORIGIN_ADDRESS	The address of the node the GET request originated from
DESTINATION_ADDRESS	The address of the node that owns the data requested by the GET
DATA_NAME	The name of the data object being requested
XFER_PORT_NUMBER	The port number that should be used for the data transfer between the API and the COM on this node
XFER_TAG	The unique tag associated with the transfer, assigned by the Daemon

GET_BEGIN Message

The GET_BEGIN message is sent by a Daemon to the API when the Daemon receives an external GET_SETUP message. The GET_SETUP message is translated to a GET_BEGIN message which is forwarded to the API that owns the requested data.

Field	Description
ORIGIN_NAME	The name of the node the GET request originated from
DATA_NAME	The name of the data object being requested
XFER_PORT_NUMBER	The port number that should be used for the data transfer between the API and the COM on this node

SET Message

The SET messages is sent by the API to the Daemon in order to initiate a transfer new data to a remotely published variable. When the Daemon receives the SET message, it will translate the message into a SET_SETUP message which is forwarded along to the COM.

Field	Description
OWNER_NAME	The name of the node the owns the variable the data will be written to
DATA_NAME	The name of the data object being written to
XFER_PORT_NUMBER	The port number that should be used for the data transfer between the API and the COM on this node

SET_SETUP Message

The SET_SETUP message is sent by the Daemon to the COM to continue the process of setting up the data transfer channel for the SET request. When the COM gets the SET_SETUP request, it forwards it to the relevant node in the steering network. When the COM on that node receives the SET_SETUP message it forwards along to that nodes Daemon. That Daemon will translate the message into a SET_BEGIN message before sending it along to the API of the node that owns the target data.

Field	Description
ORIGIN_ADDRESS	The address of the node the SET request originated from
DESTINATION_ADDRESS	The address of the node that owns the variable that will be written to
DATA_NAME	The name of the data object being written
XFER_PORT_NUMBER	The port number that should be used for the data transfer between the API and the COM on this node
XFER_TAG	The unique tag associated with the transfer, assigned by the Daemon

SET_BEGIN Message

The SET_BEGIN message is sent by a Daemon to the API when the Daemon receives an external SET_SETUP message. The SET_SETUP message is translated to a SET_BEGIN message which is forwarded to the API that owns the target variable.

Field	Description
ORIGIN_NAME	The name of the node the SET request originated from
DATA_NAME	The name of the data object being written to
XFER_PORT_NUMBER	The port number that should be used for the data transfer between the API and the COM on this node

INITIALIZE Message

The INITIALIZE message is sent from the API to the Daemon to inform the Daemon process that the steering network has been initialized. At the moment, this has no impact, but the functionality has been reserved for future implementation needs.

Field	Description
NUMBER	This field has been reserved for future use.

CONNECT Message

The CONNECT message is sent from the API to the Daemon to initialize the connection process to the steering network. When the Daemon receives this message, it results in the generation of an UPDATE message to the connection point.

Field	Description
ORIGIN_NAME	The name of the node the CONNECT request originated from (this node)
ORIGIN_ADDRESS	The address of the node the CONNECT request originated from (this node)
DESTINATION_NAME	The nae of the node to connect to
DESTINATION_ADDRESS	The address of the node to connect to

DISCONNECT Message

The DISCONNECT message is sent by the API to the Daemon, which forwards it on to every node on the steering network.

Field	Description
ORIGIN_NAME	The name of the node the DISCONNECT request originated from (this node)
ORIGIN_ADDRESS	The address of the node the DISCONNECT request originated from (this node)

UPDATE Message

The UPDATE message is sent between Daemons, following the process described in section 4.1. When Daemons receive UPDATE messages, they compare the node list and data list to that contained on the Daemon already. A list of nodes that are only one one of the two lists is generated. Then, UPDATE messages are sent to all of those nodes.

Field	Description
ORIGIN_NAME	The name of the node the UPDATE message originated from
ORIGIN_ADDRESS	The address of the node the UPDATE request originated from
CSV_NODES	A CSV list of known nodes
CSV_DATA	A CSV list of published data, in (dataname) @ (nodename) format.
DESTINATION_ADDRESS	The address of the node the UPDATE message should be sent to.

CLOSE Message

The CLOSE message is sent by the API to shutdown the Daemon and COM process and clean up the node before exiting. If the Daemon receives a CLOSE message and is not disconnected, it generates a DISCONNECT message prior to shutting down the COM and itself. The CLOSE message has no parameters.

4.4.2 LAPIS API Layer Development: The LAPIS API Standard

In development of new API layer extensions to LAPIS such as new language APIs or other components that directly interface with the Daemon layer, the LAPIS API standard provides an important guide for implementation. This API standard is composed of two distinct parts: the standard handling of API method calls and the standard handling of LAPIS Control Messages by the API layer. For development of APIs, both of these standards must be adhered to, however for development of all other API layer applications that do not implement the LAPIS API, only the standard response to control messages must be considered. Each of these halves of the API standard are discussed method by method or message by message, accordingly, below.

Standard Requirements for LAPIS API Methods

When implementing a version of the LAPIS API, all of the following methods must be implemented. A description of what actions each method must perform is given below in the heading under the method name.

The Constructor Method The constructor method is responsible for starting the Daemon. Upon starting the Daemon, the Daemon will output the nodes address on the steering network. This value should be parsed and recorded by the API constructor as part of the APIs state. Additionally, the mechanism that handles control messages sent from the Daemon must be started here.

Initialize Method This method must create and send a INITIALIZE control message to the Daemon.

Connect Method This method must create and send a CONNECT message to the Daemon.

Disconnect Method This method must create and send a DISCONNECT message to the Daemon.

NodeList Method This method must create and send a NODE_LIST message to the Daemon and then block until the monitor thread wakes the API thread. At this point, the method must return the node list.

ElementList Method This method must create and send a ELEMENT_LIST message to the Daemon and then block until the monitor thread wakes the API thread. At this point, the method must return the element list.

Publish Method This method must create and send a PUBLISH message to the Daemon for the passed data name and reference. Additionally, it should record the variable reference, and whatever other information may later be needed by the monitor thread in order to handle GET and SET requests.

Redact Method This method must create and send a REDACT message to the Daemon for the passed data name. Additionally, it should remove any record made of the variable from whatever method was used to store it when it was published

Lock Method This method must create and send a LOCK message for the appropriate data name passed to the method.

Unlock Method This method must create and send a UNLOCK message for the appropriate data name passed to the method.

Get Method This method must create and send a GET message to the Daemon that includes a local port number. This method then must read the transfered data stream from the port and return the reconstructed data object.

Set Method This method must create and send a SET message to the Daemon that includes a chosen local port. Then it must write the passed data to a local port.

Close Method This method must create and send a CLOSE message to the Daemon. Then, it must shut down the monitor thread and any and all local socket connections associated with the Daemon.

Standard Requirements for LAPIS API handling of LAPIS Control Messages

DEBUG, MESSAGE, MISC, ERROR Whenever the API receives one of these four message types, it should log the type of message, and the message text into whatever logging system is in use.

GET_BEGIN

SET_BEGIN

NODE_LIST When the API monitor receives a NODE_LIST message from the Daemon, it should store the list contained within the message in a location where

the waiting API call can access it, then wake the API call via whatever appropriate mechanism is available.

ELEMENT_LIST When the API monitor receives a ELEMENT_LIST message from the Daemon

4.4.3 LAPIS COM Layer Development: The LAPIS COM Standard

Similar to API layer development, a LAPIS COM standard provides a guide for implementation. The standard covers all the actions the COM must take when certain LAPIS Control Messages are received, either from the external interface, whatever mechanism that may be, or from the internal socket that connects the Daemon layer to the COM layer. All LAPIS Control Messages are forwarded from the interface they arrive on to the other interface, and all such messages that require no further action are not listed in this section. The following list covers all the control messages for which a standard COM must take additional action.

Internal Listener Responses

This section covers all control messages sent by the Daemon to the COM.

GET_SETUP When the COM internal listener receives a GET_SETUP message from the Daemon, it must create a data transfer thread that reads data from a connection to the steering network and writes that data to a local socket provided in the message. It must then replace the socket in the message with the address on the steering network used for that connection before forwarding the message though the steering network to the destination node.

SET_SETUP When the COM internal listener receives a SET_SETUP message from the Daemon, it must create a data transfer thread that reads data from a local socket provided in the message, and writes that data to an address on the steering network. It must then replace the socket in the message with the address on the steering network that was used for that connection before forwarding the message though the steering network to the destination node.

CLOSE When the Daemon sends a CLOSE message to the COM, it immediately terminates both listener threads and ends execution.

External Listener Responses

This section covers all messages the COM receives by monitoring the steering network.

GET_SETUP When the COM implementation receives a GET_SETUP message from the steering network, it must create a data transfer thread that reads from a local socket connected to the API, and write the data to the steering network address provided in the message. Prior to forwarding the message on to the Daemon, the steering network address must be replaced with the local socket used by the transfer thread.

SET_SETUP When the COM implementation receives a SET_SETUP message from the steering network, it must create a data transfer thread that reads from the steering network address provided in the message, and write the data to a local socket connected to the API. Prior to forwarding the message on to the Daemon, the steering network address must be replaced with the local socket used by the transfer thread.

4.5 Summary of Key System Features, Advantages, and Abilities of the LAPIS System

With a full implementation of the LAPIS system for both Java and MATLAB, it has become clear that the feature set of LAPIS is such that it resolves most of the problematic issues presented by steering systems. The development difficulty often associated with the use of steering systems is mitigated by means of a very simple and easy to understand API, simple installation, and simple setup. A scientific simulation developer can pick up and learn to use the LAPIS system in a short period of time with relatively little effort compared to other available solutions, and without changing the entire paradigm by which they develop.

Simulation portability and maintainability are likewise maintained. Simulations can be switched from running entirely on a single machine to running across a distributed collection of nodes with little more than a single change in a configuration file. Thanks to the cross platform compatibility of Java, the only constraint on where the users simulation can run is the users simulation itself. Even systems with different security constraints on inter-system communication can be accommodate via an appropriate COM implementation. Additionally, there is no reason to fork development of the simulation code body into steered and unsteered versions. So long as the simulation makes use of the LAPIS system carefully, the steering functionality can go unused and when not utilized represents nearly zero overhead.

The LAPIS system will also help encourage more third party software and hardware development and support. With the LAPIS API and COM standards, third party developers have a target to aim for in the development of more sophisticated academic and commercial tools. Additionally, knowing the standards will remain in

place will give assurance to the third party developers that effort put forth to create such applications and systems will not be in vain.

Along with this issues, LAPIS helps address a host of smaller problems previously discussed. Data compatibility between languages and systems is handled at the API layer, freeing the simulation developer of any thought of the issue. Data security issues can be addressed by implementation of a COM module that encrypts all data traffic between nodes in the steering network. LAPIS imposes no specific architecture on the simulation developer and provides enough flexibility to sufficiently model any architecture the developer may desire. Finally, LAPIS provides a steering solution with extremely low overhead in idle steering states, allowing the simulation to maintain a high level of throughput.

Chapter 5: Implementations of Example Simulations with LAPIS

To date, the LAPIS system has been used to implement a variety of example models and systems. These examples serve both as debugging opportunities, system tests and as demonstrations of the abilities of the LAPIS system in a variety of situations. In addition to the academic examples, the LAPIS system has been used to implement a multi-model simulation related to the NSF CDI project. Each of these examples as well as the multi-model simulation are discussed in further detail below.

5.1 Steering Conway's Game of Life

The first application of the MATLAB version of the LAPIS was creating a steered version of the Conway's Game of Life. Conway's Game of Life was chosen as MATLAB includes an open source implementation (see appendix A for unmodified source, and appendix A for the modified code). Using the existing implementation of the Game of Life as a starting point for development demonstrates how to apply LAPIS to an existing software structure. At the same time, this example shows how little development effort is required to create a steering tool from existing code and how

adding steering code to an existing code body has little to know impact on performance or function of that code. Additionally, using the Game of Life allows the demonstration of LAPIS' ability to decouple user interfaces from simulation cores.

5.1.1 Technical Description of the Demo

Conways Game of Life is a so called "0 player game" [?] in which a field of pixels and a basic template are used to simulate a collection of cells. The game follows the following simple rules to determine if at the next step if any given cell will be 'live' or 'dead':

1. Any live cell with fewer than two live neighbors dies
2. Any live cell with two or three live neighbors lives
3. Any live cell with more than three live neighbors dies
4. Any dead cell with exactly three live neighbors becomes a live cell

The stock MATLAB implementation of the Game of Life includes a graphical display of the field of pixels on which the simulation takes place, and some basic controls including a start and stop button (see figure 5.1). This stock program was modified with four minor ways in order to convert it to a steered, pause-able version: A section that declares an instance of the steering API and that published the data field were added (see figure 5.2), also a flag-check in the main execution loop was added and all references to the field variable, X were modified to $X.d$ (see figure 5.3). While running, the resulting version of the game of life looks and functions identically to the original in the absence of steering.

