WEB APPLICATIONS FOR INTERACTIVE ENVIRONMENTAL MODELING

A dissertation submitted by

Jeffrey D. Walker

In partial fulfillment of the requirements for the degree of

Doctor of Philosophy

In

Civil and Environmental Engineering

TUFTS UNIVERSITY

May 2014

© 2014, Jeffrey D. Walker

Adviser: Steven C. Chapra
Abstract

Mathematical models are central to the study and management of environmental and water resources systems. However, there are major and persistent challenges associated with understanding, communicating, and using models. Over the past two decades, researchers have investigated various ways of using the World Wide Web (the ‘web’) to improve model accessibility and communication. However, existing research has primarily focused on server-side approaches where the model is executed on the server and the results displayed in the browser as static text and images.

This thesis demonstrates a novel approach to web-based modeling by using client-side web applications to perform model simulations, output visualizations, and user interface control all within the browser using only standard web languages. By performing these tasks within the browser, the interface supports highly interactive visualizations that allow the user to easily explore how the model behaves in response to changing parameters and input data.

A series of demonstrations are presented to illustrate various ways of using client-side web applications to provide interactive modeling interfaces. These applications allow the user to perform simulations, explore model theory and behavior, incorporate observation data, and share models with other users. Client-side web applications for interacting with these models are coupled to server-side data
storage and integration systems allowing the user to store model data on a centralized server, and to link models with their input datasets.

The culmination of this research is the concept of a ‘Living Model,’ which consists of an interactive web application for performing simulations and refining the model over time and a server-side data integration system for automatically updating input datasets as new data become available. The Living Model illustrates the potential for web applications to enhance the model life cycle by improving the accessibility, continuity, and persistence of model simulations.

The research presented in this thesis demonstrates a new approach for creating web-based user interfaces that will fundamentally change how we access, understand, and interact with environmental models. This improved accessibility and interactivity will ultimately lead to better understanding of environmental data, models, and systems and thus more informed research, management, and decision making.
Acknowledgements

I am grateful for the support of countless friends, family, peers, and colleagues.

In particular, this research would not have been possible with the support and mentorship of my advisor, Dr. Steve Chapra, who provided the guidance and flexibility allowing me to explore a challenging and somewhat unconventional research topic. I also thank the three additional members of my committee—Dr. Alva Couch, Dr. Pete Shanahan, and Dr. Rich Vogel—who provided valuable feedback that improved the quality of this research.

I also thank my friends and peers from Tufts, Boston, and beyond. In particular, my fellow students Rose Wang and Kyle Flynn, whose thoughtful conversations kept me motivated and enjoying life as a graduate student.

To my family—Bill, Martha, and Mark—whose support, interest, and enthusiasm in both my research and my life is a true gift.

And most importantly, to Elizabeth Halliday, without whose endless love and support this simply would not have been possible.

Funding for this research was provided by the Louis Berger Chair Endowment, the Office of the Provost, and the Department of Civil & Environmental Engineering at Tufts University.
# Table of Contents

Abstract .......................................................................................................................................................... ii
Acknowledgements ........................................................................................................................................ iv
Table of Contents ........................................................................................................................................ v
List of Tables ................................................................................................................................................ viii
List of Figures ............................................................................................................................................... ix

Chapter 1 Introduction .................................................................................................................................. 1
1.1 Improving Model Understanding ................................................................................................. 2
1.2 Improving Model Accessibility ..................................................................................................... 3
1.3 Thesis Overview ............................................................................................................................... 5

Chapter 2 Web Technologies ...................................................................................................................... 8
2.1 Overview of Web Technologies .................................................................................................... 8
   2.1.1 Foundations of the World Wide Web ................................................................................. 9
   2.1.2 Web Standards ................................................................................................................. 12
   2.1.3 Server-side Web Applications ......................................................................................... 17
   2.1.4 Client-side Web Applications ......................................................................................... 22
2.2 Web-based Simulation Architectures ............................................................................................ 29
   2.2.1 Client-side Simulation ....................................................................................................... 32
   2.2.2 Client-side Visualization .................................................................................................. 34
2.3 Related Research ............................................................................................................................. 37
   2.3.1 Web-based Infrastructure ................................................................................................. 37
   2.3.2 Web-based Applications ................................................................................................... 39
2.4 Methodology ........................................................................................................................................ 43
   2.4.1 Theory ............................................................................................................................... 44
   2.4.2 User Interface .................................................................................................................... 44
   2.4.3 HTML Document ............................................................................................................. 45
   2.4.4 CSS Stylesheet .................................................................................................................. 46
   2.4.5 JavaScript Program ........................................................................................................... 47
2.5 Summary and Conclusions ............................................................................................................... 52

Chapter 3 Web-based Interactive River Model ........................................................................................ 54
3.1 Introduction .......................................................................................................................................... 56
3.2 Overview of Web-based Simulation ............................................................................................... 60
3.3 Modern Web Application Development ....................................................................................... 63
   3.3.1 Advances in Web Technology Standards ....................................................................... 65
   3.3.2 Advances in JavaScript Development Practices and Performance. 66
3.4 Design and Implementation of a Client-based Web Simulation Model. 69
   3.4.1 Model Application .............................................................................................................. 70
   3.4.2 Application Architecture .................................................................................................... 71
   3.4.3 Application Testing ............................................................................................................. 80
   3.4.4 Application Workflow ......................................................................................................... 80
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5</td>
<td>Discussion</td>
<td>83</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Comparison of Web Applications and Desktop Software</td>
<td>87</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Potential Use Cases for Client-side Web Applications</td>
<td>92</td>
</tr>
<tr>
<td>3.6</td>
<td>Conclusions</td>
<td>94</td>
</tr>
<tr>
<td>3.7</td>
<td>Appendix A: Mathematical Theory</td>
<td>96</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>99</td>
</tr>
<tr>
<td>4.2</td>
<td>Understanding Model Theory</td>
<td>102</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Mental Models</td>
<td>105</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Cognitive Load Theory</td>
<td>109</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Interactive Visualizations</td>
<td>113</td>
</tr>
<tr>
<td>4.3</td>
<td>abcd Water Balance Model</td>
<td>116</td>
</tr>
<tr>
<td>4.4</td>
<td>Model Equations</td>
<td>118</td>
</tr>
<tr>
<td>4.5</td>
<td>Mass Balance Diagram</td>
<td>122</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Generalized Mass Balance Diagram</td>
<td>123</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Groundwater Storage Compartment</td>
<td>124</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Soil Moisture Compartment</td>
<td>128</td>
</tr>
<tr>
<td>4.6</td>
<td>Web-based Interactive Visualization</td>
<td>134</td>
</tr>
<tr>
<td>4.7</td>
<td>Discussion and Conclusions</td>
<td>137</td>
</tr>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>140</td>
</tr>
<tr>
<td>5.2</td>
<td>Model Description</td>
<td>143</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Potential Evapotranspiration</td>
<td>144</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Input Data Requirements</td>
<td>145</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Parameter Descriptions</td>
<td>146</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Example Dataset</td>
<td>147</td>
</tr>
<tr>
<td>5.3</td>
<td>Application Architecture</td>
<td>147</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Application Components</td>
<td>148</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Frameworks and Libraries</td>
<td>150</td>
</tr>
<tr>
<td>5.4</td>
<td>Application Interfaces</td>
<td>151</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Theory Page</td>
<td>151</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Setup Page</td>
<td>153</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Simulation Page</td>
<td>155</td>
</tr>
<tr>
<td>5.4.4</td>
<td>Calibration Page</td>
<td>159</td>
</tr>
<tr>
<td>5.4.5</td>
<td>Optimization Page</td>
<td>161</td>
</tr>
<tr>
<td>5.5</td>
<td>Discussion and Conclusions</td>
<td>164</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction</td>
<td>168</td>
</tr>
<tr>
<td>6.2</td>
<td>Living Model Architecture</td>
<td>169</td>
</tr>
<tr>
<td>6.3</td>
<td>Server-side Web Application</td>
<td>171</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Database Architecture</td>
<td>173</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Database Management Interface</td>
<td>174</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Dataset Interface</td>
<td>179</td>
</tr>
<tr>
<td>6.3.4</td>
<td>REST API</td>
<td>181</td>
</tr>
<tr>
<td>6.4</td>
<td>Client-side Web Application</td>
<td>186</td>
</tr>
<tr>
<td></td>
<td></td>
<td>188</td>
</tr>
</tbody>
</table>
6.5 Discussion and Conclusions ................................................................. 195
Chapter 7 Conclusions ........................................................................... 199
References .............................................................................................. 204
List of Tables