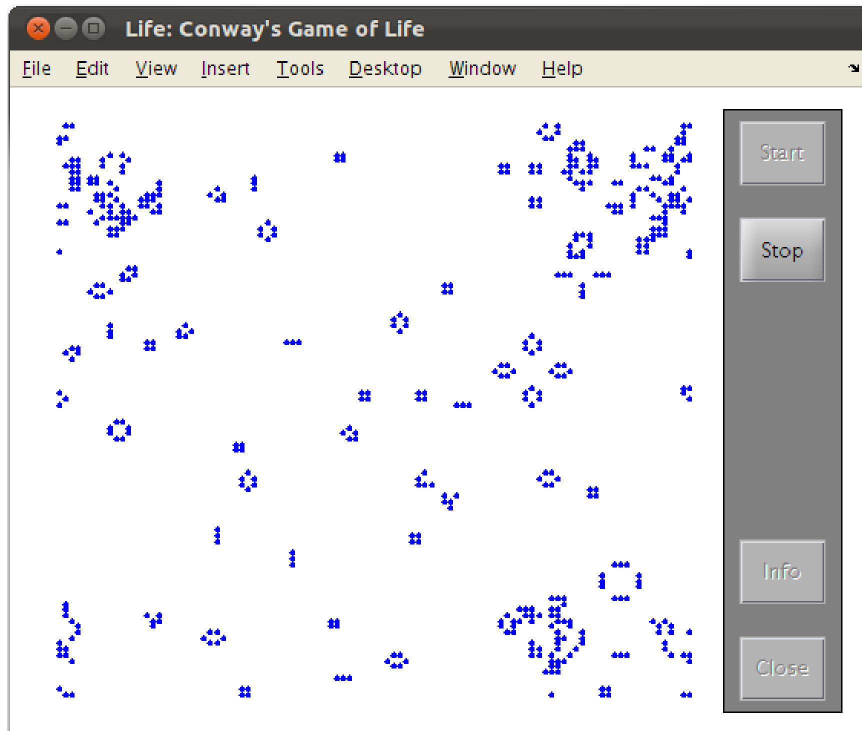

Figure 5.1: The Stock Game of Life Interface

Paired with the modified version of the Game of Life is a simple steering GUI (see figure 5.4). The GUI connects to the running instance of the modified Game of Life. Once connected, the GUI allows the user to pause the remote execution. When the Game of Life is paused the current state of the game is downloaded by the steering GUI and displayed. The user can then interactively add points to the field by clicking on grid points in the steering GUI. When the user resumes the remote simulation, an updated version of the field is automatically sent to the game before the game resumes.

```
% ====== Start of Demo
set(axHndl,'UserData',play,'DrawMode',...
   'fast','Visible','off');
m = 101;
%---added code for LAPIS-----------------------------
lapi = LapisApi('GameOfLife');
X = LapisData('field', zeros(m,m));
pause_flag = LapisData('pause', 0);
lapi.Publish('field', X, 0);
lapi.Publish('pause', pause_flag, 0);
%----------------------------------------------------
p = -1:1;
for count=1:15,
```

Figure 5.2: Initialization of the Game Of Life, additions included to initialize the LAPIS API and publish data to the steering network.

5.1.2 Lessons Learned from this Demo

As the first full-featured demo created using the LAPIS system, it quickly became clear that the start-up process for the LAPIS API was unnecessarily complicated. As such, it was subsequently modified to require fewer parameters and less information from the user. Additionally, this showed that adding an ‘if’ statement within the main execution loop that checks a pause flag is an effective way to pause a running simulation so that steering actions can take place synchronously. This methodology has been used in a number of other applications subsequently.

5.2 Managing Hill Climbing Clients with a Manager Interface Using LAPIS

The second example implemented was a multi-system, multi-language hill-climbing demonstration. The hill climbing demonstration further demonstrates LAPIS’ ability to distribute independent computational components with little effort on the part of

```
while get(axHndl,'UserData')==play,
  if(pause_flag.d(1) == 0)
      % How many of eight neighbors are alive.
      N = X.d(n,:) + X.d(s,:) + X.d(:,e) + X.d(:,w) + ...
      X.d(n,e) + X.d(n,w) + X.d(s,e) + X.d(s,w);
      % A live cell with two live neighbors, or any cell with three
      % neigbhors, is alive at the next time step.
      X.d = double((X.d & (N == 2)) | (N == 3));
```

Figure 5.3: Start of the Game Of Life simulation loop with added 'pause' code and X replaced with $X.d$

the developer.Additionally, Hill Climbing example demonstrates the interoperability between languages that LAPIS provides. Finally, the hill climbing demonstration shows an example management interface created with LAPIS. The system consists of three distinct components: MATLAB hill-climbers, Java hill-climbers, and a MATLAB management GUI, from which users can inspect and manage the hill-climbers as they run. All coordination between the MATLAB management GUI and the hill-climbers is done through the LAPIS system. The hill climbers have the capacity to act autonomously, but are also designed to take real-time direction from the management GUI.

5.2.1 The Hill-Climber Manager GUI

The management GUI (see figure 5.5 and appendix A for code) provides a view of all the active hill climbers and their current state. The GUI is started before the hill climbers and the climbers register with the manager when they start, populating a list-box on the management GUI. By selected individual hill climbers from the list-box, the management GUI can be used to view the path the selected climber has taken

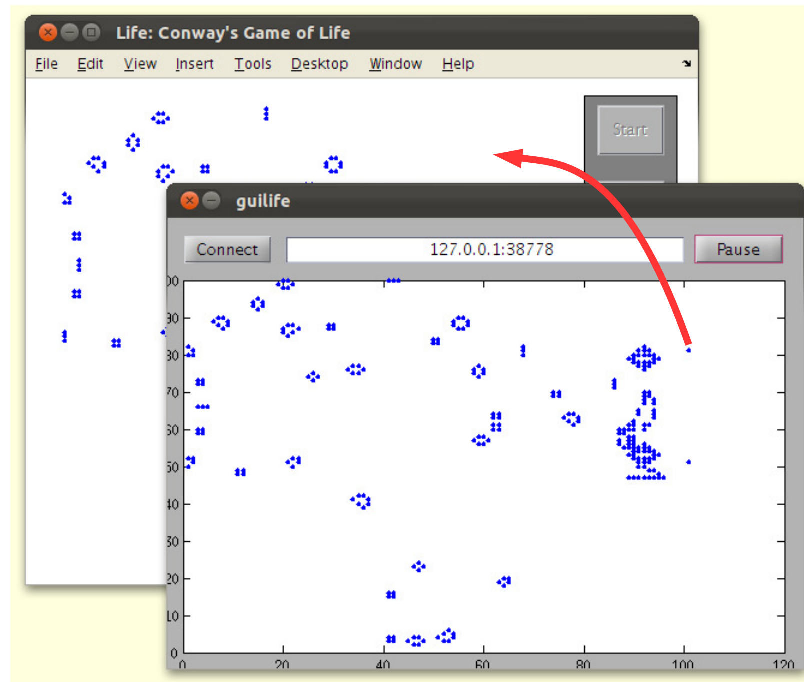

Figure 5.4: The Steered Game of Life User Interface: Changes made in the steering interface (foreground) are reflected in the simulation (background) when the simulation is resumed.

and the climbers current state. Additionally, the user can perturb the climber if it has reached a steady state, reset the climber entirely, pause the climbers progress, or upload a new topography to the climber, all with single button presses. From the end users perspective, the implementation language of the climber is unknown and there is no functional difference between the Java climbers and the MATLAB climbers.

5.2.2 The Java and Matlab Hill Climbers

Both the MATLAB and Java hill climbers (see appendix A for code) use a simple template method to find peaks in a topography. The climber inspects the values of

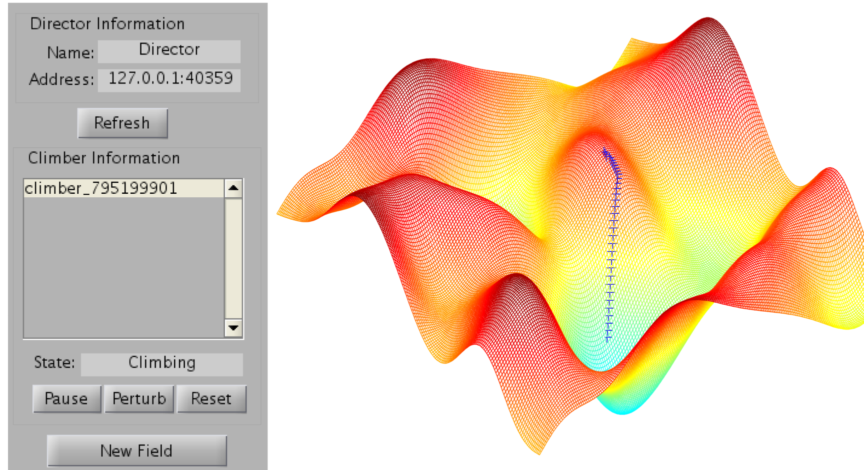

Figure 5.5: The Hill-Climbing Director

all values in the topography grid of it's 8-neighbors. Upon finding the largest, the climber then sets it's current position to the position amongst the 8-neighbors with the highest value (or leaves the position value alone if none of the 8-neighbors have a higher value). The climbers behavior at a given time-step is determined by an internal state variable. Initially, the state variable is set to an initialization state. This causes the climber to download the topography provided by the manager GUI and then switch to the climbing state. Nominally, once the climber is in the climbing state it performs a step of hill climbing at every iteration of the simulation loop unless a peak has been found, or external action has been taken, such as state modification by the Climbing Manager. The climbers allow for such external changes of state by publishing a state variable. The climbers internal state can be set so as to pause the climber, cause the climber to randomly deviate from it's current location, or to download a new topography. Despite being implemented in different languages, both the Java and MATLAB climbers operate identically as far as the director is concerned.

The manner in which the climbers access the internal state of the manager, and the way in which the manager manipulates the internal state of the climbers does not vary at all between languages. Additionally, the implementation details relating to this functionality are nearly identical in both the Java and MATLAB versions.

5.2.3 Lessons Learned from this Demo

The most striking lesson learned during this demo was with regard to the limitations of the MATLAB timer system. Specifically, an I/O blocked timer in MATLAB cannot be interrupted by another timer. As the MATLAB version of LAPIS uses a timer internally in order to approximate threading, and since that timer thread might block, inclusion of another I/O blocking timer that blocks on LAPIS calls can cause deadlock. Currently, there is no known way around this problem, but hopefully The Mathworks will improve the explicit multi-threading support in future versions of MATLAB.

5.3 A Simple Multi-Model Simulation of a Sailboat, Wind & Current Connected Via LAPIS

The Sailboat model simulates the path of a sailboat in the presence of wind and water currents. This example demonstrates the use of the LAPIS framework in the creation of a multi-model simulation based on similar component models with each component model executing independently and asynchronously. The example involves three component simulations: the sailboat simulator, the current simulator and the wind simulator. The wind and water simulations each generate a constantly varying two dimensional vector field used to represent force vectors applied by the corresponding environmental factor. The sailboat tracks it's own position and velocity

vector through each of it's iteration loops and samples the near by space in both the wind and water vector fields continuously, using the near by vectors to compute both a new position and velocity at each time step.

5.3.1 Wind and Water Models

The wind and water models are nearly exactly the same in terms of implementation (see appendix A for code), varying only by choice of a central periodic function. The models both generate a slowly varying vector fields based on these periodic functions. This vector field is an internal state variable of the simulation and is published to the LAPIS steering network at the initialization of the models at run-time. Subsequent to the publishing of this internal state, the models makes no additional reference to the LAPIS API for the remainder of execution time, thus limiting performance loss due to use of the LAPIS steering tool. At the end of simulation, the models make one final call to the LAPIS API to clean up the execution environment prior to exiting.

The models create their internal vector fields state at each time step of simulation. This process starts with evaluation of the the periodic functions below for the water current field (eq. 5.1) and the wind current field (eq. 5.2) over a 20 by 20 integer grid. This evaluation results in a 20 by 20 grid of values that represent a topography. Using this topography, the gradient is calculated at each of the points in the 20 by 20 grid, resulting in the desired internal vector field. Finally, after generation of this vector field, the values of dx and dy are incremented by a small random amount in order to induce smooth variation between points in the vector field between time-steps. While the simulation does not implement an accurate model of wind or water current, changing to a more accurate model of water and wind current and sailboat

model does not require any significant addition LAPIS integration effort. The wind and water current simulators run independently and asynchronously from each other and the sailboat simulation. Both simulations simply generate data and never initiate communication with the sailboat model. Example vector field outputs can be found in figures 5.6 and 5.7.

$$\begin{aligned} water(x, y) &= \sin((x + dx)^2) + \cos(x + dx)^2 \\ &\quad + \sin((x + dx) * (y + dy)) + \sin(y) \end{aligned} \tag{5.1}$$

$$\begin{aligned} wind(x, y) &= \sin((x + dx)^2) + \cos(x + dx)^2 \\ &\quad + \sin((x + dx) * (y + dy)) + \sin(y) + \sin((y + dy)^2) \end{aligned} \tag{5.2}$$

5.3.2 Sailboat Models

The sailboat model (see appendix A for code) contains no internally published state and has no external inputs into the model. The sailboat model instead relies entirely upon the published state of the wind and water models, and upon LAPIS to access that published state. At each time-step, the sailboat first updates its position based upon an internal velocity vector computed during the previous time-step. Then, the sailboat model GETs a copy of the vector fields from both the wind and water models. The sailboat model then takes the vectors closest to it's current location in both fields and computes a linear average vector representing the influence of wind

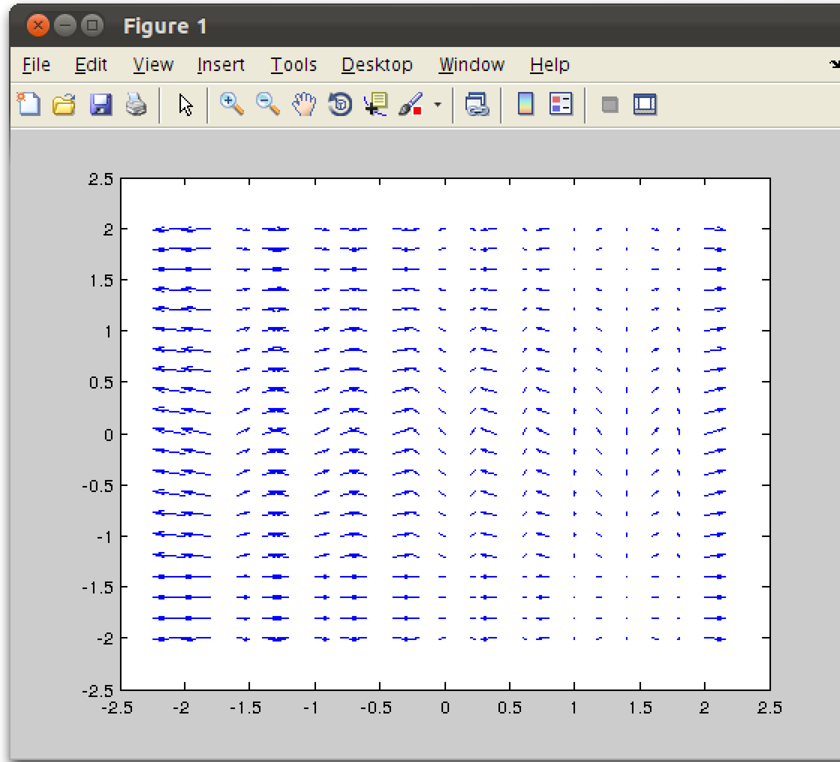

Figure 5.6: The vector field generated at one particular time-step of the Water Models execution

and water currents at the current time-step. Using these two vectors, the sailboats current velocity vector and position, the sailboat model computes a new velocity vector, and the computational loop for the current time-step ends. The results of this simulation are displayed in real-time via an output plot generated by the sailboat model. An example execution of the simulation can be seen in figure 5.8, where the wind vector field is shown in blue, the water vector field is shown in blue, and the path of the sailboat is shown in red. Note that the edges of the simulation field wrap in both directions (representing a flattened toroid).

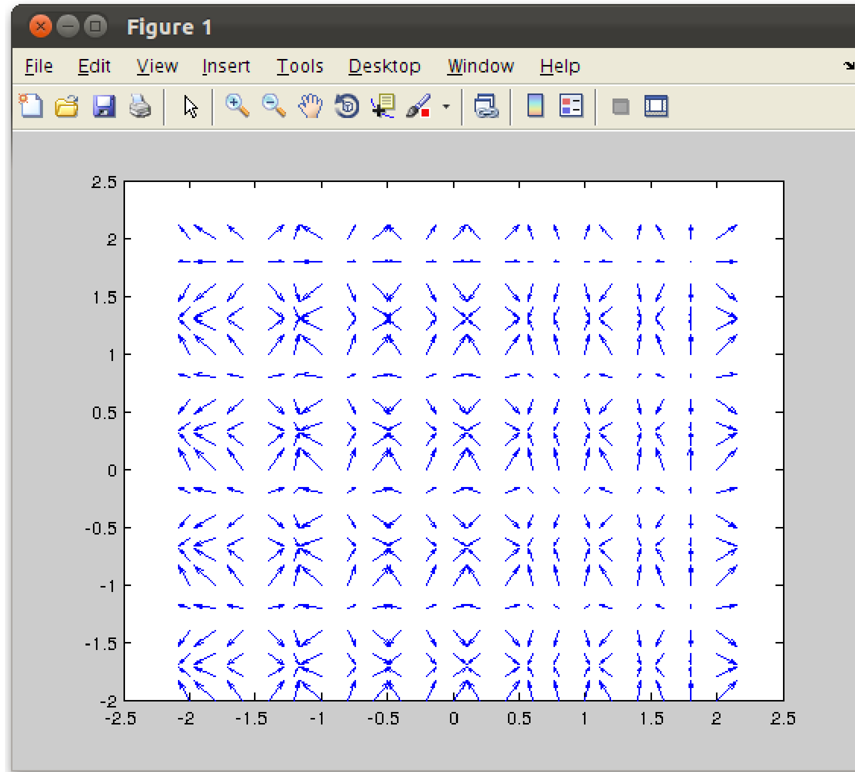

Figure 5.7: The vector field generated at one particular time-step of the Wind Models execution

5.4 Implementation of a Feed-Back Control System of a Driver & an Engine With LAPIS

The Driver & Engine multi-model simulation highlights the ability of LAPIS to act as an interconnect between independently executing models arranged in a feed-back loop. By making such connections, LAPIS can be used to efficiently create more complicated multi-model simulations, control system simulations, or data-flow simulation out of stand alone, simpler component simulations. In the case of this example, the Driver simulation seeks to model the behavior of a driver when presented with a speed limit and the velocity output of a engine or vehicle. At the same time, the

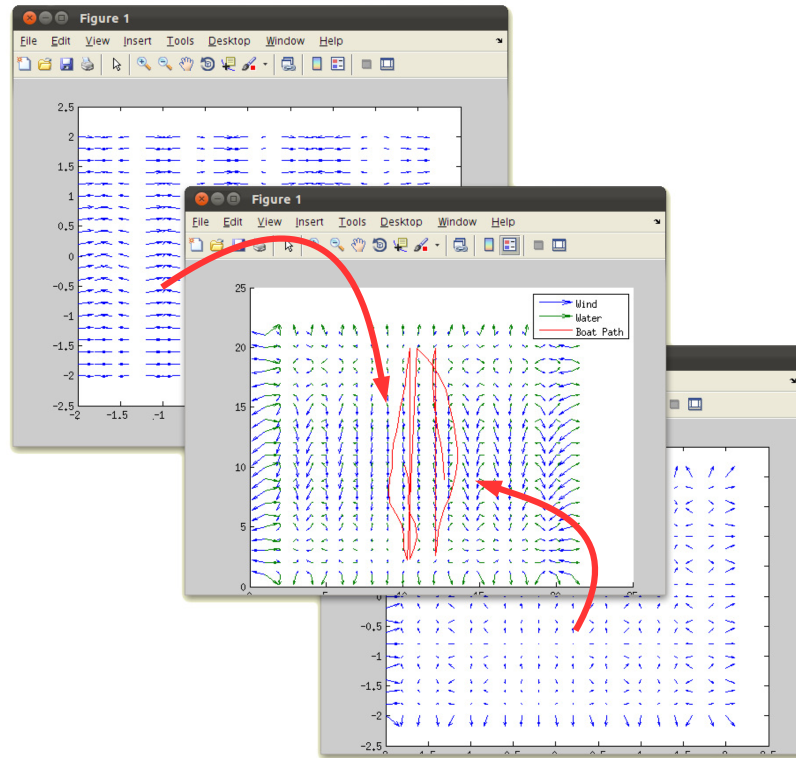

Figure 5.8: The Boat simulation is a simple multi-model simulation. The boat reads a position vector from both other simulations to determine it's next travel vector.

Engine simulation models the behavior of typical engine, responding to an accelerator pedal position. Both of these models run asynchronously and in separate memory spaces, as with the previous models. Unlike previous models, the behavior of each model is specifically tied to the internal state of the companion model. Additionally, the Driver simulation includes a graphical output that demonstrates how LAPIS can be used to monitor the state of a long running simulation with little to no performance impact on the simulation itself. Both the Driver and Engine model as well as the graphical output are discussed in more detail below.

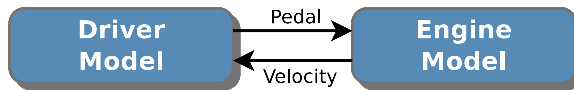

Figure 5.9: Driver/Engine Model is arranged in an asynchronous feedback loop where the instantaneous state of each model directly influences the next state of each model

The Driver Model

The roll of the Driver model in this example is to simulate the behavior of the driver of a theoretical vehicle. Similar to the sailboat model in the previous example, the Driver model runs asynchronously from the Engine model, and also relies on internal data from the engine model to compute it's internal state at each time-step (see appendix A for code). Unlike the sailboat model in the previous example, the Driver model does publish its internal state variables for use by the Engine model. The driver model reads values from a 'speed limit' file gets the current speed of the vehicle from the engine model via a call to the LAPIS 'GET' command. Using these two values, the driver model produces a pedal position value.

The Engine Model

The engine model likewise runs asynchronously from the driver model (see appendix A for code). At the start of each iteration, the pedal position from the Driver model is read, again with a LAPIS 'GET' command. The accelerator pedal position and the previous speed of the engine are used by the Engine model to calculate the new engine speed for the next time-step. The Engine models dependence on the Driver models 'pedal position' state, and the Driver models dependence on the Engine models 'engine speed' state creates a rudimentary feedback loop. This feedback

loop allows two simple models to engage in more complex and interesting behavior, and emergent behavior is seen from the overall simulation (see figure 5.10).

The Output Plot

The output plot acts as a window into the simulation (see figure 5.10). The GUI starts with the driver Driver model and output from both models is plotted during each iteration of the driver model. This window allows the user to observe the emergent behavior of the two models. In this case, the parameters and sampling speeds of both the Driver model and Engine model lead to an oscillation around the speed limit. The user can observe qualitatively that the model is most likely stable, though the response of the model to the speed limit input file is less than ideal. For a more interesting and thorough investigation of this system, a steering GUI could be created that gives user access to the internal parameters of both models, including sample speed, and allows the user to control the speed limit input rather than using a file. Such a setup would allow for real-time, intuition based experimentation. While the resulting parameter set might no be idealized, it will doubtlessly be much closer to an idealized parameter set than a randomly selected parameter set. Such a resulting parameter set would then be an ideal starting point for a more rigorous and thorough optimization problem.

5.5 The Use of LAPIS in Implementation of the CDI Multi-Model Simulation

LAPIS has been used as part of the NSF Cyber-enabled Discovery and Innovation (CDI) project, specifically in the creation of several multi-model simulations of energy consumption and provisioning. The CDI project seeks to study power consumption,

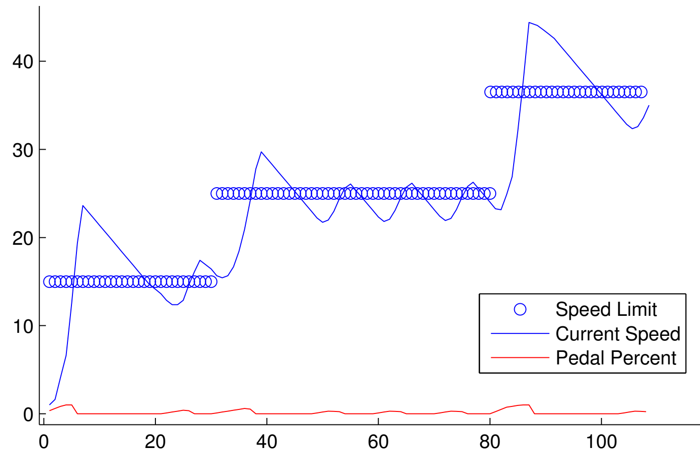

Figure 5.10: Example output of the Driver/Engine model. In this case, the models result in an emergent behavior of oscillation around the speed limit.

generation and provisioning, transportation, and technological effects on the power grid. A deeper understanding of these factors will allow for a better understanding of how advanced technologies (such as plug-in hybrid electric vehicles, renewable energy sources, smart grids, and so on), government policy (tax policy, credits, incentives, etc) and human behavior interact. Understanding these factors will allow development of better social policies, allow for all levels of government and industry to better prepare for technological changes coming in the near future, and will allow consumers to better understand how certain technologies, policies and behaviors effect their daily lives.

LAPIS will act as a mechanism for coordination within the CDI Multi-Model simulation. Modeling such a wide variety of behaviors, systems, and functions requires a multi-disciplinary team, and a diverse collection of models. Developing all of these different facets as a single, monolithic model is infeasible. At the same time, the

required level of interaction between the models during simulation makes running the models in isolation equally infeasible. Given this, the LAPIS system was chosen as a means of interconnecting these models into a larger multi-model simulation. Initial work has already been done and several successfully multi-model simulations have been built using the LAPIS system to interconnect models. These simulations have included sophisticated household energy consumption models, power system commit and dispatch optimization models, and commercial and industrial power consumption models. These initial simulation setups are a starting point for larger and more sophisticated end-user systems that will be developed as the project progresses. As the overall multi-model simulation grows in size and complexity, the roll of LAPIS will become critical to the systems ability to operate.

To date, two different multi-model systems have been developed, and a third more advanced system is in active development. Each of these systems is described below.

5.5.1 The First Generation CDI Multi-Model: The Utility Provider Simulation Website

The first CDI Multi-Model was designed and built as a proof of concept, showing that a multi-model simulation could be built out of existing models, using LAPIS to interconnect the components. Additionally, the first generation simulation showed that the entire multi-model simulation could be controlled and observed via a web based user interface. For the first generation simulation, the user takes on the roll of a power company that must meet the power needs of a growing community. The user chooses which type of power plant to build, and when to build them. They must consider not only peek demand, but total system efficiency, cost, greenhouse gas (GHG) production, and power system ramping.

This simulations web-based front-end user interface links to models developed as part of the CDI project running on remote computational resources (see figure 5.11. For the first generation simulation, the models used were a Residential Power Demand (RPD) model based on the work in [?], and a Power Optimization Engine (POE) model based on the CPLEX optimization toolbox from IBM. These models underwent minimal modification in order to allow them to work together to provide the relevant output data to the end user. The coordination of the models, directing of input from the website to the models, and directing simulation results back to the website were all accomplished with the LAPIS system though a module referred to as the Back-End Manager (BEM). Specific details of the web-based front end, the component models, and the BEM are discussed below.

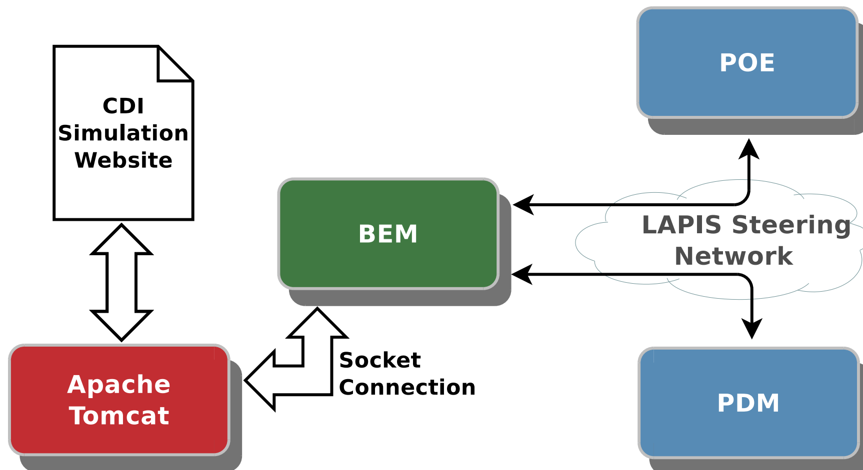

Figure 5.11: The structure of the 1st generation CDI Multi-Model. Note that the BEM and models may all be running on separate physical hardware.

5.5.2 The Website

The web-based user interface (see figure 5.12) is a simple website which gives the user the controls to select how many of each type of power plant to build each year, as well as a control to start a year of simulation. Additionally, the website provides feedback to the player in the form of graphs that show peak demand, critical power spikes, total demand, total generation, population, GHG emissions, cost, and other key factors. The website uses Apache Tomcat to implement much of its functionality, sending input in the form of a JSON file from the user via a socket connection to the BEM. Once input is sent, the Tomcat process then waits, monitoring the same socket, until another JSON file is returned from the BEM, at which point the charts and other data on the website are updated with the data found in the returned JSON file.