Table 1: List of Existing Web-based Simulation Applications ......................... 40
Table 2: FOSS JavaScript libraries used by the WIRM client-side web application................................................................. 76
List of Figures

Figure 2-1: Web Communications................................................................. 10
Figure 2-2: Static Web Pages................................................................. 18
Figure 2-3: Server-side Scripting......................................................... 20
Figure 2-4: AJAX Communications.................................................. 25
Figure 2-5: Client-side Web Application............................................. 27
Figure 2-6: WBS Architectures ......................................................... 31
Figure 2-7: Unit Hydrograph Interaction............................................. 45
Figure 2-8: Unit Hydrograph HTML Document................................. 46
Figure 2-9: Unit Hydrograph CSS File................................................ 47
Figure 3-1: Schematic diagram of the client and server architectures for WIRM. 72
Figure 3-2: Screenshot of the WIRM user interface.............................. 76
Figure 3-3: Application sequence for a typical use case of WIRM.......... 81
Figure 4-1: Solution to First-order Decay Equation............................ 104
Figure 4-2: abcd Model Diagram ..................................................... 119
Figure 4-3: Generalized Mass Balance Diagram................................... 124
Figure 4-4: Groundwater Storage and Discharge Components............. 125
Figure 4-5: Groundwater Mass Balance Diagram............................... 126
Figure 4-6: Effect of Changing Parameter d on Groundwater Mass Balance Diagram................................................................. 126
Figure 4-7: Example of Groundwater Algorithm................................. 127
Figure 4-8: Time series of Groundwater Storage................................. 128
Figure 4-9: Soil Moisture Mass Balance Diagram................................. 129
Figure 4-10: Example Calculation of Soil Moisture Components........... 131
Figure 4-11: Effect of Changing Parameter a on Soil Moisture Compartment.. 131
Figure 4-12: Effect of Changing Parameter b on Soil Moisture Mass Balance.. 132
Figure 4-13: Effect of Changing Parameter c on Soil Moisture Mass Balance.. 133
Figure 4-14: Effect of Changing Potential Evapotranspiration on Soil Moisture Mass Balance ................................................................. 133
Figure 6-14: Living Model Interface showing Period Selection by Brushing.... 190
Figure 6-15: Living Model Interface for the Aberjona River after Calibrating over Period 2010–2011 ........................................................................................................ 191
Figure 6-16: Living Model Interface for the Aberjona River after Saving the Model .................................................................................................................. 192
Figure 6-17: Living Model Interface for the Aberjona River comparing Validation Period .................................................................................................................. 193
Figure 6-18: Living Model Interface for the Aberjona River with the Updated Period Selected............................................................................................................. 194
Chapter 1

Introduction

Mathematical models are central to the study of environmental and water resources systems. Models are used to confirm theories, make predictions, and support planning, management, and policy decisions. But despite their widespread use, there are major and persistent challenges associated with understanding and using models (Friedman et al., 1984; NRC, 1999; NRC, 2007) Model software is often considered too complex, difficult to use, and inaccessible. The communication of model theory and results between experts and non-experts is encumbered by technical jargon and mathematical notation. Addressing these challenges requires not only innovative technical approaches to software development, but thinking beyond the science and engineering represented by models and considering the cognitive aspects of the modeling process.

In this thesis, I demonstrate the use of modern web technologies to create interactive visualizations that enhance understanding and communication of model theory and improve the usability and accessibility of model software. This research provides a novel methodology for creating web-based model software in which the application logic, user interface, numerical computations, and output visualizations are all managed and executed within a standard web browser. Input and observational data are incorporated either by loading simple text files or by coupling the application to a server-side data storage and integration system that
automatically retrieves input datasets from external data providers. Together, these systems illustrate the concept of a “living model” as an accessible, web-based modeling framework that enhances understanding and usability through interactive visualizations and improves model persistence by automatically updating input datasets as new data become available.

1.1 Improving Model Understanding

A common criticism of models is that they can be treated as black boxes. The user feeds input data into the model, the model performs some computations, and output data are returned to the user. But how does the user know how the computations are performed? How do they know the effect each parameter value and configuration option has on the model and resulting output? Unless the user personally wrote the software, understanding how the software works requires first understanding the underlying model theory and then making a mental connection between the software interface and that theory.

Traditionally, model theory is provided in the form of model documentation comprised of equations, diagrams, and text describing the relationships and processes represented by the model. By studying the documentation and learning the model theory, the user constructs a mental representation of the model. Cognitive scientists sometimes refer to this mental representation as a mental model (Johnson-Laird, 1983b; Norman, 2002; Jones et al., 2011). It is through this mental model that users make the connection between the software and the underlying theory. Improving the ability to understand how the software works and how to use
it requires focusing on ways of enhancing and supplementing this mental connection.

Visualizations are commonly used to explain model theory in a form that is more intuitive than equations and mathematical notation. Conceptual model diagrams provide an overview of the components and processes that are represented in a model. Static visualizations such as line charts and scatter plots depict the relationships between variables and parameters and effectively provide a visual representation of the underlying equations. However, by being limited to the medium of paper and ink (and its digital equivalent, the PDF document), static visualizations do not provide a means for exploring and understanding how changing parameters or inputs affect the underlying relationships and behavior of the model.

Interactive visualizations provide an alternative approach to better understand the model theory by allowing the user to interactively explore how changing the parameters affects the underlying equations. Interactive visualizations are believed to enhance both cognitive processing and human reasoning (Card et al., 1999; Pike et al., 2009). This thesis focuses on using interactive visualizations to improve understanding of model theory and behavior, as well as to provide direct and instantaneous feedback through an intuitive user interface.

1.2 Improving Model Accessibility

The inaccessibility of model software is another major challenge for the use of environmental models. For the modeling process to be open and transparent, the
model itself must be accessible. Although model accessibility has been greatly improved since the 1980s through the widespread adoption of personal computers (Chapra and Canale, 1987), there remain a number of practical issues related to the accessibility of desktop software for multiple users. Hardware incompatibilities, operating system requirements, software dependencies, and licensing restrictions are just some of the problems that prevent multiple users from accessing and sharing models. In addition, the diversity of computing platforms and operating systems is quickly expanding with the rise of mobile smartphones and tablets. The web has become a new software platform providing greater accessibility and compatibility by requiring only an Internet connection and a standard web browser.

Use of the web to improve model accessibility has been known and discussed for many years within the environmental modeling community (Voinov and Costanza, 1999). Although there has been a considerable amount of research on using the web to improve model accessibility, much of this previous research has focused on server-side approaches to web development where the model is executed remotely, either on the server or in the cloud. However, current web development practices are shifting towards client-side approaches, where more of the application—in particular the user interface—is managed and executed within the web browser and independent of the server. By executing more of the application within the browser, this client-side approach provides a richer and more interactive user experience. And yet, there has been relatively little research in environmental modeling focused on the design and implementation of client-side user interfaces for accessing web-based models and data.
In addition to improving software accessibility, the web also provides the technologies and infrastructure for linking models to their input data sources (Buytaert et al., 2012). The ability to automatically fetch input data improves the efficiency of the modeling process by eliminating the costly and time-consuming tasks of manually gathering, processing, and formatting input data. This automated data integration allows for persistent and continuous simulations allowing users to monitor and better understand the long-term behaviors of both the model and the physical system.

1.3 Thesis Overview

The goal of this thesis is to demonstrate the use of modern web technologies to improve model understanding and accessibility. A series of demonstration projects are presented to illustrate the various ways web technologies, and in particular client-side technologies and development approaches, can enhance how we use and understand models through interactive visualizations. These projects also demonstrate various ways of linking client-side web applications to server-side data storage and integration systems for storing model configurations and data, and for linking models with their input datasets. The organization of the thesis is as follows.

In Chapter 2, I provide an overview of modern web technologies and discuss the trade-offs between client-side and server-side development approaches. I discuss how client-side web technologies facilitate improved user interfaces and allow for highly interactive visualizations, which cannot be achieved using traditional server-side approaches. I then review recent advances in web-based approaches to environmental modeling, and discuss how the majority of existing research has
focused primarily on server-side approaches. I then describe the general methodology for creating interactive visualizations and user interfaces based on client-side technologies, which are then applied throughout the remainder of the thesis.

In Chapter 3, I present the Web-based Interactive River Model (WIRM), which provides an interactive and visual interface for the classic Streeter-Phelps water quality model of dissolved oxygen and biochemical oxygen demand in rivers and streams. WIRM is comprised of a client-side web application used to set the parameter values and perform the simulation and a server-side web application that stores the user-defined parameter values and facilitates the sharing of model results and collaboration between multiple users.

In Chapter 4, I discuss theories and research from cognitive science, human-computer interaction, information visualization, and other related fields that explain why interactive visualizations are effective for improving model understanding. Using these theories as a foundation, I demonstrate how interactive visualizations facilitate a better understanding of model theory compared to the traditional approaches of static model documentation. A novel interactive visualization called the mass balance diagram is described and used to explain the theory and underlying equations of a simple hydrologic model—the $abcd$ water balance model.

In Chapter 5, I present a client-side web application called the Web-based Interactive Watershed Model (WIWM). WIWM is an implementation of the $abcd$
water balance model and provides a series of interactive visual interfaces for understanding the behavior of dynamic simulations. The application allows the user to load and save input data using simple text files, evaluate the goodness of fit of the model using standard graphical and statistical methods, and search the parameter space to minimize model error and perform model calibration. This application also illustrates the potential for client-side web applications to maintain application state across multiple pages and between browser sessions without requiring any server-side data processing or storage.

In Chapter 6, I demonstrate the concept of a ‘living model’ by coupling an interactive client-side web application of the abcd water balance model to a server-side data integration system that automatically processes and stores input data obtained from external data providers. I discuss how this client-side modeling application coupled with server-side data management allows for the realization of the living model concept, and discuss the potential implications of this approach for fundamentally changing how we use models to study and manage environmental systems.

Finally, in Chapter 7, I provide some conclusions on the results and broader implications of this work. I also discuss the limitations of this work and provide some suggestions for further research to explore additional ways of using client-side web applications and interactive visualizations for environmental modeling and data analysis.
Chapter 2

Web Technologies

This chapter describes the technical background, related research, and methodology used in this thesis. The chapter begins with an overview of the core concepts and technologies used on the web. This overview focuses on the evolution from static to dynamic web pages, and from documents to interactive applications. The next section describes alternative architectures for web applications of models and other analytical tools involving numerical computations and data visualizations. With this background information, a review of related research in the field of environmental and water resources engineering is presented. This review focuses on which technologies and architectures are commonly used and how existing approaches can benefit from modern development practices. Finally, the general methodology used in this thesis is illustrated with a simple example: an interactive visualization of a synthetic unit hydrograph. This example serves to explain the technical details of performing numerical computations and visualizations using only standard web languages within the web browser.

2.1 Overview of Web Technologies

This section provides a high-level overview of web technologies and development practices. This information will provide a basis for discussing alternative approaches to web-based simulation modeling. It will also be used to describe
related research in the field of environmental and water resources engineering, and how the methodology used in this thesis provides a unique contribution to the field.

### 2.1.1 Foundations of the World Wide Web

The World Wide Web (the “web”) was created in the early 1990s by Tim Berners-Lee at the European Organization for Nuclear Research (CERN) (Berners-Lee, 1998). The web was designed as a hypertext-based information system to be accessed over the Internet (Berners-Lee and Cailliau, 1990). Hypertext is described as a “way to link and access information of various kinds as a web of nodes in which the user can browse at will” (Berners-Lee and Cailliau, 1990). The web was created to provide a common interface for accessing and retrieving information on separate computer systems as well as a means of linking information across these systems.

The creation of the web involved three core technologies (Shklar and Rosen, 2003):

- **Hypertext Transfer Protocol (HTTP):** a communications protocol
- **Universal Resource Locator (URL):** an addressing scheme
- **Hypertext Markup Language (HTML):** a document format

A web document is written in HTML, identified using a URL, and transferred using HTTP. Users access the web through a special kind of software called a web browser. The browser renders HTML documents in a visual and human-readable form. The browser also performs communications with a web server, retrieving HTML documents and allowing users to navigate from one document to another through hyperlinks.
2.1.1.1 Web Communication

The web involves an exchange of information between two systems: the client and the server (Casteleyn et al., 2009; Deitel et al., 2011). The client is the user’s local computer, which runs the web browser. These two systems communicate over the Internet using HTTP. This communication involves exchanging messages called requests and responses. A request message is sent from the client to the server, and a response message is returned from the server to the client (Figure 2-1).

![Figure 2-1: Web Communications](image)

By and large, this simple request-response protocol and client-server architecture has not changed since the web was created. What has changed is the kind of information being exchanged and how that information is generated and processed on the client or the server.

2.1.1.2 Web Addresses

The Universal Resource Locator (URL) is an addressing scheme used to locate web resources (W3C, 2004a). Resources represent a piece of information. Originally,
resources primarily referred to HTML documents stored on the server. For example, the URL:

\[ http://www.example.com/foo/bar.html \]

specifies a resource accessible using the HTTP protocol, located on the server with domain name \textit{www.example.com}, and corresponding to an HTML file named \textit{bar.html} located within a directory called \textit{foo/}. Today, web resources are no longer limited to HTML files stored on the server. Resources can be other types of files stored on the server (e.g. videos or images), or forms of information that are not files such as records in a database.

\textbf{2.1.1.3 Web Content}

In order for web content to be accessible from different browsers, a standard document format was needed to define the syntax and rules used to render that content. The Hypertext Markup Language (HTML) is the core language for defining web content and was created specifically for the web (W3C, 2014a). Hypertext is a way of embedding within a document links that refer to other documents. Although the concept of hypertext was invented long before the web was created, it proved to be very effective for linking information over a global computer network (Nielsen, 1995). HTML itself is a specific document format for specifying hypertext links (hyperlinks) using URLs. Although HTML was originally intended for text-based documents, it has evolved into a more generalized format for creating not only documents, but also application interfaces. The details of HTML are described in the Section 2.1.2.1 below.
2.1.2 Web Standards
As the web grew in the early 1990s, new web browsers were developed to access the web from different platforms and operating systems. Although HTML was generally accepted as the core document format, the different browser vendors began adding additional languages and features to supplement HTML. These features provided animation, interaction, improved aesthetic design, and other enhancements designed to attract new users. However, the different languages and features available across the various browsers led to incompatibilities of web content. These browser incompatibilities posed a major challenge for web developers. Developers were forced to either create content compatible with only a single browser, or create multiple versions of the same content that would appear and behave similarly across different browsers. In response to these incompatibilities, the web community adopted standardized languages and interfaces known as the web standards.

The web standards are a core set of language specifications that serve as guidelines for developers and browser vendors to ensure the interoperability and long-term stability of web content (WaSP, 2002). Most web standards are governed by the World Wide Web Consortium (W3C), an international organization comprised of interested stakeholders including web developers, browser vendors, non-profit organizations, researchers, and government agencies, among others. The W3C was formed in 1994 by the founder of the web, Tim Berners-Lee, with the mission to “lead the World Wide Web to its full potential by developing protocols and guidelines that ensure the long-term growth of the Web” (W3C, 2012).
The W3C uses a consensus-driven decision process to define the standard languages of the web, receiving input from its member organizations and reflecting the views of the web community as a whole. The web standards are open source and free so they can be used and implemented in software without imposing licensing fees or other restrictions. Although the use of web standards is not required, it is generally considered best practice for browsers and developers to use these common languages so that the content is accessible and interoperable across different browsers and platforms.

The web standards have been critical to maintain the openness of the web, ensuring that web content is accessible regardless of the device, platform, or browser being used. Without these standards, the web would have become increasingly fragmented with certain websites only accessible by certain browsers or devices. This would have hindered the tremendous growth and success of the web as we know it today.

There are currently four core standards used for web application development (Hales, 2013):

- **Hypertext markup Language (HTML):** a markup language for specifying the content
- **Cascading Style Sheets (CSS):** a styling language for specifying the presentation of the content
- **Document Object Model (DOM):** an interface for dynamically manipulating the content within the browser
• **JavaScript:** a scripting language for programming web content and accessing the DOM, officially called ECMAScript

### 2.1.2.1 Hypertext Markup Language

HTML is a mark-up language that defines the content and semantic structure of a web page (W3C, 2014a). Mark-up refers to the use of tags to wrap text or other content. For example, a paragraph of text is wrapped in a paragraph tag (e.g. `<p>This is a paragraph.</p>`). These individual pieces of content are called elements. An HTML document is comprised of multiple elements, which can be nested in a hierarchical structure. Web browsers use these different tags to determine how each element should be displayed to the user. Some elements are not shown to the user at all, but rather provide links to other files such as images or scripts, or contain metadata about the document such as the author’s name. Although HTML tags can include attributes to define the aesthetic properties (e.g. font size and color), it is considered a best practice to separate HTML content from the presentation of that content using CSS (W3C, 2003).