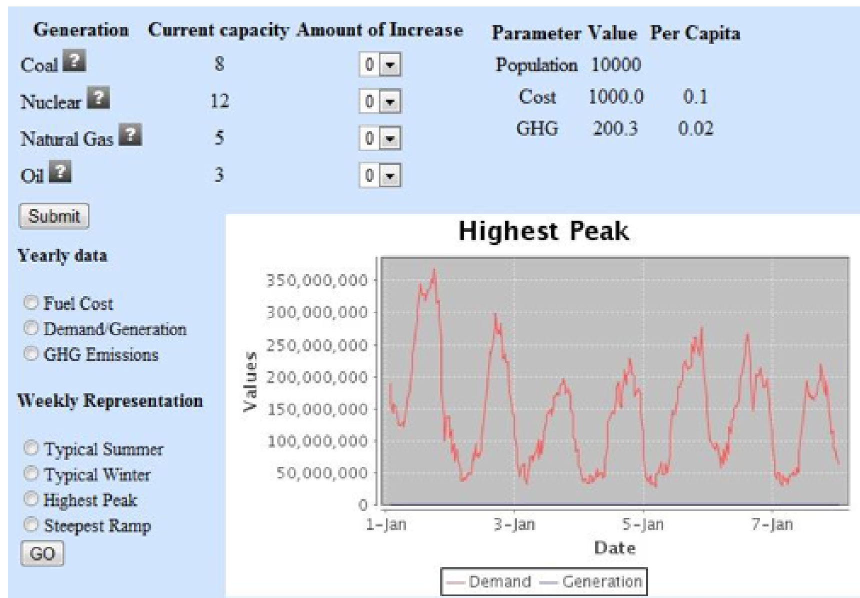

Figure 5.12: User interface web page for the power generation/consumption simulation

5.5.3 The Back End Manager

The Back End Manager was designed to act as an interface between the Apache Tomcat powered front end website and the Java and MATLAB models on the back end of the system. Requests for simulation come from the website, at which point the BEM handles setting of initial values of the simulations, coordination of simulation runs to ensure that data dependencies are met, and collection, packaging, and transmission of outputs back to the front end Tomcat process. Additionally, the BEM creates the steering network that the back end models connect to when they are started up.

The BEM is a good example of the flexibility of the LAPIS system. Many simulation systems have or require components that qualify as neither user interface or simulation. In many computational steering systems, if the architecture of the users simulation doesn't conform to the architecture dictated by the steering system, either the tool could not be used, or there would be significant recoding effort required in order to make the existing simulation 'fit' the steering tool. LAPIS, however, dictates no such architecture, and additionally, is flexible enough to implement any such architecture. As such, LAPIS easily conforms to whatever software architecture the developer might wish to use.

5.5.4 The PDM and POE Models

The first generation simulation consisted of two models, the Power Demand Model (PDM) and the Power Optimization Engine (POE). Both models were initially developed without inclusion of the LAPIS API but both only required slight modification in order to grant the BEM access to their internal state through LAPIS. In both

cases, the models create instances of the LAPIS API and connect to the steering network started by the BEM. Both nodes then publish critical internal state variables, giving access to them to any other node within the steering network, including the BEM. Additionally, both models were modified with the inclusion of a 'run-loop' and a run flag, also published to the steering network. The models have internal state variables that are expected to be set externally by the BEM. When the run flag is set externally, the model begins computation and runs one full year of simulation, filling another output buffer with the resulting data. When the model has completed a years worth of simulation, it once again clears the run flag, signaling to the BEM that computation has completed.

5.5.5 System Operation

The first generation multi-model was a coarse-grain system, performing one full year of simulation at a time, and only upon user request. Additionally, only two models were involved in the simulation, and they were always run sequentially rather than in parallel. As such, a very well defined order of events was in place during the simulation of each year of data. The specific order of events is as follows:

1. The user enters the number of plants to build and presses the 'Run' button.
2. The web interface sends the user's input to the BEM as a JSON file via a local socket connection and then waits for output from the BEM.
3. Using LAPIS, the BEM sets the initial state of the PDM (population size, simulation year) and then sets the run flag within the PDM. The BEM then monitors the run flag, waiting to see it cleared, signaling that the PDM has completed a years worth of simulation.

4. The PDM, having its run flag set, simulates one year and clears the run flag.
5. Seeing the PDM's run flag clear, the BEM uses LAPIS to get the resulting data from the PDM and, again using LAPIS, uses that data to set the initial state of the POE. The BEM then sets the run flag of the POE and monitors the flag, waiting to see it cleared.
6. The POE, having its run flag set, simulates one year and clears the run flag.
7. Seeing the POE's run flag clear, The BEM uses LAPIS to get the output of the POE.
8. Using the output collected from both the PDM and the POE, the BEM writes an output JSON file and sends the data back to the Tomcat process via the same socket it initially received the input from.
9. The web interface displays the new data to the user.

This process is repeated for every year of simulation the end user requests. Given the amount of time taken to simulate one year within any given model, running one year of simulation time was not ideal. Additionally, only ever running models sequentially was also less than ideal. Finally, the initial web interface, while somewhat functional, was far to static and overall did not interface well with the BEM and the rest of the system. All of these issues fed into the design of the second and third generation CDI Multi-Models.

5.5.6 The Second Generation CDI Multi-Model: The Continuously Executing Power Demand And Supply Model

The second generation of the CDI Multi-Model simulation was built upon the first generation design, but includes several redesigns. First and foremost, the design of the overall system was changed of a controlled-rate execution simulation to a continuously executing simulation. The PDM and POE models were modified to accommodate this methodology. Additionally, a Commercial & Industrial (CI) model was added to the overall system. See figure 5.13 for a full diagram of the second generation Multi-Model. All three models were modified to allow direct coordination between models. Details of this mechanism and its implementation are discussed below in section 5.5.6. In addition to this functional change, the initial website was abandoned as it no longer fit with the operational paradigm of the model. In stead, data is accessed on the fly directly from the MATLAB command line.

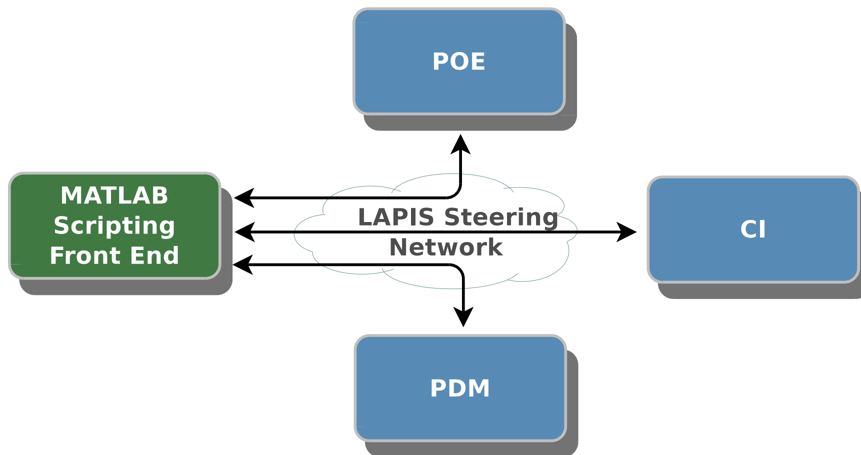

Figure 5.13: The structure of the 2nd generation CDI Multi-Model. Note the lack of static user interface as well as the lack of any central controller.

The Commercial & Industrial Model

The Commercial & Industrial (CI) model provides power demand simulation of commercial and industrial entities that correspond to the population being simulated at the same time by the PDM. The POE was modified to consider the output of the CI model in the optimization calculations. Unlike the PDM and POE, the CI model was developed from the start with the LAPIS API, however the use of LAPIS within the model is nearly the same as LAPIS' use in the PDM model.

Model Manifests and Hassle Free Model Integration

The most interesting part of the second generation Multi-Model simulation is the development of the Model Manifest system. While development of various models was undertaken by different parties, it became quite clear that tight coordination between developers was at least undesirable, and more likely unworkable as system complexity continues to increase. To mitigate this problem, the Model Manifest system was introduced.

Within the Model Manifest system, each developer is responsible for maintaining a manifest for their model. This manifest includes a number of pieces of information. Most critically, it includes a listing of the internal state the simulation publishes, that state's type, size, rate of growth, and other key factors. Correspondingly, the developer of a model is responsible for monitoring the models upon which the developers model is dependent upon for data. In this way, developers can work without significant coordination or worry about changes made to one model breaking the function of other models.

5.5.7 The Third Generation CDI Multi-Model: The CDI Simulation System

The third generation Multi-Model is currently in development, making use of the initial concept for a web-based front end, as well as the idea of a continuously running, non-centrally-coordinated back-end collection of models. Additionally, the system will be set up to support multiple users simultaneously (with separate instances of back-end models for each user). A weather model and transportation model will also be included, once the models are fully developed and tested.

The structure of the third generation Multi-Model (see figure 5.14) again uses a web-based front-end, but in this design, the web front end reads data to display from a database, and writes user input to the same database. Additionally, the layout of the user website will be generated dynamically based on the model manifests, which will be parsed at load-time of the website. In the future, there are additionally plans to provide the user of the site a list of available models, allow the user to connect them into a custom structure dynamically, and start execution of the full system, all with a simple visual interface.

On the back-end, attached to the LAPIS steering network is an Data Coordinator that gathers all the generated states from all the models, and enters all the data into the database. Having a single module within the steering network that has access to all historical data generated by the models also presents an opportunity to easily implement a check-pointing system, as well as a stop and rewind system. By having all the models publish a pause flag and all internal state needed to set up initial conditions, the Data Coordinator can pause all models, read data for a particular time out of the database, and set all the internal states appropriately. Additionally, by

parsing the manifests included with each model, this process can happen procedurally at run-time, and adding additional models or changing existing models will require no additional implementation, so long as manifests are maintained.

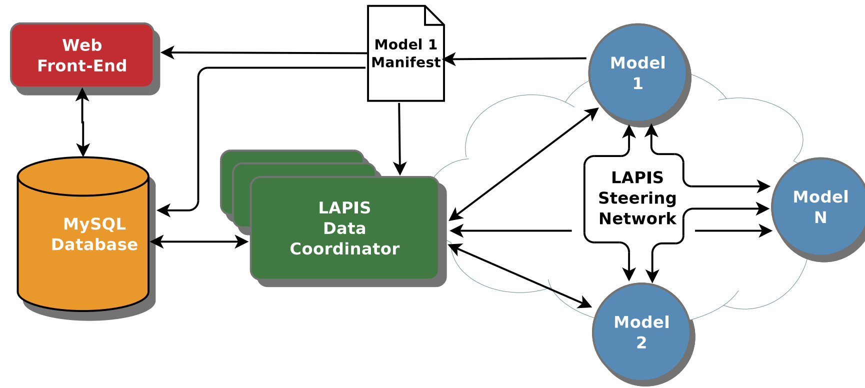

Figure 5.14: The structure of the 2nd generation CDI Multi-Model. Note the lack of static user interface as well as the lack of any central controller.

Chapter 6: Conclusion

Computational steering shows great potential when applied to scientific computing problems. Despite this, limitations in existing technology make creating, maintaining and using such systems difficult, particularly for the typical researcher. This work provides a thorough definition of computational steering and analysis of the entire history and development of the field of computational steering. An analysis of the problems with computational steering, an analysis of other historical developments in other high performance computing fields, and a number of other design considerations are discussed in detail, resulting in a theoretically ideal computational steering architecture. LAPIS has been implemented using this architecture, and a summary of the development itself as well as the implementation details have been presented. Finally, a collection of implementation examples have been presented, demonstrating a range of computational techniques and abilities that LAPIS provides.

6.1 LAPIS' Contribution to Computational Steering

Today, the LAPIS system already provides simple, robust and flexibly computational steering functionality. The API is simple enough to be quickly understood, is easy to setup and use, and should be adoptable by any programmer of minimal experience. Such a simple to use steering system, with the level of flexibility provided by

LAPIS, does not otherwise exist. In addition to being easy to use, the high portability levels provided confidence that time spent developing more complex behavior will not be wasted, and additionally, can be shared with the community at large. Likewise, implementation of larger system components are far more likely to be applicable to the work of others, saving them the trouble of developing the same components over and over again. Additionally, this framework allows for interoperability between languages and models, all while requiring minimal changes to existing code.

The design and implementation of LAPIS addresses in part or in full every problem identified with computational steering systems, as discussed in 4.5. By simplifying development, LAPIS ensures that new users will not be turned back by a high learning curve. Maintainability and portability are both very good in comparison to most other steering tools in the field. This ensures that simulation developers will not risk their simulations developmental health, or limit themselves to a particular system setup, giving developers further confidence in using LAPIS. Additionally, the drafting of the LAPIS standards documents provides a mechanism that is historically shown to increase adoption rates of a given computational technique. The unique architecture of the LAPIS system additionally allows easy expansion by third party developers, helping to ensure that a wide variety of modules and systems can be developed independently, while at the same time all making use of the same underlying technology. LAPIS also provides flexibility to handle issues of data security, while abstracting away issues of data compatibility and to a large extent, computational scaling.

6.2 Future Work

There are a number of extensions and improvements planned to the LAPIS framework. While the core LAPIS systems are currently functional and already provide a wide variety of benefits, there are a number of features that should still be added. Tracking and confirming messages has not yet been implemented, nor has error control for dropped, lost, or mishandled messages. Additional information about published variables, including dimension and type, will in the future be tracked in order to simplify the API. Also, in the future it will be possible to GET and SET subsections of a shared data array. Finally, additional synchronization modes may be added to allow nodes to 'subscribe' to a remote data object, which would cause updates to that remote data object to be pushed to the subscribing node. In addition to these enhancements of the core software, additional APIs are planned, including C/C++, Python, and FORTRAN. Also, additional communicators will be implemented, including a File-based communicator for use on systems with more restrictive network setups.

At this point, LAPIS shows great promise as a computational tool. LAPIS is unique among steering tools and frameworks in its ease of use, flexibility, and robustness. With further work, more APIs can be developed, allowing more simulations in different languages to leverage the benefits of computational steering. Additional COM modules will likewise increase the number of ways and locations that LAPIS can be used on. Further work on the infrastructure of the tool will provide even more robustness, a fuller feature set for greater efficiency in use, and new features and possibilities. Through additional use in a variety of areas, functional requirements and abilities will be further refined into a truly universal computational steering standard.