### 2.1.2.2 Cascading Style Sheets

CSS is the standard language for defining the aesthetics properties (e.g. color, size, font type) and layout of HTML content (W3C, 2011a; W3C, 2013c). CSS defines style rules for specific types of HTML elements. For example, `{p color: red;}` is a CSS rule to set the font color to red for all paragraph elements on the page. The use of HTML to define the content and CSS to define the styles allows for a separation of content from presentation (W3C, 2003; W3C, 2013c). This separation lets developers more easily manage how a page, or collection of pages, is displayed.
CSS can be used to change the presentation of a page based on the kind of device being used. For example, a single HTML document can be displayed differently depending on whether the user is using a desktop computer or a mobile smartphone by defining separate CSS files for each platform.

2.1.2.3 Document Object Model

The DOM is a standard interface for accessing and manipulating the content on a web page (W3C, 2014b). When an HTML document is loaded by the browser, an instance of the DOM is created. The DOM stores the page as a tree-like data structure with each node on the tree corresponding to an HTML element. This data structure is used to both render the page as well as manipulate the individual elements through a client-side script. Although the DOM is often less well-known than the other web standards, it plays a central role in displaying and programming web content.

To understand what the DOM is, consider a spreadsheet application. When a spreadsheet loads an ASCII text file containing rows and columns of data, it creates a 2-dimensional array to hold the data. That array is used to both display the values of the array on the spreadsheet interface, and also to access and manipulate those values using formulas or programming scripts. Similarly, the DOM is used to both display the HTML elements in the browser and also access and manipulate those elements using a programming script. When an element of the DOM is changed, the visual appearance of that element is automatically updated by the browser. Whereas the values in a spreadsheet are identified using cell references (e.g. A1 or
B2), the elements in the DOM can be selected using the tag name or other attributes such as an ID or class.

The DOM also provides an event management system that listens for and responds to user actions or other processes than occur on the page (W3C, 2013a). When an event occurs, the DOM automatically triggers a response. For example, when the user clicks on a hyperlink, a “click” event is triggered, which causes the browser to navigate to a new page. Developers can use the DOM event manager to create interactive features by defining custom events or overriding default events, as will be demonstrated in Section 2.4 below.

2.1.2.4 JavaScript

JavaScript is a programming language designed to control and manipulate web content. It was created in 1995 for the release of version 2.0 of the Netscape Navigator web browser by Brendan Eich, who designed and implemented the language in only 10 days (Severance, 2012). JavaScript was originally intended to be a lightweight alternative to a different programming language called Java. Although the two languages are often assumed to be related, they in fact have few similarities besides their name and syntax. JavaScript is a dynamic, loosely-typed language, and was designed for web developers with little programming experience to add small interactive elements such as alert pop-ups, animation or form validation (Severance, 2012). Java, on the other hand, is a static-typed language that requires compilation of the code before it can be executed. The name JavaScript was chosen primarily for marketing purposes and to attract developers who were already familiar with Java (Severance, 2012).
For many years, JavaScript suffered from inconsistencies in its implementation between the different browsers (Flanagan, 2011). In an effort to compete with Netscape, Microsoft created its own version of the same language for its browser, Internet Explorer, called JScript (Casteleyn et al., 2009). Although these languages were similar, they did not provide identical features or syntax. This led to incompatibilities between browsers, forcing developers to either target a single browser or write multiple versions of their scripts to run on different browsers. To address this problem, the language was standardized in 1998 by Ecma International (formerly called the European Computer Manufacturers Association, ECMA) (Ecma International, 2011). Because the name “JavaScript” was a registered trademark by Sun Microsystems, the official name of the language was changed to ECMAScript. However, the name JavaScript is still commonly used.

2.1.3 Server-side Web Applications
Originally, the web was comprised of static HTML documents stored as files on the web server (Casteleyn et al., 2009). The term “static” refers to the fact that these documents could only be changed by directly editing the files on the server. Figure 2-2 illustrates the process of retrieving static web pages.
The process begins when a user enters a URL in the browser address bar or clicks on a hyperlink and an HTTP request is sent to the server (labeled (1) in Figure 2-2). The server receives the request and locates the corresponding HTML file on the file system (2, 3). The server then constructs an HTTP response message containing the HTML file and sends this response back to the client (4). Once the client receives the response, it reloads the page and shows the new HTML document (5).

When the web was first created, it was primarily an information retrieval and navigation system comprised of static web pages. Users could enter a URL, retrieve the HTML page, read the contents, and click a link to navigate to another page. The only way to add or change a web page was to add a new HTML file or edit an
existing file directly on the server. However, new technologies were soon created to dynamically generate HTML documents in response to each request.

The ability to dynamically generate HTML documents was one of the first major advances in web technologies (Shklar and Rosen, 2003). Using a process known as server-side scripting, developers could configure web sites to dynamically generate each page using data stored in a database. This process allowed developers to customize the HTML content for each user. For example, a banking website can create an HTML document containing information specific to the current user, such as account balances and transactions. Another example is a search engine that generates an HTML page containing the results of a specific search query. The process of dynamically generating HTML documents is shown in Figure 2-3.
As with static web pages, this process begins when a request is sent from the client to the server (labeled (1) in Figure 2-3). When the server receives the request, it determines what information is being requested based on the URL or information about the user (e.g. log-in credentials). The server then fetches the relevant data from a database (2, 3), which is returned in some raw format (e.g. an array). The server processes the data (4), and uses a template engine to inject the results into an HTML template (5). The server then returns a response to the client containing the
final HTML document (6), and the browser reloads the interface with this new content (7).

A key component of this architecture is the template engine. HTML templates define the common components and layout of a website (e.g. navigation bar, logo, etc.). Templates also include placeholders where the application injects the specific data associated with each request. For example, a shopping website could define a template for showing individual products. When the user requests the page for a specific product, the server fills the template placeholders with the information for that product (e.g. name, description, price, and image). By using templates, developers can more easily make changes to a website by simply editing the templates. This is far more efficient than if the site were comprised of static HTML files, with one file corresponding to each product. In this case, the developer would need to change the HTML file for each product individually, as well as ensure consistency across all of the product pages.

Although server-side scripting provided a way of dynamically generating web content, the dependence on the server to generate this content limited the user experience (Farrell and Nezlek, 2007). For every update to the interface, the user had to wait for a request to be sent to the server, a response to be generated, and the response to be sent back to the client. Even with modern high-speed Internet connections, this start-stop-start-stop pattern imposes a significant delay in the responsiveness of the application. Alternative approaches were needed to create a richer, more interactive user experience that could match that of traditional desktop applications.
2.1.4 Client-side Web Applications
As the web evolved, it transitioned from a system for accessing and retrieving information to a new kind of software platform. The interoperability and compatibility of the web across different operating systems and computing platforms made it an attractive environment for hosting applications. New technologies and development approaches provide a variety of ways to run applications within the web browser, and thus eliminate the delay associated with requesting interface updates from the server. As a result, client-side web applications provide a user experience similar to that of desktop applications. These types of web applications are sometimes called Rich Internet Applications (RIAs) (Farrell and Nezlek, 2007; Lawton, 2008).

2.1.4.1 Plug-in Based Applications
The first web applications were primarily developed using browser plug-ins such as Java applets and Adobe Flash (Farrell and Nezlek, 2007; Deitel and Deitel, 2008; Lawton, 2008). Browser plug-ins are self-contained runtime environments executed within the browser. Unlike JavaScript, plug-ins generally require compilation before being sent to the client. They often include features, libraries, and frameworks for constructing user interfaces that look and behave similarly to desktop applications. Plug-ins also addressed the problem of browser incompatibility by providing a consistent runtime environment across different browsers (although plug-ins themselves are also subject to incompatibilities associated with different plug-in versions).
Although plug-ins proved to be useful platforms for delivering rich user experiences, they have some limitations and disadvantages. The primary limitation is that the user must install the plug-in before it can be used. Some users may not be able to install a plug-in due to administrative restrictions on their computer. Mobile devices such as smartphones and tablets often do not support plug-ins. Plug-ins also pose security risks as they provide greater access to the local computer’s file system and hardware than standard web languages.

One of the greatest criticisms of plug-ins is that they are based on proprietary languages and technologies. Unlike the standard web languages that are open, free, and developed by the web community, plug-ins depend on the long-term support of a single company. If a plug-in is no longer supported, content using that plug-in will no longer be accessible. Web standards, on the other hand, ensure the long-term persistence of web content by being open and maintained by the greater web community (WaSP, 2002).

2.1.4.2 Web 2.0 and AJAX Communications

In the mid-2000s, the web underwent a second major transition commonly referred to as Web 2.0. Although there is no clear definition, Web 2.0 generally refers to interactive web applications that allow users to easily retrieve content as well as generate their own content (Murugesan, 2007; O'Reilly, 2007). Blogs, wikis, office suites, and social networks are some of the iconic examples of Web 2.0 applications. Web 2.0 is also characterized by the term “mash-up” referring to the integration of data and web services originating from separate sources (Murugesan,
Web 2.0 applications make it easier for users to create and share, not just find and read, information.

One of the major contributing factors to Web 2.0 was a new approach to client-server communications called Asynchronous JavaScript and XML (AJAX) (Garrett, 2005). AJAX was not a new language, but a new way of using the existing standard languages to exchange information between the client and the server. AJAX improves the interactivity of web content by requesting data from the server in a raw format (instead of a complete HTML document), and updating a portion of the current page without reloading the entire page by manipulating the DOM. It thus provides a solution to the start-stop-start-stop process of requesting every interface update from the server. Figure 2-4 shows the process of AJAX-style communications.
The process begins when the user performs an action (e.g. clicks a button) that requires an update to the page (labeled (1) in Figure 2-4). A JavaScript program running in the browser responds to the event by sending a request for new data to the server (2). The server receives the request, fetches the relevant data from the database (3, 4), and processes the data (5). The server then sends back a response message containing the data in a raw format such as Extensible Markup Language (XML) or JavaScript Object Notation (JSON) (6). The JavaScript program receives the response, formats the data as HTML elements (7), and injects those elements
into the DOM (8). When the DOM is changed, the visual display of the page is automatically updated.

AJAX had a major impact on web development practices. It was a new approach for creating interactive web applications that used only the standard web languages. Because it is asynchronous, AJAX allows the user to continue interacting with the interface as new data are being requested. Google Maps was one of the first canonical examples of AJAX-based applications (Garrett, 2005). Google Maps allows users to interactively pan and zoom by loading new map tiles as needed using AJAX. In contrast, previous mapping websites provided navigational buttons that triggered an entire refresh of the page each time the user wanted to pan or zoom. Overall, AJAX marked the start of a transition in application architectures from the server to the client. As part of this transition, new JavaScript frameworks and libraries emerged that provide a consistent, high-level interface to AJAX functions and features.

2.1.4.3 Client-side Web Applications

Client-side web applications are a relatively new architecture where the entire application is written in JavaScript and executed within the web browser (Hales, 2013). The role of the server is to provide the application code (HTML, CSS, and JavaScript files) upon an initial request, and to provide access to a remote database through an Application Programming Interface (API). Instead of requesting fully-formed HTML documents from the server, a client-side application simply requests the raw data, which are then processed and injected into HTML templates to construct or update the user interface. Client-side web applications commonly use
AJAX communications to create, retrieve, update and delete data through the server’s API. This process is illustrated in Figure 2-5.

Figure 2-5: Client-side Web Application

Development of client-side web applications is facilitated by open-source JavaScript libraries and frameworks (Hales, 2013). These frameworks provide a structured approach to application development using variations of the model-view-controller (MVC) pattern (MacCaw, 2011). MVC is a common software
pattern for separating data (model), presentation (view), and application logic (controller) into distinct, reusable components. Using this approach, developers can create better organized and maintainable application codebases than by using plain JavaScript.

2.1.4.4 HTML5

In 2012, the W3C released version 5 of the HTML standard, commonly called HTML5 (W3C, 2014a). HTML5 introduced many new features and capabilities for client-side web applications. It included new element tags such as `<video>` and `<audio>` for adding multimedia content, and `<header>`, `<section>`, and `<footer>` for defining new semantic structures to a web page. HTML5 also included a number of new client-side APIs (W3C, 2013b). These APIs are accessed using JavaScript and provide new functionalities for client-side web applications. Some examples of these new APIs include:

- **Web Storage**: a mechanism for storing persistent data within the browser using key-value pairs
- **Indexed Database**: a database for storing larger quantities of structured data within the browser
- **File API**: a mechanism for reading and writing files within the browser
- **Web Workers**: support for multi-threaded applications
- **Offline Web Applications**: a mechanism for instructing web browsers to cache application files
These new APIs provide many of the features needed to develop complete client-side web applications without the need for browser plug-ins. Using the Web Storage and Indexed Database APIs, client-side web applications can store data and manage the application state (e.g. user preferences) with the browser. The File API provides the ability to read and write files from the local computer’s file system, which previously could only be performed by uploading the files to the server or by using browser plug-ins. The Web Workers API enables multi-threaded applications where the user interface is managed by one primary script, which connects to other scripts that are executed in separate threads. Finally, the Offline Web Application API allows developers to instruct the browser to cache all files necessary to run a client-side application, allowing the user to access the application through the browser without an Internet connection. These APIs thus include many of the features commonly found in traditional desktop software and thus provide a more complete software environment for client-side web applications. Use of the Web Storage and File APIs for maintaining application state and reading/writing text files within the browser are demonstrated in Chapter 5 of this thesis.

2.2 Web-based Simulation Architectures

The integration of web technologies and simulation models is sometimes called web-based simulation (WBS) (Byrne et al., 2010). Fishwick (1996) is credited with being the first to discuss the advantages and limitations of using the web as a platform for simulation modeling. WBS commonly refers to the use of simulation models where the primary user interface is accessed through a web browser (Byrne et al., 2010). However, there are other ways of using web technologies for
simulation modeling such as through web services that can be accessed using desktop software (see Section 2.3.1). For the purpose of this discussion, the term WBS will specifically refer to models accessed and controlled through a web browser.

WBS applications are a type of web application where the purpose of the application is to perform simulation modeling. The design considerations for WBS applications are similar to those of regular web applications. However, WBS applications often include two unique components: the simulation engine, and the visualization engine. The simulation engine takes input data, performs numerical computations, and generates output data. The visualization engine converts the output data into a data visualization (i.e. a chart or graph). Byrne et al. (2010) define three WBS architectures—local, remote and hybrid—based on which system (the client or the server) the simulation engine and visualization engine are each located (Figure 2-6).
With the remote WBS architecture, the simulations and visualizations are both performed on the server; the client simply displays the HTML documents containing the model results as tables of data or static images, which are generated by the server. With the local WBS architecture, the simulation and visualization are both performed within the browser; the server simply provides the code and, in some cases, access to input datasets or model configurations. In between these two extremes is the hybrid WBS architecture, in which the simulation is performed on the server and visualizations are generated on the client.

The three WBS architectures shown in Figure 2-6 are similar to the web applications architectures described in Section 2.1. The remote WBS architecture uses a server-side web application to perform the simulation and convert the results to a static HTML document. The local WBS architecture uses client-side approaches to execute the model and generate the visualization within the browser.
The hybrid architecture is similar to the use of AJAX communications where the client requests the model output data from the server and then generates a visualization of the output within the browser.

The choice of architecture depends in part on the size and complexity of the simulation model (Byrne et al., 2010). The remote WBS architecture can run models of any size or complexity and written in any programming language. The local WBS architecture is generally limited to simpler models that can be executed within the browser using JavaScript or plug-ins. However, as with client-side web applications in general, the local WBS architecture can provide a more interactive user experience by executing the model and generating the visualization within the client.

Traditionally, the local WBS architecture required the use of plug-ins (Byrne et al., 2010). Java applets are the most common environment for performing client-side computations and visualizations. However, client-side web applications built using web standards (HTML, CSS, and JavaScript) are now a viable alternative to browser plug-ins, as will be demonstrated in this thesis. The following two sections describe how the current web standards have overcome many of the previous limitations for performing client-side simulations and visualizations.

### 2.2.1 Client-side Simulation

Although JavaScript had been incorporated in all major browsers by the late 1990s, it was not considered a suitable language for performing numerical computations or any other computationally-intensive tasks. One of the primary limitations was the performance of the JavaScript engines in the earlier web browsers. As a
scripting language, JavaScript code is transferred from the server to the client as plain text (i.e. the source code). The browser runs this code through a JavaScript interpreter that would execute the script line-by-line. However, as JavaScript became a more integral part of web development through the use of AJAX and client-side web applications, browser vendors recognized the need to improve the performance of JavaScript programs.