Appendix A: Implementation Example Code Bodies

A.1 Conways Game of Life, Steered Version Code

A.1.1 Game of Life Code

```
function life(action)
%LIFE   MATLAB's version of Conway's Game of Life.
%   "Life" is a cellular automaton invented by John
%   Conway that involves live and dead cells in a
%   rectangular, two-dimensional universe. In
%   MATLAB, the universe is a sparse matrix that
%   is initially all zero.
%
%   Whether cells stay alive, die, or generate new
%   cells depends upon how many of their eight
%   possible neighbors are alive. By using sparse
%   matrices, the calculations required become
%   astonishingly simple. We use periodic (torus)
%   boundary conditions at the edges of the
%   universe. Pressing the "Start" button
%   automatically seeds this universe with several
%   small random communities. Some will succeed
%   and some will fail.

%   C. Moler, 7-11-92, 8-7-92.
%   Adapted by Ned Gulley, 6-21-93
%   Copyright 1984-2004 The MathWorks, Inc.
%   $Revision: 5.10.4.1 $  $Date: 2004/08/16 01:38:30 $

addpath('/home/ben/workspace/LAPIS_API_Matlab');

play= 1;
stop=-1;

if nargin<1,
   action='initialize';
end;
```

```
if strcmp(action,'initialize'),
   figNumber=figure( ...
      'Name','Life: Conway''s Game of Life', ...
      'NumberTitle','off', ...
      'DoubleBuffer','on', ...
      'Visible','off', ...
      'Color','white', ...
      'BackingStore','off');
   axes( ...
      'Units','normalized', ...
      'Position',[0.05 0.05 0.75 0.90], ...
      'Visible','off', ...
      'DrawMode','fast', ...
      'NextPlot','add');

   text(0,0,'Press the "Start" button to see the Game of Life demo', ...
      'HorizontalAlignment','center');
   axis([-1 1 -1 1]);

   %===================================
   % Information for all buttons
   labelColor=[0.8 0.8 0.8];
   yInitPos=0.90;
   xPos=0.85;
   btnLen=0.10;
   btnWid=0.10;
   % Spacing between the button and the next command's label
   spacing=0.05;

   %====================================
   % The CONSOLE frame
   frmBorder=0.02;
   yPos=0.05-frmBorder;
   frmPos=[xPos-frmBorder yPos btnLen+2*frmBorder 0.9+2*frmBorder];
   h=uicontrol( ...
      'Style','frame', ...
      'Units','normalized', ...
      'Position',frmPos, ...
      'BackgroundColor',[0.50 0.50 0.50]);

   %====================================
   % The START button
   btnNumber=1;
   yPos=0.90-(btnNumber-1)*(btnWid+spacing);
   labelStr='Start';
   cmdStr='start';
   callbackStr='life(''start'');';

   % Generic button information
   btnPos=[xPos yPos-spacing btnLen btnWid];
```

```
startHndl=uicontrol( ...
   'Style','pushbutton', ...
   'Units','normalized', ...
   'Position',btnPos, ...
   'String',labelStr, ...
   'Interruptible','on', ...
   'Callback',callbackStr);

%====================================
% The STOP button
btnNumber=2;
yPos=0.90-(btnNumber-1)*(btnWid+spacing);
labelStr='Stop';
% Setting userdata to -1 (=stop) will stop the demo.
callbackStr='set(gca,''Userdata'',-1)';

% Generic button information
btnPos=[xPos yPos-spacing btnLen btnWid];
stopHndl=uicontrol( ...
   'Style','pushbutton', ...
   'Units','normalized', ...
   'Position',btnPos, ...
   'Enable','off', ...
   'String',labelStr, ...
   'Callback',callbackStr);

%====================================
% The INFO button
labelStr='Info';
callbackStr='life(''info'')';
infoHndl=uicontrol( ...
   'Style','push', ...
   'Units','normalized', ...
   'Position',[xPos 0.20 btnLen 0.10], ...
   'String',labelStr, ...
   'Callback',callbackStr);

%====================================
% The CLOSE button
labelStr='Close';
callbackStr='close(gcf)';
closeHndl=uicontrol( ...
   'Style','push', ...
   'Units','normalized', ...
   'Position',[xPos 0.05 btnLen 0.10], ...
   'String',labelStr, ...
   'Callback',callbackStr);

% Uncover the figure
hndlList=[startHndl stopHndl infoHndl closeHndl];
set(figNumber,'Visible','on', ...
```

```
        'UserData',hndlList);

elseif strcmp(action,'start'),
    cla;
    axHndl=gca;
    figNumber=gcf;
    hndlList=get(figNumber,'Userdata');
    startHndl=hndlList(1);
    stopHndl=hndlList(2);
    infoHndl=hndlList(3);
    closeHndl=hndlList(4);
    set([startHndl closeHndl infoHndl],'Enable','off');
    set(stopHndl,'Enable','on');

    % ====== Start of Demo
    set(axHndl, ...
        'UserData',play, ...
        'DrawMode','fast', ...
        'Visible','off');
    m = 101;
    %---HBS addition: LAPISS init, data init------------------------
    % create steering api instance
    lapi = LapissApi('GameOfLife');
    % init. steering network
    lapi.Initalize();
    % init of steered variables
    X = LapissData('field', zeros(m,m));
    pause_flag = LapissData('pause', [0 0]);
    % publish steered variables
    lapi.Publish('field', X, 0);
    lapi.Publish('pause', pause_flag, 0);
    %---------------------------------------------------------------

    p = -1:1;
    for count=1:15,
        kx=floor(rand*(m-4))+2; ky=floor(rand*(m-4))+2;
        X.d(kx+p,ky+p)=(rand(3)>0.5);
    end;

    % The following statements plot the initial configuration.
    % The "find" function returns the indices of the nonzero elements.
    [i,j] = find(X.d);
    figure(gcf);
    plothandle = plot(i,j,'.', ...
        'Color','blue', ...
        'MarkerSize',12);
    axis([0 m+1 0 m+1]);

    % Whether cells stay alive, die, or generate new cells depends
    % upon how many of their eight possible neighbors are alive.
    % Here we generate index vectors for four of the eight neighbors.
```

```
   % We use periodic (torus) boundary conditions at the edges of
   % the universe.

   n = [m 1:m−1];
   e = [2:m 1];
   s = [2:m 1];
   w = [m 1:m−1];

   while get(axHndl,'UserData')==play,
      % HBS addition: check if paused
      if(pause_flag.d(1) == 0)
          % How many of eight neighbors are alive.
          N = X.d(n,:) + X.d(s,:) + X.d(:,e) + X.d(:,w) + ...
             X.d(n,e) + X.d(n,w) + X.d(s,e) + X.d(s,w);

          % A live cell with two live neighbors, or any cell with three
          % neigbhors, is alive at the next time step.
          X.d = double((X.d & (N == 2)) | (N == 3));

          % Update plot.
          [i,j] = find(X.d);
          set(plothandle,'xdata',i,'ydata',j)
          drawnow

          % Bail out if the user closed the figure.
          if ¬ishandle(startHndl)
              return
          end
          pause(0.25);
          % HBS addition: slow down the simulation
      else
        pause(0.25);
        % HBS addition: wait until pause flag is clear
      end

   end

   % ====== End of Demo
   set([startHndl closeHndl infoHndl],'Enable','on');
   set(stopHndl,'Enable','off');

elseif strcmp(action,'info');
   helpwin(mfilename);

end;    % if strcmp(action, ...
```

A.1.2 Steering GUI Code

```
function varargout = guilife(varargin)

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @guilife_OpeningFcn, ...
                   'gui_OutputFcn',  @guilife_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT

function guilife_OpeningFcn(hObject, eventdata, handles, varargin)

% Choose default command line output for guilife
handles.output = hObject;

addpath('/home/ben/workspace/LAPIS_API_Matlab');
% Update handles structure
guidata(hObject, handles);

% UIWAIT makes guilife wait for user response (see UIRESUME)
% uiwait(handles.figure1);


function varargout = guilife_OutputFcn(hObject, eventdata, handles)

% Get default command line output from handles structure
varargout{1} = handles.output;


function pushbutton1_Callback(hObject, eventdata, handles)
    % create the lapi object
    lapi = LapissApi('GameOfLifeClient');

    % connect to the address provided
    lapi.Connect('GameOfLife', get(handles.edit1, 'String'));

    % save the lapi instance in handles
    setappdata(0, 'lapi', lapi);
```

```
function edit1_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'), ...
 get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


function pushbutton2_Callback(hObject, eventdata, handles)
    lapi = getappdata(0, 'lapi');
    if (strcmp(get(hObject, 'String'),'Pause'))
        % send the 'pause' command to the GoL Server
        lapi.Set('pause', 'GameOfLife', [1 0]);
        % get the current field from GoL Server
        field = lapi.Get('field', 'GameOfLife');
        setappdata(0, 'field', field);
        % display field in axes
        [i,j] = find(field);
        q = get(handles.axes1, 'ButtonDownFcn');
        plot(handles.axes1,i,j,'.');
        set(handles.axes1, 'ButtonDownFcn', q);
        % change button to 'resume' mode
        set(handles.pushbutton2, 'String', 'Resume');
    else
    % if 'resume' mode:
        % send the field to GoL Server
        field = getappdata(0, 'field');
        lapi.Set('field', 'GameOfLife', field);
        % send 'resume' command to GoL Server
        lapi.Set('pause', 'GameOfLife', [0 0]);
        % change button to 'pause' mode
        set(handles.pushbutton2, 'String', 'Pause');
    end


function axes1_ButtonDownFcn(hObject, eventdata, handles)
  % sanitize the point
  point = get(hObject, 'CurrentPoint');
  x = round(point(1,1));
  y = round(point(1,2));

  % add it to field
  field = getappdata(0, 'field');
  field(x,y) = 1;
  setappdata(0, 'field', field);

  % redraw the chart
  [i,j] = find(field);
  q = get(handles.axes1, 'ButtonDownFcn');
  plot(handles.axes1,i,j,'.');
  set(handles.axes1, 'ButtonDownFcn', q);
```

A.2 Hill Climbing Code

A.2.1 Hill Climbing Manager

```
function varargout = director(varargin)

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @director_OpeningFcn, ...
                   'gui_OutputFcn',  @director_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


function director_OpeningFcn(hObject, eventdata, handles, varargin)

    % Choose default command line output for director
    handles.output = hObject;

    % Update handles structure
    guidata(hObject, handles);

    % UIWAIT makes director wait for user response (see UIRESUME)
    % uiwait(handles.figure1);

    addpath('/home/ben/workspace/LAPIS_API_Matlab');

    % create an instance of the LAPIS API
    lapi = LapissApi('Director');
    setappdata(0, 'lapi', lapi);
    lapi.Initalize();

    % create the data field
    field = LapissData('field', rsgeng2D(200,100,30,20));
    setappdata(0, 'field', field);

    % publish the data field
```

```
    lapi.Publish('field', field, 0);

    % set default climber name
    setappdata(0, 'selectedClimber', 'none');

    % update the director info
    set(handles.text_name, 'String', 'Director');
    set(handles.text_address, 'String', lapi.GetAddress);

    % adjust the color map
    h = gcf;
    map = get(h,'Colormap')
    map = map(22:end,:);
    set(h,'Colormap',map);

    % update the graph with the mesh
    mesh(field.d);

    % start the polling timer
    %start(handles.updateTimer);


function varargout = director_OutputFcn(hObject, eventdata, handles)
    varargout{1} = handles.output;




function listbox_climbers_Callback(hObject, eventdata, handles)

    lapi = getappdata(0, 'lapi');
    field = getappdata(0, 'field');

    % set 'selected climber' name
    contents = cellstr(get(hObject,'String'));
    selectedClimber = contents{get(hObject,'Value')};
    setappdata(0, 'selectedClimber', selectedClimber);

    % download all information to display from selected climber
    path = lapi.Get('path', selectedClimber)
    state = lapi.Get('state', selectedClimber)
    % state translation:
    % 0 -   reset: download field, generate random point, then climb
    % 1 -   climb: inspect field, climb hill
    % 2 -   pert: move by random vector then climb
    % 3 -   peak: local maximum, don't wast CPU cycles, just sit
    % 4 -   pause: paused by controller. Do nothing
    % 5 -   exit: set the exit flag on the main loop and terminate

    % set pause/unpause button as appropriate for pause state
    if (state == 4)
```

```
        set(handles.pushbutton_pause, 'String', 'Resume');
    else
        set(handles.pushbutton_pause, 'String', 'Pause');
    end

    % display climb path
    hold off;
    mesh(field.d);
    hold on;
    plot3(path(:,2), path(:,1), path(:,3), 'b+');

    % display everything else
    if (state == 0)
        set(handles.text_state, 'String', 'Resetting!');
    elseif (state == 1)
        set(handles.text_state, 'String', 'Climbing');
    elseif (state == 2)
        set(handles.text_state, 'String', 'Perturbed!');
    elseif (state == 3)
        set(handles.text_state, 'String', 'Peaked');
    elseif (state == 4)
        set(handles.text_state, 'String', 'Paused');
    elseif (state == 5)
        set(handles.text_state, 'String', 'Exiting!');
    else
        set(handles.text_state, 'String', 'Unknown!!!');
    end


function listbox_climbers_CreateFcn(hObject, eventdata, handles)
    if ispc && isequal(get(hObject,'BackgroundColor'), ...
     get(0,'defaultUicontrolBackgroundColor'))
        set(hObject,'BackgroundColor','white');
    end


function pushbutton_pause_Callback(hObject, eventdata, handles)
    lapi = getappdata(0, 'lapi');
    selectedClimber = getappdata(0, 'selectedClimber');

    % get 'selected climber' name

    % if unpaused
    if (strcmp(get(handles.pushbutton_pause, 'String'), 'Pause'))

        % set state of climber to 'pause'
        lapi.Set('state', selectedClimber, 4);

        % set button to unpause
        set(handles.pushbutton_pause, 'String', 'Resume');
```

```
    else

        % set state of climber to 'climbing'
        lapi.Set('state', selectedClimber, 1);

        % set button to pause
        set(handles.pushbutton_pause, 'String', 'Pause');

    end


function pushbutton_perturb_Callback(hObject, eventdata, handles)

    lapi = getappdata(0, 'lapi');
    selectedClimber = getappdata(0, 'selectedClimber');

    % tell the climber to generate a perturbance
    lapi.Set('state', selectedClimber, 2);


function pushbutton_reset_Callback(hObject, eventdata, handles)

    lapi = getappdata(0, 'lapi');
    selectedClimber = getappdata(0, 'selectedClimber');

    % set state to 'reset' -- climber will check back, redownload, and
    % restart climbing automatically
    lapi.Set('state', selectedClimber, 0);


function pushbutton_newfield_Callback(hObject, eventdata, handles)
    lapi = getappdata(0, 'lapi');
    field = getappdata(0, 'field');

    % generate a new data field
    field.d = rsgeng2D(200,100,30,20);

    % loop though all clients
    climbers = cellstr(get(handles.listbox_climbers,'String'));
    for i = 1:length(climbers)
        % set state to 'reset'
        lapi.Set('state', climbers{i}, 0);
    end


function pushbutton_refresh_Callback(hObject, eventdata, handles)

    lapi = getappdata(0, 'lapi');

    % get list of attached nodes
    nodeList = lapi.GetNodeList();
```

```
    %nodes = strtok(nodeList, ',');
    nodes = textscan(nodeList, '%s', 'Delimiter', ',');
    nodes = nodes{1};

    % trim ourselves from the list
    [¬,loc] = max(cellfun(@(fun_x) (strcmp(fun_x, 'Director')), nodes));

    nodes_temp1 = {};
    nodes_temp2 = {};
    if(loc > 1)
        nodes_temp1 = nodes(1:loc-1);
    end
    if(loc ≠ length(nodes))
        nodes_temp2 = nodes(loc+1:end);
    end
    nodes = [nodes_temp1 nodes_temp2];

    % populate the list box with the node list
    set(handles.listbox_climbers, 'String', nodes);
```

A.2.2 Matlab Hill Climber

```
function climber(directorAddress)
addpath('/home/ben/workspace/LAPIS_API_Matlab');

% initalize base state
% states:
% 0 --   reset: download field, generate random point, then climb
% 1 --   climb: inspect field, climb hill
% 2 --   pert: generate random vector and move to that point, then climb
% 3 --   peak: local maximum, don't wast CPU cycles, just sit
% 4 --   pause: paused by controller. Do nothing
% 5 --   exit: set the exit flag on the main loop and terminate
state = LapissData('state', 0);

% the path the climber has taken. current location is at (0,0,:)
path = LapissData('path', zeros(100,3));

% exit flag
exit = 0;

% local copy of the field we will climb. The actual data will sit within
% a 200x200 buffered region
field = -1*inf(202,202);


% create instance of LAPIS
id = floor(1000000000*rand());
```

```
lapi = LapissApi(['climber_' num2str(id)]);

% publish our path
lapi.Publish('state', state, 0);
lapi.Publish('path', path, 0);

% connect to the network
lapi.Connect('Director', directorAddress);

% let LAPIS connect and settle
pause(1);

% enter execution loop
while (¬exit)
    disp(['current stat is ' num2str(state.d)]);
    if (state.d == 0)
        disp('Climber in state 0');
        % clear the path
        path.d = zeros(100,3);

        % download field from director
        field(2:201,2:201) = lapi.Get('field', 'Director');

        % generate random starting point. +1s in z calc account for −inf
        % buffer
        path.d(1,1) = floor(200*rand());
        path.d(1,2) = floor(200*rand());
        path.d(1,3) = field(path.d(1,1)+1,path.d(1,2)+1);

        % set state.d to climb
        state.d = 1;
    elseif (state.d == 1)
        disp('Climber in state 1');
        % get the current point. +1s account for −inf buffer
        Xc = path.d(1,1)+1;
        Yc = path.d(1,2)+1;

        % get surrounding points
        set = field(Xc−1:Xc+1,Yc−1:Yc+1);

        % find the biggest point around our point
        [t ypos] = max(max(set,[],1));
        [t xpos] = max(max(set,[],2));

        ydif = ypos−2;
        xdif = xpos−2;

        % if the biggest point is the middle, we are at a local max
        if (xdif==0 && ydif==0)
            % set state.d to peaked to save CPU cycles
            state.d = 3;
```

```
        else
            % shift the path down 1 space
            path.d(2:end,:) = path.d(1:end-1,:);
            % add the new end point
            path.d(1,1) = path.d(2,1) + xdif;
            path.d(1,2) = path.d(2,2) + ydif;
            path.d(1,3) = field(path.d(1,1)+1, path.d(1,2)+1);
        % +1s account for -inf buffer
        end

        % add artificial delay to simulation to represent something more
        % significant
        pause(0.5);

    elseif (state.d == 2)
        disp('Climber in state 2');
        % generate a perturbance
        dx = floor(20*rand())-10;
        dy = floor(20*rand())-10;

        % shift the path
        path.d(2:end,:) = path.d(1:end-1,:);

        % jump in x!
        path.d(1,1) = path.d(2,1) + dx;
        if (path.d(1,1) < 1)
            path.d(1,1) = 1;
        elseif (path.d(1,1) > 200)
            path.d(1,1) = 200;
        end

        % jump in y!
        path.d(1,2) = path.d(2,2) + dy;
        if (path.d(2,1) < 1)
            path.d(2,1) = 1;
        elseif (path.d(2,1) > 200)
            path.d(2,1) = 200;
        end

        % find new height
        path.d(1,3) = field(path.d(1,1)+1, path.d(1,2)+1);

        % set next state.d to climbing
        state.d = 1;

        % simulate this being hard
        pause(0.5);

    elseif (state.d == 3)
        disp('Climber in state 3');
        % just sit forever
```

```
        pause(2);

    elseif (state.d == 4)
        disp('Climber in state 4');
        % just sit until woken
        pause(0.5);

    elseif (state.d == 5)
        disp('Climber in state 5');
        % clean up LAPIS
        % set exit flag
        exit = 1;
    else
        % report error
        disp('Error: climber entered unknown state!');
        % clean up LAPIS
        % set exit flag
        exit = 1;
    end

end
```

A.2.3 Java Hill Climber

```
import java.io.IOException;
import java.util.Arrays;
import edu.osc.LAPIS.API.Java.LapissJavaApi;

public class HillClimber {

  public static void main(String[] args) throws IOException,
    InterruptedException {
      // initalize base state
      // states:
      // 0 -   reset: download field, and climb
      // 1 -   climb: inspect field, climb hill
      // 2 -   pert: move by random vector then climb
      // 3 -   peak: local maximum, don't wast CPU cycles, just sit
      // 4 -   pause: paused by controller. Do nothing
      // 5 -   exit: set the exit flag on the main loop and terminate

      // the state of the climber
      // declared as an array to allow it to be shared by LAPIS
      Integer[] state = new Integer[1];
      state[0] = 0;
      Integer[] SizeOfScalar = new Integer[1];
```

```
      SizeOfScalar[0] = 1;

      // the path the climber has taken. current location is at (0,:)
      Double[][] path = new Double[100][3];
      Integer[] SizeOfPath = new Integer[2];
      SizeOfPath[0] = 100;
      SizeOfPath[1] = 3;

      // exit flag
      Integer exit = 0;

      // local copy of the field we will climb.
      Object[][] field = new Double[200][200];

      // create instance of LAPIS
      Long id = Math.round(1000000000*Math.random());
      LapissJavaApi lapi = new LapissJavaApi("climber_"
+ id.toString());

      // publish the state and path
      lapi.Publish("path", path, LapissJavaApi.DOUBLE, 2,
SizeOfPath, 0);
      lapi.Publish("state", state, LapissJavaApi.INTEGER, 1,
SizeOfScalar, 0);

// connect to the network
lapi.Connect("Director", args[0]);

// let LAPIS connect and settle
Thread.sleep(1000);

// enter execution loop
while (exit == 0)
{
  if (state[0] == 0)
  {
    // clear the path out;
    for(int i=0; i<100; i++)
    {
      path[i][0] = 0.0;
      path[i][1] = 0.0;
      path[i][2] = 0.0;
    }
```

```
    // download the field from the director
    field = (Object[][]) lapi.Get("field", "Director");

    // generate random starting point.
    path[0][0] = 200*Math.random();
    path[0][1] = 200*Math.random();
    path[0][2] = (Double)field[(int)Math.round(
      path[0][0])+1][(int)Math.round(path[0][1])+1];

    // change state to climb
    state[0] = 1;
  }
  else if (state[0] == 1)
  {
    // get the current point. +1s account for -inf buffer
    int Xc = (int)Math.round(path[0][0]) + 1;
    int Yc = (int)Math.round(path[0][1]) + 1;

    // make some diff variables
    int Xdiff = 0;
    int Ydiff = 0;

    // current high
    Double Zc = (Double)field[Xc][Yc];

    // check the 8 neighbors for the high point
    if (Xc > 0)
    {
      // check -1,-1
      if (Yc > 0)
{
if ((Double)field[Xc-1][Yc-1] > Zc)
  {
  Zc = (Double)field[Xc-1][Yc-1];
  Xdiff = -1;
  Ydiff = -1;
  }
}
      // check -1,0
      if ((Double)field[Xc-1][Yc] > Zc)
{
  Zc = (Double)field[Xc-1][Yc];
```

```
  Xdiff = -1;
  Ydiff = 0;
}
      // check -1,+1
      if (Yc < 200)
{
if ((Double)field[Xc-1][Yc+1] > Zc)
  {
  Zc = (Double)field[Xc-1][Yc+1];
  Xdiff = -1;
  Ydiff = 1;
  }
}
      }
    // check 0,-1
    if (Yc > 0)
      {
      if ((Double)field[Xc][Yc-1] > Zc)
{
Zc = (Double)field[Xc][Yc-1];
Xdiff = 0;
Ydiff = -1;
}
      }
    // check 0,1
    if (Yc < 200)
      {
      if ((Double)field[Xc][Yc+1] > Zc)
{
Zc = (Double)field[Xc][Yc+1];
Xdiff = 0;
Ydiff = 1;
}
      }
    if (Xc < 200)
      {
// check 1,-1
if (Yc > 0)
  {
  if ((Double)field[Xc+1][Yc-1] > Zc)
    {
    Zc = (Double)field[Xc+1][Yc-1];
    Xdiff = 1;
```

```
    Ydiff = -1;
    }
  }
// check 1,0
if ((Double)field[Xc+1][Yc] > Zc)
  {
  Zc = (Double)field[Xc+1][Yc];
  Xdiff = 1;
  Ydiff = 0;
  }
// check 1,1
if (Yc < 200)
  {
  if ((Double)field[Xc+1][Yc+1] > Zc)
    {
    Zc = (Double)field[Xc+1][Yc+1];
    Xdiff = 1;
    Ydiff = 1;
    }
  }
}
      // if the biggest point is the middle, we are at a local max
      if ((Xdiff == 0) && (Ydiff == 0))
{
state[0] = 3;
}
      else
{
// shift the path down 1 space
for(int i=99; i>0; i--)
  {
  path[i][0] = path[i-1][0];
  path[i][1] = path[i-1][1];
  path[i][2] = path[i-1][2];
  }
// add the new end point to the path
path[0][0] = path[1][0] + Xdiff;
path[0][1] = path[1][1] + Ydiff;
path[0][2] = Zc;
      }

    Thread.sleep(500);
    }
```

```
else if (state[0] == 2)
  {
  // generate a perturbance
  int dx = (int)Math.round(20*Math.random())-10;
  int dy = (int)Math.round(20*Math.random())-10;

  // shift the path down 1 space
  for(int i=99; i>0; i--)
    {
    path[i][0] = path[i-1][0];
    path[i][1] = path[i-1][1];
    path[i][2] = path[i-1][2];
    }

  // jump in x!
  path[0][0] = path[1][0] + dx;
  // check that we are in bounds
  if (path[0][0] < 0)
    {
    path[0][0] = 0.0;
    }
  else if (path[0][0] > 200)
    {
    path[0][0] = 200.0;
    }

  // jump in y!
  path[0][1] = path[1][1] + dy;
  // check that we are in bounds
  if (path[0][1] < 0)
    {
    path[0][1] = 0.0;
    }
  else if (path[0][1] > 200)
    {
    path[0][1] = 200.0;
    }

  // find new height.
  path[0][2] = (Double)field[(int)Math.round(
    path[0][0])+1][(int)Math.round(path[0][1])+1];

  // set next state to climbing
```

```
    state[0] = 1;

    // simulate this being hard
    Thread.sleep(500);
    }
  else if (state[0] == 3)
    {
    // just sit forever
    Thread.sleep(2000);
    }
  else if (state[0] == 4)
    {
    // just sit until woken
    Thread.sleep(500);
    }
  else if (state[0] == 5)
    {
    // clean up LAPIS
    // set exit flag
    exit = 1;
    }
  else
    {
    // report error
    System.out.println("Error: climber entered unknown state!\n");
    // clean up LAPIS
    // set exit flag
    exit = 1;
    }
  }
}
}
```

A.3 Sailboat Model Code

A.3.1 Sailboat Code

```
function Boat( frontAddress )
%BOAT Summary of this function goes here
%   Detailed explanation goes here
```

```
% setup environment
addpath('/home/ben/workspace/LAPIS_API_Matlab');

% create lapis instnace
lapi = LapissApi('Boat');

% connect to front
%lapi.Connect('Front', '127.0.0.1:48237');

% create shared state
% the current position of the boat
position = LapissData('position', [10.1 10.1]);
% the last movement vector of the boat
vector = LapissData('vector', [0 0]);

% publish shared state
lapi.Publish('position', position, 0);
lapi.Publish('vector', vector, 0);

% wait until 'Wind' and 'Water' show up in node list
ready = 0;
while (¬ready)
    % once there are 2 commas, start running
    list = lapi.GetNodeList();
    if (length(strfind(list,',')) == 2)
        ready = 1;
    end
end

disp('here we go!');

% create a position buffer
posbuf = [position.d];

% create a grid to use in plotting
[X,Y] = meshgrid(1:1:21);

figure

% start simulation loop
while (true)

    % read Wind and water fields
    wind = lapi.Get('field', 'Wind');
    water = lapi.Get('field', 'Water');

    % get vectors for my location
    subwind = wind(floor(position.d(1)):ceil(position.d(1)), ...
  floor(position.d(2)):ceil(position.d(2)),:);
    subwater = water(floor(position.d(1)):ceil(position.d(1)), ...
  floor(position.d(2)):ceil(position.d(2)),:);
```

```
        % average the vectors and create the weighted sum vector
        subwind = mean(mean(subwind));
        subwater = mean(mean(subwater));
        both = 2*subwind + subwater;

        % use 25% of the previous vector and 75% of the new vector to
        % create the next vector
        newV = [both(1,1,1), both(1,1,2)];
        oldV = vector.d;
        vector.d = 0.5 * newV + 0.5 * oldV;
        vector.d = vector.d / sqrt(vector.d(1)^2 + vector.d(2)^2);

        % adjust the position by nextV
        position.d = position.d + vector.d;

        % check for overflow and wrap
        if (position.d(1) > 20)
            position.d(1) = 2.1;
        elseif (position.d(1) < 2)
            position.d(1) = 19.9;
        end

        if (position.d(2) > 20)
            position.d(2) = 2.1;
        elseif (position.d(2) < 2)
            position.d(2) = 19.9;
        end

        % update the position vector
        posbuf = [posbuf ; position.d];

        % plot the position
        cla;
        hold on
        quiver(X,Y,water(:,:,1),water(:,:,2))
        quiver(X,Y,wind(:,:,1),wind(:,:,2))
        plot(posbuf(:,1), posbuf(:,2), 'r');

        % pause
        pause(.25);

    end

end
```