In 2008, Google released a new browser called Chrome. Chrome included a new kind of JavaScript engine called V8, which uses Just-In-Time (JIT) compilation to translate JavaScript code into machine code (Google, 2012). Traditionally, interpreted languages like JavaScript are compiled to bytecode, which is then executed by an interpreter that translates the bytecode to machine code at run time. But by compiling directly to machine code and thus bypassing bytecode, the program can be executed at speeds similar to compiled languages such as C, C++, and Java (Clifford, 2012). V8 also includes the ability to dynamically optimize sections of code that are used repeatedly. This optimization can boost the performance of numerical algorithms that use iteration. Following Google’s lead, other browser vendors introduced their own highly optimized JavaScript engines (e.g. Mozilla’s SpiderMonkey, Microsoft’s Chakra, and Apple’s SquirrelFish). Today, modern browsers provide far more powerful JavaScript engines making it a more practical language for client-side simulation.

Another limitation of JavaScript is the lack of libraries for scientific computing. Although JavaScript includes a standard math library containing basic arithmetic and trigonometric functions, there is no built-in support for vectorized operations,
linear algebra, optimization, or other common numerical algorithms. However, a
number of FOSS JavaScript libraries are now emerging that provide scientific
computing toolkits that can be used in the browser. For example, Science.js
provides functions for linear algebra and statistics (Davies, 2014) and Numeric.js
includes ODE and PDE solvers and support for complex numbers (Loisel, 2014).
Although these packages are far less mature or comprehensive than scientific
computing libraries found in desktop software, they demonstrate the possibility of
implementing numerical algorithms in JavaScript. Over time, more libraries will
likely be developed and provide additional features and toolkits designed for
scientific computing in the browser.

2.2.2 Client-side Visualization
For many years, client-side visualization required plug-ins such as Java applets and
Adobe Flash that included libraries and features for generating data visualizations
in the browser (Johnson and Jankun-Kelly, 2008). Although HTML has long
provided the ability to include an image on a web page (e.g. the <img> tag), it did
not provide a means of creating an image from within the browser until recently.
HTML5 introduced two new graphics formats for generating graphics using
JavaScript: the Scalable Vector Graphics (SVG) format and the HTML5 canvas
element.

SVG is a vector-based graphics format in which the individual visual elements (e.g.
points and lines) are defined by markup tags similar to HTML (W3C, 2011b). For
example, <circle cx="50" cy="50" r="1"> defines a circle element with
a center position at x=50, y=50 and a radius of 1. These visual elements are
represented in the DOM as individual nodes, similar to the other HTML elements such as paragraphs or links. Because each circle, line, and other visual element is represented in the DOM, the styles of the elements (e.g. colors and sizes) can be defined using CSS. Furthermore, events can be bound to individual elements. For example, a click event can be bound to a circle element, which can trigger a detailed view of the data represented by that point. However, because each element is represented individually in the DOM, SVG is not well suited to large and complex visualizations involving many elements, as this can impede the performance of the overall interface (Cecco, 2011).

The HTML5 canvas element is a raster-based graphics format meaning that each pixel is assigned a set of values specifying the color and transparency (i.e. a bitmap) (W3C, 2014a). Unlike SVG, a canvas graphic is represented by a single HTML element in the DOM (i.e. the `<canvas>` element). Therefore, once a line or point has been added to the graphic, it can no longer be accessed through the DOM. Furthermore, to change these visual elements, the entire canvas must be redrawn because the individual elements cannot be manipulated. The canvas element provides higher performance for complex visualizations, but is generally harder to use and integrate with the rest of the interface.

Together, the SVG format and the HTML5 canvas element provide two complimentary graphics formats for creating visualizations within the browser. The choice of format depends on the specific requirements of the application. SVG is better suited to simpler graphics and when interaction is needed. The canvas
element provides better performance when the visualization contains a large number of elements.

To create visualizations using these standard graphics formats, there are a number of FOSS JavaScript visualization libraries such as Processing (Reas and Fry, 2007), Protovis (Bostock and Heer, 2009), and D3 (Bostock et al., 2011), among others. Many of these libraries take an approach to visualization that is similar to desktop-based visualization packages. Visualizations are constructed by choosing a chart type (e.g. bar chart, line chart, or scatterplot), providing the data, and defining the various attributes (e.g. colors, size, etc.). The visualization library then generates the chart and inserts it into the DOM. However, this traditional approach imposes a limitation on the kind of visualizations that can be created. Creating customized visualizations requires understanding the inner workings of the library and making modifications as needed. One of the newer visualization libraries called Data Driven Documents (or D3) takes a unique approach designed to improve the expressiveness and accessibility of web-based visualization (Bostock et al., 2011).

The central concept in D3 is to bind data directly to elements in the DOM. For example, an array of numbers can be bound to a set of SVG circle elements to create a scatter plot. When the underlying data change, the positions of the circles can be easily updated to reflect the new values in the data. Unlike other visualization libraries, D3 does not include any specific lexicon of graphic marks or chart types. Instead, it uses the element types provided by the HTML and SVG standards. D3 also allows visualizations to be closely integrated with the DOM, which allows developers to define aesthetic styles using CSS and to bind events using JavaScript.
The benefit of this approach is that the visualization can be better integrated with the other web standards (HTML, CSS, DOM, and JavaScript) so that the individual visual elements can be more efficiently added, removed, and manipulated. D3 is the primary visualization library used in this thesis and is demonstrated below in Section 2.4.

2.3 Related Research

The web has long been viewed as a promising platform for facilitating environmental management and decision making. Voinov and Costanza (1999) stated that the “interactivity of the Web offers great potential for linking science, planning, and public action.” Buytaert et al. (2012) describe how the web could reverse the traditional top-down flow of information from scientists to citizens by providing a platform for the public to easily access and interact with environmental data and models. Many researchers have explored a variety of ways to improve the accessibility and interoperability of environmental data and models on the web. The remainder of this section provides a review of existing research on web-based approaches to environmental modeling. For this review, the research projects are grouped in two broad categories: web-based infrastructure and end-user web applications.

2.3.1 Web-based Infrastructure

The development of web-based infrastructure, also known as cyber infrastructure, focuses on developing ways of discovering, accessing, and linking environmental models, databases, and other information through the web (Muste et al., 2013). This research generally involves developing server-side applications to access and link
models and data using web services. Web services facilitate machine-to-machine interactions and “provide a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks” (W3C, 2004b). Web services are implemented using a variety of protocols and architectures that expose models and datasets using standardized interfaces. A popular example is the U.S. Geological Survey (USGS) National Water Information System (NWIS), which provides web services for retrieving data in machine-readable form (Goodall et al., 2008). Two of the primary architectures include service-oriented architecture (SOA) and resource-oriented architecture (ROA) (Granell et al., 2013). These services can then be accessed using either web applications or desktop software.

Using web services, separate models can be linked together through standardized interfaces. Geller and Turner (2007) introduced the concept of a Model Web as a distributed network of interoperable model services. Nativi et al. (2013) describe various technologies and architectures for implementing the Model Web as part of an initiative by the Group on Earth Observation (GEO) and the Global Earth Observation System of Systems (GEOSS). Goodall et al. (2011) couple a locally-executed rainfall-runoff model with a web service that implements a routing algorithm. Peckham and Goodall (2013) show how web services can be used to link models with data sources. Other examples of the use of web services for environmental modeling include Castronova et al. (2013) and Goodall et al. (2013).

Another area of research is the creation of integrated database systems that catalogue data provided by multiple sources. One example is the Consortium of
Universities for the Advancement of Hydrologic Sciences Inc. (CUAHSI) Hydrologic Information System (HIS) (Horsburgh et al., 2009; Tarboton et al., 2009; Horsburgh et al., 2011). The HIS maintains a centralized catalogue of datasets available from a variety of sources including the U.S. Geological Survey (USGS), National Climatic Data Center (NCDC), and U.S. Environmental Protection Agency (USEPA), among others. With HIS, users can search across these separate databases with a single query, and retrieve data in a consistent and standardized format. The primary interface for HIS is a desktop application called HydroDesktop (Ames et al., 2012), although a web-based interface is currently under development.

In summary, web services are being used to create reusable and interoperable components that allow developers to couple environmental models and data. However, the majority of research to date has focused on the backend, or server-side, technologies and architectures. In order to use web services, there need to be client applications for discovering, linking, and accessing these services through a user interface.