A.3.2 Water Feild Code

```
function Water( frontAddress)
%WATER Summary of this function goes here
%   Detailed explanation goes here

    % setup environment
    addpath('/home/ben/workspace/LAPIS_API_Matlab');

    % create lapis instnace
    lapi = LapissApi('Water');

    % connect to front
    lapi.Connect('Boat', frontAddress);

    % create shared state
    field = LapissData('field', zeros(21,21,2));

    % publish shared state
    lapi.Publish('field', field, 0);

    % generate random init. cond.
    dx = 2 * pi * rand(1,1);
    dy = 2 * pi * rand(1,1);

    % start simulation loop
    while(true)

        % generate a vector field
        field.d = fieldFunction(dx, dy);

        % move the field slightly
        dx = dx + (pi/32) * rand(1,1);
        dy = dy + (pi/32) * rand(1,1);

        % pause
        pause(1);


    end

end

function field = fieldFunction(dx, dy)

    field = zeros(21,21,2);

    [X,Y] = meshgrid(-2:.2:2);

    Z = sin((X+dx).^2) + cos(X+dx).^2 + sin((X+dx)*(Y+dy)) + sin(Y);

    [DX,DY] = gradient(Z,.2,.2);
```

```
    field(:,:,1) = DX./max(max(DX));
    field(:,:,2) = DY./min(min(DY));

    quiver(X,Y,DX,DY);

end
```

A.3.3 Wind Feild Code

```
function Wind( frontAddress )
%WIND Summary of this function goes here
%   Detailed explanation goes here

    % setup environment
    addpath('/home/ben/workspace/LAPIS_API_Matlab');

    % create lapis instnace
    lapi = LapissApi('Wind');

    % connect to front
    lapi.Connect('Boat', frontAddress);

    % create shared state
    field = LapissData('field', zeros(21,21,2));

    % publish shared state
    lapi.Publish('field', field, 0);

    % generate random init. cond.
    dx = 2 * pi * rand(1,1);
    dy = 2 * pi * rand(1,1);

    % start simulation loop
    while(true)

        % generate a vector field
        field.d = fieldFunction(dx, dy);

        % move the field slightly
        dx = dx + (pi/32) * rand(1,1);
        dy = dy + (pi/32) * rand(1,1);

        % pause
        pause(1);

    end

end
```

```
function field = fieldFunction(dx, dy)

    field = zeros(21,21,2);

    [X,Y] = meshgrid(−2:.2:2);

    Z = sin((X+dx).^2) + cos(X+dx).^2 + sin((X+dx)*(Y+dy)) + sin(Y) ...
      + sin((Y+dy)^2);

    [DX,DY] = gradient(Z,.2,.2);

    field(:,:,1) = DX./max(max(DX));
    field(:,:,2) = DY./max(max(DY));

    quiver(X,Y,DX,DY);

end
```

A.4 Driver-Engine Model Code

A.4.1 Driver Code

```
function NormalDriver()

addpath('/home/ben/workspace/LAPIS_API_Matlab_working_copy/');

% initalize variables and LAPIS, speed limit file
myName = 'NormalDriver';
engineName = '';
lapi = LapissApi(myName);
speedlimits = csvread('speeds.csv');

% wait for the engine to show up
enginePresent = false;
list = '';
while (¬enginePresent)
    % if there is no comma, then the list is just us still.
    list = lapi.GetNodeList();
    enginePresent = ¬isempty(strfind(list,','));
end

% parse the list into names
[n1 n2] = strtok(list,',');
n2 = n2(2:end);

% if our name is n1, then n2 is the engine
```

```
if (strcmp(n1, myName) == 0)
    engineName = n1;
else
    engineName = n2;
end

% start the engine!
lapi.Set('Start', engineName, [1 0]);

% start 'driving' (loop)
road = true;
step = 0;
limit = 0;
speed = 0;
pedal = 0;

limitbuf = [];
speedbuf = [];
pedalbuf = [];

while (road)

    % get current speed limit every 10 steps
    if ( floor(step/10) == (step/10) )
        % if there is a speed left
        if (isempty(speedlimits))
            % if no more speeds, set road = 0
            road = 0; % limit remains the same
        else
            % read a speed limit in
            limit = speedlimits(1);
            speedlimits = speedlimits(2:end);
        end
    end

    % get current speed and pedal postion from engine
    speed = lapi.Get('Speed', engineName);
    pedal = lapi.Get('Pedal', engineName);

    % apply accellerator
    if (speed(1) > limit)
        pedal(1) = 0;
    elseif (speed(1) > limit * 0.9)
        pedal(1) = min(1, 0.9 * pedal(1));
    elseif (speed(1) > 0.5 * limit)
        pedal(1) = min(1, pedal(1) + 0.1);
    elseif (speed(1) < limit)
        pedal(1) = min(1, pedal(1) + 0.25);
    else
        pedal(1) = pedal(1);
    end
```

```
    lapi.Set('Pedal', engineName, pedal);

    % report to terminal
    disp('============================');
    disp(['Speed Limit:    ' num2str(limit)]);
    disp(['Current Speed:  ' num2str(speed(1))]);
    disp(['Pedal Position: ' num2str(pedal(1))]);

    % buffer for graphing
    limitbuf = [limitbuf limit];
    speedbuf = [speedbuf speed(1)];
    pedalbuf = [pedalbuf pedal(1)];

    % plot
    cla;
    hold on;
    plot(limitbuf, 'ob');
    plot(speedbuf, '-b');
    plot(pedalbuf, '-r');
    legend('Speed Limit','Current Speed','Pedal Percent');

    hold off;

    step = step + 1;

% end loop
end

end
```

A.4.2 Engine Code

```
function NormalEngine(driverAddress)

addpath('/home/ben/workspace/LAPIS_API_Matlab_working_copy/');

% read address of driver
driverName = 'NormalDriver';

% initalize variables and LAPIS
lapi = LapissApi('NormalEngine');
disp('connecting');
lapi.Connect(driverName, driverAddress);
disp('connected');
start = LapissData('Start', [0 0]);
pedal = LapissData('Pedal', [.1 0]);
speed = LapissData('Speed', [1 0]);
lapi.Publish('Start', start, 0);
```

```
lapi.Publish('Pedal', pedal, 0);
lapi.Publish('Speed', speed, 0);

% wait for start signal
while (start.d == 0)
    pause(.25);
end

disp('started');

% be an engine (loop until driver goes away)
while (¬isempty(strfind(lapi.GetNodeList(),',')))
    % adjust speed based on accel. position
    if (speed.d > 100)
        speed.d = 100;
    elseif (speed.d > 75)
        speed.d = speed.d + 1.5 * pedal.d - 1.0;
    elseif (speed.d > 50)
        speed.d = speed.d + 2.0 * pedal.d - 1.0;
    elseif (speed.d > 25)
        speed.d = speed.d + 2.0 * pedal.d - .5;
    else
        speed.d = speed.d + 2.5 * pedal.d - 0.25;
    end

    %pause(0.05);
% (end loop)
end

end
```

Bibliography

[1] C.R. Johnson and S.G. Parker. A Computational Steering Model Applied to Problems in Medicine. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 540–549, 1994.

[2] J.S. Vetter. Computational Steering Annotated Bibliography. *ACM SIGPLAN Notices*, 32(6):40–44, 1997.

[3] S.M. Pickles, R. Haines, R.L. Pinning, and A.R. Porter. A Practical Toolkit for Computational Steering. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 363:1843–1853, 2005.

[4] H. Liu and M. Parashar. Rule-Based Monitoring and Steering of Distributed Scientific Applications. *International Journal of High Performance Computing and Networking*, 3(4):272–282, 2005.

[5] K. Zhang, K. Damevski, V. Venkatachalapathy, and SG Parker. SCIRun2: A CCA Framework For High Performance Computing. In *Proceedings of the International Workshop on High-Level Parallel Programming Models and Supportive Environments*, volume 9, pages 72–79, 2004.

[6] S.M. Pickles, R. Haines, R.L. Pinning, and A.R. Porter. Practical Tools for Computational Steering. In *Proceedings of the UK e-Science All Hands Meeting*, volume 31, 2004.

[7] S. Jha, S. Pickles, A. Porter, and M. Computing. A Computational Steering API for Scientific Grid Applications: Design, Implementation and Lessons. In *Proceedings of the Workshop on Grid Application Programming Interfaces*, 2004.

[8] C.M. Poteras, M. Mocanu, and C. Petrişor. A Distributed Design for Computational Steering with High Availability of Data. 2012.

[9] W. Gu, J. Vetter, and K. Schwan. An Annotated Bibliography of Interactive Program Steering. *ACM SIGPLAN Notices*, 29(9):140–148, 1994.

[10] S.G. Parker, C.R. Johnson, and D. Beazley. Computational Steering Software Systems and Strategies. *IEEE Computer Science and Engineering*, 4(4):50–59, 1997.

[11] J.D. Mulder, J.J. van Wijk, and R. van Liere. A Survey of Computational Steering Environments. *Future Generation Computer Systems*, 15(1):119–129, 1999.

[12] J.D. Mulder and J.J. van Wijk. Logging in a Computational Steering Environment. In *Proceedings of the Sixth Eurographics Workshop*, pages 118–125, 1995.

[13] C.T. Silva, A.E. Kaufman, and C. Pavlakos. PVR: High-Performance Volume Rendering. *IEEE Computational Science and Engineering*, 3(4):18–28, 1996.

[14] S. Bullock, J. Cartlidge, and M. Thompson. Prospects for Computational Steering of Evolutionary Computation. In *Proceedings of International Conference on Artificial Life*, pages 131–137, 2002.

[15] J.D. Mulder and J.J. van Wijk. Parametrizable Cameras for 3D Computational Steering. In *Proceedings of the Eurographics Workshop on Visualization in Scientific Computing*, pages 165–175, 1997.

[16] T. Kesavadas and A. Sudhir. Computational Steering in Simulation of Manufacturing Systems. In *Proceedings of the IEEE International Conference on Robotics and Automation*, volume 3, pages 2654–2658, 2000.

[17] C. Lejdfors, M.O. Khan, A. Ynnerman, and B. Jonsson. GISMOS: Graphics and Interactive Steering of MOlecular Simulations. In *Proceedings of the Conference on Simulation and Visualization on the Grid: Parallelldatorcentrum*, volume 13, pages 154–164, 2000.

[18] P. Wenisch, O. Wenisch, and E. Rank. Harnessing High-Performance Computers for Computational Steering. *Lecture Notes in Computer Science*, pages 536–543, 2005.

[19] D. Lorenz, P. Buchholz, C. Uebing, W. Walkowiak, and R. Wismuller. Online Steering of HEP Grid Applications. In *Proceedings of the Cracow Grid Workshop*, pages 191–198, 2006.

[20] J. Harting, S. Wan, and P.V. Coveney. RealityGrid: High Performance Computing, Visualisation, Computational Steering and Teragrids. *Capability Computing Newsletter, HPCx Community*, 2:4–7, 2003.

[21] J. Wood, K. Brodlie, and J. Walton. gViz: Visualization and Steering for the Grid. In *Proceedings e-Science All Hands Meeting*, 2003.

[22] S.M. Pickles, R.J. Blake, B.M. Boghosian, J.M. Brooke, J. Chin, P.E.L. Clarke, P.V. Coveney, N. Gonzalez-Segredo, R. Haines, J. Harting, et al. The TeraGyroid Experiment. In *Proceedings of the Workshop on Case Studies on Grid Applications at GGF*, volume 10, 2004.

[23] J.W. Atwood, M.M. Burnett, R.A. Walpole, E.M. Wilcox, and S. Yang. Steering Programs Via Time Travel. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 4–11, 1996.

[24] M. Parris, C. Mueller, J. Prins, A. Duggan, Q. Zhou, and E. Erikson. A Distributed Implementation of an N-body Virtual World Simulation. In *Proceedings of the Workshop on Parallel and Distributed Real-Time Systems*, pages 66–70, 1993.

[25] J.S. Vetter and K. Schwan. Techniques for High-Performance Computational Steering. *IEEE Concurrency*, 7(4):63–74, 1999.

[26] P.R. Woodward. Interactive Scientific Visualization of Fluid Flow. *Computer*, 26(10):13–25, 1993.

[27] J.F. Prins, J. Hermans, G. Mann, L.S. Nyland, and M. Simons. A Virtual Environment For Steered Molecular Dynamics. *Future Generation Computer Systems*, 15(4):485–495, 1999.

[28] R. Marshall, J. Kempf, S. Dyer, and C.C. Yen. Visualization Methods and Simulation Steering for a 3D Turbulence Model of Lake Erie. *ACM SIGGRAPH Computer Graphics*, 24(2):89–97, 1990.

[29] D.A. Reed, C.L. Elford, S.E. Lamm, T.M. Madhyastha, and E. Smirni. The Next Frontier: Interactive and Closed Loop Performance Steering. In *Proceedings of the ICPP Workshop on Challenges for Parallel Processing*, pages 20–31, 1996.

[30] L.J.K. Durbeck, N.J. Macias, D.M. Weinstein, C.R. Johnson, and J.M. Hollerbach. SCIRun Haptic Display for Scientific Visualization. In *Proceedings of the Phantom Users Group Workshop*, 1998.

[31] J.E. Swan II, M. Lanzagorta, D. Maxwell, E. Kuo, J. Uhlmann, W. Anderson, H.J. Shyu, and W. Smith. A Computational Steering System for Studying Microwave Interactions with Space-Borne Bodies. In *Proceedings of the IEEE Conference on Visualization*, pages 441–445, 2000.

[32] Y. Jean. An Integrated Approach for Steering, Visualization, and Analysis of Atmospheric Simulations. In *Proceedings of the IEEE Conference on Visualization*, pages 383–387, 1995.

[33] D.M. Beazley and P.S. Lomdahl. Lightweight Computational Steering of Very Large Scale Molecular Dynamics Simulations. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, page 50, Los Alamitos, CA, USA, 1996. IEEE Computer Society.

[34] H. Wright, K. Brodlie, and T. David. Navigating High-Dimensional Spaces to Support Design Steering. In *Proceedings of the IEEE Conference on Visualization*, pages 291–296, 2000.

[35] J. Leech, J.F. Prins, and J. Hermans. SMD: Visual Steering of Molecular Dynamics for Protein Design. *IEEE Computational Science and Engineering*, 3(4):38–45, 1996.

[36] D.J. Jablonowski, J.D. Bruner, B. Bliss, and R.B. Haber. VASE: The Visualization and Application Steering Environment. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 560–569, 1993.

[37] E. Rank, A. Borrmann, A. Düster, A. Niggl, V. Nübel, R. Romberg, and D. Scholz. From Adaptivity To Computational Steering: The Long Way Of Integrating Numerical Simulation Into Engineering Design Processes. In *Proceedings of International Conference on Adaptive Modeling and Simulation*, 2005.

[38] K. Brodlie, A. Poon, H. Wright, L. Brankin, G. Banecki, and A. Gay. GRASPARC: A Problem Solving Environment Integrating Computation and Visualization. In *Conference on Visualization*, volume 4, pages 25–29, 1993.

[39] S.G. Parker and C.R. Johnson. SCIRun: A Scientific Programming Environment for Computational Steering. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, 1995.

[40] C Johnson, S.G. Parker, C. Hansen, G.L. Kindlmann, and Y. Livnat. Interactive Simulation and Visualization. *Computer*, 32(12):59–65, 1999.

[41] E.H. Winer and C.L. Bloebaum. Visual Design Steering for Optimization Solution Improvement. *Structural and Multidisciplinary Optimization*, 22(3):219–229, 2001.

[42] J. Chin, P.V. Coveney, and J. Harting. The TeraGyroid Project - Collaborative Steering and Visualization in an HPC Grid for Mmodelling Complex Fluids. In *Proceedings of the UK e-Science All Hands Meeting*, 2004.

[43] J.X. Chen, D. Rine, and H.D. Simon. Advancing Interactive Visualization and Computational Steering. *IEEE Computational Science and Engineering*, 3(4):13–17, 1996.

[44] D.R. Mason and A.P. Sutton. Computational Steering in Monte Carlo Simulations of Thin Film Polystyrene. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 363:1961–1974, 2005.

[45] F.P. Brooks. Grasping Reality Through Illusion Interactive Graphics Serving Science. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1–11. ACM Press New York, NY, USA, 1988.

[46] H. Polachek. Before the ENIAC [Weapons Firing Table Calculations]. *IEEE Annals of the History of Computing*, 19(2):25–30, 1997.

[47] C.R. Johnson, M. Berzins, L. Zhukov, and R. Coffey. SCIRun: Application to Atmospheric Dispersion Problems Using Unstructured Meshes. *Numerical Methods for Fluid Dynamics VI*, pages 111–122, 1998.

[48] J. Vetter and K. Schwan. High Performance Computational Steering of Physical Simulations. In *Proceedings of the International Symposium on Parallel Processing*, pages 128–132, 1997.

[49] Y.C. Chou, D. Ko, and H.H. Cheng. Mobile agent-based computational steering for distributed applications. *Concurrency and Computation: Practice and Experience*, 21(18):2377–2399, 2009.

[50] J.E. Moreira, V.K. Naik, and D.W. Fan. Design and Implementation of Computational Steering for Parallel Scientific Applications. In *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*, pages 14–17, 1996.

[51] D.W. Miller, J. Guo, E. Kraemer, and Y. Xiong. On-the-Fly Calculation and Verification of Consistent Steering Transactions. In *Proceedings of the ACM/IEEE conference on Supercomputing*, pages 1–17, 2001.

[52] R.S. Kalawsky, S.P. Nee, I. Holmes, and P.V. Coveney. A Grid-Enabled Lightweight Computational Steering Client: A .NET PDA Implementation. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 363:1885–1894, 2005.

[53] R.S. Kalawsky and S.P. Nee. Important Issues Concerning Interactive User Interfaces in Grid Based Computational Steering Systems. In *Proceedings of the UK e-Science All Hands Meeting*, 2004.

[54] C.F. Sanz-Navarro, S.D. Kenny, A.R. Porter, and S.M. Pickles. Real-time Visualization and Computational Steering of Molecular Dynamics simulations of Materials Science. In *Proceedings of the UK e-Science All Hands Meeting*, volume 31, 2004.

[55] O.V. Aslanidi, K.W. Brodlie, R.H. Clayton, J.W. Handley, A.V. Holden, and J. Wood. Remote Visualization and Computational Steering of Cardiac Virtual Tissues Using gViz. In *Proceedings of the UK e-Science All Hands Meeting*, 2005.

[56] R. Muralidhar and M. Parashar. A Distributed Object Infrastructure for Interaction and Steering. *Concurrency and Computation: Practice and Experience*, 15(10):957–977, 2003.

[57] L. Jiang, H. Liu, M. Parashar, and D. Silver. Rule-Based Visualization in a Computational Steering Collaboratory. *Future Generation Computer Systems*, 21(1):53–59, 2005.

[58] T. Kesavadas and A. Sudhir. Control of Transient Phase for Discrete Event Simulation Using Computational Steering. In *Proceedings of the IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, volume 1, pages 111–116, 2001.

[59] W. Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. Stasko, J. Vetter, and N. Mallavarupu. Falcon: On-line Monitoring for Steering Parallel Programs. In *Proceedings of the Frontiers of Massively Parallel Computation*, volume 10, pages 422–429, 1995.

[60] K. Mayes, G.D. Riley, R.W. Ford, M. Luján, L. Freeman, and C. Addison. The Design of a Performance Steering System for Component-Based Grid Applications. *Performance Analysis and Grid Computing*, pages 111–127, 2003.

[61] G.S. Eisenhauer, W. Gu, K. Schwan, and N. Mallavarupu. Falcon-toward interactive parallel programs: The on-line steering of a molecular dynamics application. Georgia Institute of Technology, 1994.

[62] A.M. Bakic and M.W. Mutka. An Integrated Approach to Real-Time System Design and On-Line Performance Visualization With Steering. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 141–150, 2000.

[63] E. Kraemer and J. Wallis. Visualization and Interactive Steering of Simulated Annealing. In *Proceedings of SPDP Workshop Program Visualization and Instrumentation*, pages 12–17, 1996.

[64] J.J. van Wijk, R. van Liere, and J.D. Mulder. Bringing Computational Steering to the User. In *Proceedings of the Dagstuhl Seminar on Scientific Visualization*, pages 304–313, 1997.

[65] G.D. Kerlick and E. Kirby. Towards Interactive Steering, Visualization and Animation of Unsteady Finite Element Simulations. In *Proceedings of the IEEE Conference on Visualization*, pages 374–377, 1993.

[66] B. Parvin, J. Taylor, and G. Cong. Deepview: A Collaborative Framework for Distributed Microscopy. In *IEEE Confrence on High Performance Computing and Networking*, 1998.

[67] Kohl J. A. Semeraro B. D. Papadopoulos, P. M. CUMULVS: Extending a Generic Steering and Visualization Middleware for Application Fault-Tolerance. In *Proceedings of the Hawaii International Conference on System Sciences*, volume 7, pages 127–136, 1998.

[68] F. Tölke and K. Nachtwey. Computational Steering in Civil Engineering. In *IKM-Internationales Kolloquium über Anwendungen der Informatik und Mathematik in Architektur und Bauwesen Weimar*, 2003.

[69] J. Linxweiler, J. Tolke, and M. Krafczyk. Applying Modern Soft- and Hardware Technologies for Computational Steering Approaches in Computational Fluid Dynamics. In *Proceedings of the International Conference on Cyberworlds*, pages 41–45, 2007.

[70] A. Modi, N. Sezer-Uzol, L.N. Long, and P.E. Plassmann. Scalable Computational Steering System for Visualization of Large-Scale CFD Simulations. *American Institute of Aeronautics and Astronautics*, pages 24–27, 2002.

[71] JD Wood, M. Riding, and KW Brodlie. A user interface framework for grid-based computational steering and visualization. In *Proceedings of the UK e-Science All Hands Meeting*, 2007.

[72] J.D. Mulder and J.J. van Wijk. 3D Computational Steering with Parametrized Geometric Objects. In *Proceedings of the IEEE Conference on Visualization*, pages 304–311, 1995.

[73] J. Vetter and K. Schwan. Models for Computational Steering. In *Proceedings of the International Confrence on Configurable Distributed Systems*, pages 100–107, 1996.

[74] G. Eisenhauer and K. Schwan. An Object-Based Infrastructure for Program Monitoring and Steering. In *Proceedings of the SIGMETRICS symposium on Parallel and Distributed Tools*, pages 10–20, 1998.

[75] J.D. Mulder, R. van Liere, and J.J. van Wijk. Computational Steering in the CAVE. *Future Generation Computer Systems*, 14(3-4):199–207, 1998.

[76] R. Muralidhar, S. Kaur, and M. Parashar. An Architecture for Web-based Interaction and Steering of Adaptive Parallel/Distributed Applications. In *Proceedings of Euro-Par*, pages 1332–1339, 2000.

[77] M. Zhu, Q. Wu, and N.S.V. Rao. Computational monitoring and steering using network-optimized visualization and ajax web server. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12. IEEE, 2008.

[78] S.P. Nee and R.S. Kalawsky. Developing a Roaming PDA-Based Interface for a Steering Client for OGSI:: Lite using. Net: Practical Lessons Learned. In *Proceedings of the UK e-Science All Hands Meeting*, 2004.

[79] J Chin, J. Harting, S. Jha, P. V. Coveney, A. R. Porter, and S. M. Pickles. Steering in Computational Science: Mesoscale Modelling and Simulation. *Contemporary Physics*, 44(5):417–434, 2003.

[80] J.A. Kohl and P.M. Papadopoulas. Efficient and Flexible Fault Tolerance and Migration of Scientific Simulations Using CUMULVS. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 60–71, 1998.

[81] P.M. Papadopoulos and J.A. Kohl. Dynamic Visualization and Steering Using PVM and MPI. In *Proceedings of the European PVM/MPI Users' Group Meeting*, pages 297–303, 1998.

[82] H. Wright. Putting Visualization First in Computational Steering. In *Proceedings of UK e-Science All Hands Meeting*, pages 326–331, 2004.

[83] J. Brooke, T. Eickermann, and U. Woessner. Application Steering in a Collaborative Environment. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 61–70, 2003.

[84] A. Borrmann, P. Wenisch, M. Egger, C. van Treeck, and E. Rank. Collaborative computational steering: Interactive collaborative design of ventilation and illumination of operating theatres. In *Proc. of the Int. Conf. on Intelligent Computing in Engineering*, 2008.

[85] M. Riedel, W. Frings, T. Eickermann, S. Habbinga, P. Gibbon, A. Streit, F. Wolf, and T. Lippert. Collaborative interactivity in parallel hpc applications. *Remote Instrumentation and Virtual Laboratories*, pages 249–262, 2010.

[86] F. Niebling, A. Kopecki, and M. Becker. Collaborative steering and post-processing of simulations on hpc resources: Everyone, anytime, anywhere. In *Proceedings of the 15th International Conference on Web 3D Technology*, pages 101–108. ACM, 2010.

[87] O. Kreylos, A.M. Tesdall, B. Hamann, J.K. Hunter, and K.I. Joy. Interactive Visualization and Steering of CFD Simulations. In *Proceedings of the Symposium on Data Visualisation*, pages 25–34, 2002.

[88] M. Burnett, R. Hossli, T. Pulliam, B. VanVoorst, and X. Yang. Toward Visual Programming Languages For Steering Scientific Computations. *IEEE Computational Science and Engineering*, 1(4):44–62, 1994.

[89] C.R. Johnson and S.G. Parker. Applications in Computational Medicine Using SCIRun: A Computational Steering Programming Environment. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 2–19, 1995.

[90] J. Vetter and K. Schwan. Progress: A Toolkit for Interactive Program Steering. In *International Conference on Parallel Processing*, 1995.

[91] J. Linxweiler, M. Krafczyk, and J. Tölke. Highly interactive computational steering for coupled 3d flow problems utilizing multiple gpus. *Computing and visualization in science*, 13(7):299–314, 2010.

[92] C.M. Poteraş and M.L. Mocanu. A state machine-based parallel paradigm applied in the design of a visualization and steering framework. In *Proceedings of the 2nd international conference on Applied informatics and computing theory*, pages 232–236. World Scientific and Engineering Academy and Society (WSEAS), 2011.

[93] R. Walker, P. Kenny, and J. Miao. Balancing resolution and response in computational steering with simulation trails. In *Proceedings of the 2009 International Conference on Modeling, Simulation and Visualization Methods*, pages 167–172. CSREA Press, 2009.

[94] P.M. Papadopoulos and J.A. Kohl. A Library For Visualization and Steering of Distributed Simulations Using PVM and AVS. In *High Performance Computing Symposium*, pages 243–254, 1995.

[95] M. Miller, C.D. Hansen, and C.R. Johnson. Simulation Steering with SCIRun in a Distributed Environment. In *Proceedings of the International Workshop on Applied Parallel Computing*, pages 366–376, 1998.

[96] J.M. Brooke, P.V. Coveney, J. Harting, S. Jha, S.M. Pickles, R.L. Pinning, and A.R. Porter. Computational Steering in RealityGrid. In *Proceedings of the UK e-Science All Hands Meeting*, volume 2, pages 885–889, 2003.

[97] A. Costanzo, C. Jin, C.A. Varela, and R. Buyya. Enabling computational steering with an asynchronous-iterative computation framework. In *Proceedings of the 2009 Fifth IEEE International Conference on e-Science*, pages 255–262. IEEE Computer Society, 2009.

[98] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[99] R. Hempel. The mpi standard for message passing. In *High-Performance Computing and Networking*, pages 247–252. Springer, 1994.

[100] C.W. Harrop, S.T. Hackstadt, J.E. Cuny, A.D. Malony, and L.S. Magde. Supporting Runtime Tool Interaction for Parallel Simulations. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 1–15, 1998.

[101] R. van Liere, J.D. Mulder, and J.J. van Wijk. Computational Steering. *Future Generation Computer Systems*, 12(5):441–450, 1997.

[102] A. Modi, L.N. Long, and P.E. Plassmann. Real-Time Visualization of Wake-Vortex Simulations Using Computational Steering and Beowulf Clusters. In *Proceedings of the International Conference on Vector and Parallel Processing Systems and Applications*, pages 464–478, 2003.

[103] J. Knezevic, J. Frisch, RP Mundani, and E. Rank. Interactive computing framework for engineering applications. *Journal of Computer Science*, 7(5):591–599, 2011.

[104] J. Conway. The game of life. *Scientific American*, 223(4):4, 1970.

[105] M. Muratori, M. Roberts, R. Sioshansi, Marano V., and G. Rizzoni. A Highly Resolved Modeling Technique to Simulate Residential Power Demand. *Applied Energy (Submitted)*, 2012.