2.3.2 Web-based Applications
The design and implementation of end-user web applications is another area of active research. These applications are sometimes characterized as decision support systems (DSS) and provide a user interface for discovering and retrieving data, and running simulations on the web. However, the web-based simulation (WBS) terminology will be used here instead to focus on the different WBS architectures described in Section 2.2.
Table 1 provides a list of research articles demonstrating WBS applications for environmental modeling and decision support. Table 1 compares the different technologies and application architectures that are commonly used to implement WBS applications in the field of environmental and water resources engineering. The table specifies on which system (the client or the server) the simulation engine and visualization engine are each executed, and whether the interface requires a browser plug-in. The citations are listed in chronological order beginning with the most recent articles. This list is limited to applications where the user interface is accessed through a web browser and allows the user to change the model configuration and generate new simulation results. It is also limited to papers that provide sufficient details about the underlying architecture and technologies to enable this comparison.

Table 1: List of Existing Web-based Simulation Applications

<table>
<thead>
<tr>
<th>Citation</th>
<th>Name</th>
<th>Browser Plug-in</th>
<th>Visualization Engine</th>
<th>Simulation Engine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delipetrev et al. (2014)</td>
<td>Reservoir Optimization Model</td>
<td>N</td>
<td>Client</td>
<td>Server</td>
</tr>
<tr>
<td>Sun (2013)</td>
<td>Nutrient Loading EDSS, PLOAD v1</td>
<td>N</td>
<td>Server</td>
<td>Server</td>
</tr>
<tr>
<td>Sun (2013)</td>
<td>Nutrient Loading EDSS, PLOAD v2</td>
<td>N</td>
<td>Client</td>
<td>Server</td>
</tr>
<tr>
<td>Muste et al. (2013)</td>
<td>Intelligent Digital Watershed</td>
<td>N</td>
<td>Server</td>
<td>Server</td>
</tr>
<tr>
<td>Mauget et al. (2013)</td>
<td>Cotton Irrigation Tool</td>
<td>N</td>
<td>Client</td>
<td>Server</td>
</tr>
<tr>
<td>Bürger et al. (2012)</td>
<td>ParFlow</td>
<td>N</td>
<td>Server</td>
<td>Server</td>
</tr>
<tr>
<td>Zhu et al. (2012)</td>
<td>REACHER</td>
<td>N</td>
<td>Client/Server</td>
<td>Server</td>
</tr>
<tr>
<td>Booth et al. (2011)</td>
<td>SPARROW DSS</td>
<td>N</td>
<td>Client/Server</td>
<td>Server</td>
</tr>
<tr>
<td>Granell et al. (2010)</td>
<td>AWARE Application - Geoportal</td>
<td>N</td>
<td>Client/Server</td>
<td>Server</td>
</tr>
<tr>
<td>Jia et al. (2009)</td>
<td>Rainfall-Runoff Model</td>
<td>N</td>
<td>Server</td>
<td>Server</td>
</tr>
<tr>
<td>Horak et al. (2008)</td>
<td>TANDEM-DSS</td>
<td>N</td>
<td>Client/Server</td>
<td>Server</td>
</tr>
<tr>
<td>Rao et al. (2007)</td>
<td>Conservation Reserve Program (CRP)-DSS</td>
<td>N</td>
<td>Server</td>
<td>Server</td>
</tr>
<tr>
<td>Choi et al. (2005)</td>
<td>L-THIA DSS</td>
<td>N</td>
<td>Server</td>
<td>Server</td>
</tr>
<tr>
<td>Citation</td>
<td>Name</td>
<td>Browser Plug-in</td>
<td>Visualization Engine</td>
<td>Simulation Engine</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------</td>
<td>-----------------</td>
<td>----------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>Castrogiovanni et al. (2005)</td>
<td>Rainfall-Runoff Model</td>
<td>N</td>
<td>Server</td>
<td>Server</td>
</tr>
<tr>
<td>Miller et al. (2004)</td>
<td>AGWA SDSS</td>
<td>Y</td>
<td>Client/Server</td>
<td>Server</td>
</tr>
<tr>
<td>Zeng et al. (2002)</td>
<td>Multi-reaction Transport Model (MRTM)</td>
<td>Y</td>
<td>Client</td>
<td>Client</td>
</tr>
</tbody>
</table>

The applications listed in Table 1 were created for a variety of different purposes. The most common type of web application is the geospatial decision support system. These applications use map-based interfaces sometimes called WebGIS (Jia et al., 2009). Many of these interfaces were implemented using server-side GIS software such as ArcIMS (Rao et al., 2007), Oracle MapViewer (Booth et al., 2011), and MapServer (Choi et al., 2005; Zhu et al., 2012). Recently, more researchers have begun using client-side libraries that run in the browser such as Google Maps API (Granell et al., 2010; Demir and Krajewski, 2013) and OpenLayers (Zhu et al., 2012; Sun, 2013; Delipetrev et al., 2014). Although these interfaces are interactive to some degree by allowing users to pan and zoom, they provide limited interactivity for running the simulation model. The user must often configure the model and set parameter values using forms comprised of text fields. A request is then sent to the server to execute the model using these values and,
once the simulation is complete, the results are sent back to the client as static summary tables and images. This process thus limits the user experience associated with configuring and running the model, much like server-side web applications limit the interactivity of dynamic web content (see Section 2.1.3). However, it is important to note that in many of these examples, the size and complexity of the model requires execution on the server.

Many of the applications listed in Table 1 use legacy models that were designed for desktop computers. Examples include SPARROW (Booth et al., 2011), PLOAD (Sun, 2013), SWAT (Miller et al., 2004; Rao et al., 2007; Muste et al., 2013), HSPF (Dymond et al., 2004), and HEC-RAS and MODFLOW (Horak et al., 2008). Some applications use customized rainfall-runoff models (Castrogiovanni et al., 2005; Jia et al., 2009) or reservoir optimization models (Delipetrev et al., 2014). In many of these cases, the local WBS architecture is not practical. To run these models in the client, the source code would need to be translated to JavaScript. Furthermore, many of these models are simply too large and complex to execute within the browser environment.

Although many of these models cannot be executed in the browser, client-side web technologies would improve how the user configures a model and interprets the model results. Instead of presenting a traditional form containing text boxes for entering numerical values, visualization-based controls would help the user better understand how the parameter values affect the underlying theory. Instead of presenting the model output using static summary tables and images, interactive visualizations would allow the user to filter and aggregate output data.
The technologies and languages listed in Table 1 also reflect the general evolution of web technologies as described in Section 2.1. Before 2005, all but one application required a browser plug-in (primarily a Java applet). When AJAX emerged in 2005, researchers shifted towards using JavaScript on the client. Visualizations were commonly generated on either the client or the server, and, in many cases, both systems. Simulation engines, however, have almost always been executed on the server indicating prevalence toward the remote and hybrid WBS architectures. Zeng et al. (2002) is the only example of the local WBS architecture in which the client performs the simulation and visualization within the browser through a Java applet. In summary, none of these applications demonstrate the local WBS architecture with both the simulation and visualization performed in the browser using only web standards, which is the approach used in this thesis.

2.4 Methodology

This section describes the novel methodology used in this thesis for creating interactive visualizations and model interfaces. Unlike existing research found in the literature, this methodology implements the local WBS architecture—where the simulation engine and visualizations are both executed within the web browser—using only standard web languages. By executing the model and generating the visualizations within the browser, this approach provides a high level of interactivity with the model output responding to changes in the parameters almost instantaneously. To demonstrate this approach, a simple application for generating a synthetic unit hydrograph is presented.
2.4.1 Theory
The unit hydrograph is a tool for creating synthetic hydrographs based on measured precipitation (NRCS, 2007). The shape of the hydrograph is governed by the specific properties of the watershed including the shape, soil type, topology, land use, etc. The details of the unit hydrograph and its uses can be found in NRCS (2007). The dimensionless unit hydrograph was chosen for this example because of its familiarity and simplicity.

The dimensionless unit hydrograph is computed using the gamma equation defined in Equation 16-1 of the NRCS National Engineering Handbook (NRCS, 2007):

\[
Q_u(t) = \frac{Q(t)}{Q_p} = e^m \left[ \left( \frac{t}{T_p} \right)^m \right] \left[ e^{\left( -m \left( \frac{t}{T_p} \right) \right)} \right]
\]

(2-1)

where \(Q_u\) is the unit flow rate, which is the actual flow \(Q\) scaled by the peak flow rate \(Q_p\) such that the peak value of \(Q_u\) is one. There are two parameters including \(m\), which is a shape factor, and \(T_p\), which is the time when peak flow occurs.

2.4.2 User Interface
For the purposes of describing the underlying code and methodology, the user interface was designed to be as simple as possible. It includes three components: a line chart showing the calculated unit hydrograph and two sliders for adjusting the values of the parameters \((m\) and \(T_p\)). The user can adjust the parameters by dragging the sliders, which triggers a re-computation of the model equation and an update of the chart. The interactivity of the interface is illustrated in Figure 2-7. As the slider for parameter \(T_p\) is dragged from left to the right, the hydrograph chart
automatically updates, reflecting the increase in peak time and change in the overall shape of the hydrograph.

![Unit Streamflow](image)

**Figure 2-7: Unit Hydrograph Interaction**

### 2.4.3 HTML Document

The HTML document is comprised of three sections: a header, a body, and links to the JavaScript files (Figure 2-8). The header section specifies the title of the page and includes a link to a CSS stylesheet (style.css). The body section defines the three components of the interface: a chart and two sliders. Note that the chart element is initially empty. The chart is inserted into this element as an SVG element by the JavaScript program after the page is loaded as described below in Section 2.4.5. At the end of the body section of the page are two external links to JavaScript
files containing the D3 visualization library (d3.js) and the hydrograph application (hydrograph.js).

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8" />
  <title>Unit Hydrograph</title>
  <link rel="stylesheet" type="text/css" href="style.css">
</head>
<body>
  <!-- chart -->
  <div id="chart"></div>
  <!-- slider for parameter m -->
  <div>
    <input class="slider" type="range" min="0.1" max="5" step="0.1" value="1" name="m" />
    m: <span id="param-m">1</span>
  </div>
  <!-- slider for parameter Tp -->
  <div>
    <input class="slider" type="range" min="0.1" max="3" step="0.1" value="1" name="Tp" />
    Tp: <span id="param-Tp">1</span>
  </div>
<script type="text/javascript" src="d3.js"></script>
<script type="text/javascript" src="hydrograph.js"></script>
</body>
</html>
```

Figure 2-8: Unit Hydrograph HTML Document

### 2.4.4 CSS Stylesheet

The CSS stylesheet (style.css) defines the font (sans-serif), and the color and width of the lines on the plot (Figure 2-9).
2.4.5 JavaScript Program
The application code is located in the hydrograph.js JavaScript file. At the top of
the file, three global variables are defined: a parameters object for storing the
values of the model parameters ($m$ and $T_p$), and a lineFunction object used to
update the chart representing the computed hydrograph. These variables are global
in scope and thus accessible to any other function in the script.

```javascript
// global variables
var parameters, lineFunction;
```

Next, the `window.onload` variable is assigned to a function that will initialize
the application. This function first sets the initial parameter values to the global
parameters variable, and then calls two functions to 1) create the SVG chart,
and 2) bind the slider events to re-compute the model output and update the chart.

By setting this function to the global variable `window.onload`, it will be
automatically executed once the HTML document and all externally linked files have been loaded into the browser.

```javascript
window.onload = function() {
    parameters = {Tp: 1, m: 1}; // set initial parameter values
    createChart(); // create svg chart
    bindSliderEvents(); // bind slider events
}
```

The `createChart` function uses the D3 library to create an SVG element for showing the hydrograph. This element will be inserted into the `div` element with class `chart` during initializing (Figure 2-8). Because D3 is a low-level library, all chart components including the scales, axes, and lines are individually constructed. For more details about this process, the reader is referred to the D3 documentation (Bostock, 2014). At the end of the `createChart` function, the `updateLine` function is called to compute the initial model solution and draw the hydrograph on the chart.

```javascript
function createChart() {
    // define size and margins of chart
    var margin = {top: 30, right: 20, bottom: 30, left: 50},
        width = 500,
        height = 300;

    // set up x and y scales that values to pixel positions
    var xScale = d3.scale.linear().nice()
        .range([0, width - margin.left - margin.right])
        .domain([0, 10]);
    var yScale = d3.scale.linear().nice()
        .range([height - margin.top - margin.bottom, 0])
        .domain([0, 1]);

    // define x and y axes
    var xAxis = d3.svg.axis().orient("bottom").scale(xScale),
        yAxis = d3.svg.axis().orient("left").scale(yScale);

    // create line function that maps arrays to pixel locations
    lineFunction = d3.svg.line()
        .x(function(d) { return xScale(d[0]);})
        .y(function(d) { return yScale(d[1]);});
```
The `updateLine` function performs the model computation using the current values stored in the global `parameters` variable. It then updates the model output data bound to the chart line representing the hydrograph, and updates the corresponding SVG element.

```javascript
function updateLine() {
    // compute model output based on current parameter values
    var data = compute(parameters['m'], parameters['Tp']);
    // select lines and bind new data
    updateLine();
}
```
The `compute` function performs the model computation by taking two arguments representing the values of parameters $m$ and $T_p$, and returning a two-dimensional array representing the computed flow time series.

```javascript
function compute(m, Tp) {
    // array of time points, e.g. [0, 0.01, 0.02, ..., 10]
    var times = d3.range(0, 10, 0.01);

    // empty array to store computed [time, flow] data points
    var results = [];

    // for each time, compute flow and add [time, flow] to results
    for (var i = 0; i < times.length; i++) {
        var time = times[i];
        var flow = Math.exp(m) * Math.pow(time/Tp, m) * (Math.exp(-m*time/Tp));
        results.push([time, flow]);
    }

    return results;
}
```

Finally, the `bindSliderEvents` function performs the important task of binding the interface controls to the model and output chart. The `d3.selectAll` function is used to select all elements with class `slider`, which correspond to the two `<input>` elements defined in the HTML document (Figure 2-8). Then, for each slider, the `input` event is bound to an anonymous response function. This response function will first update the text that shows the parameter value beside each slider (Figure 2-7). Then, the value of the corresponding parameter will be set in the global `parameters` object. Finally, the `updateChart` function is called to re-compute the solution with the updated parameter value and re-draw the output.
line on the chart. Note that the this variable is a special JavaScript variable that refers to which slider is currently being moved. The name and value attributes of this (i.e. this.name, this.value) refer to the parameter name (i.e. m or Tp) and the value currently set on the slider. Because the input event is automatically triggered by the DOM whenever the slider is dragged by the user, this function is executed immediately in response to the user’s actions. Thus the event binding performed in this function is the key component that couples the interface controls with the model computation and output visualization.

function bindSliderEvents() {
    // for each slider, bind input event to update chart
    d3.selectAll(".slider").on("input", function() {
        // update slider label with current value
        d3.selectAll("#param-"+this.name).text(this.value);

        // set value in parameters object
        parameters[this.name] = +this.value;

        // update chart with new parameters
        updateChart();
    });
}

This simple example illustrates the core components of the methodology used in Chapters 4-6 of this thesis. The important element is the event binding that couples the user actions with the model computations and visualization. The demonstrations described later in this thesis use more advanced architectures and programming patterns, including multiple linked charts and the ability to load external datasets into the application. However, this fundamental concept of event binding enables the high level of interactivity and responsiveness demonstrated by these applications. Although the interactivity and responsiveness of this approach is
limited to relatively simple models that can be executed in a fraction of a second, the demonstrations presented later in this thesis suggest the performance of modern JavaScript engines is sufficient for a variety of potential use cases.

2.5 Summary and Conclusions

Over the past two decades, the web has evolved from a collection of static documents to a new platform capable of supporting interactive applications. The technologies available today offer new opportunities for creating dynamic and interactive user interfaces that rival traditional desktop software. Client-side web applications provide a rich user experience through AJAX-style communications, HTML5 APIs, and the high performance JavaScript engines included in modern web browsers. Using these new technologies, numerical calculations and interactive visualizations can be performed within the browser using only web standards.

Buytaert et al. (2012) describe the potential for modern web technologies to facilitate environmental management and decision making, improving the transparency and accessibility of scientific knowledge for the general public. In their conclusions, they state that: “Many challenges will need to be addressed of how to design models, link them into networks, and let people interact with them.” Based on a review of related research in environmental and water resources engineering, it is clear that much work is being devoted to the first two challenges in this list. New cyber infrastructure is being developed to expose models and datasets using web services. These services will allow model developers to create networks of reusable and interoperable models that can be coupled with data.
sources. However, in order to use these services, there must be client applications providing intuitive and easy-to-use user interfaces. Although map-based interfaces provide a more powerful means of displaying and discovering simulated and observed datasets, new kinds of web-based interfaces for configuring models, assigning parameters, and analyzing results are needed.

This thesis demonstrates a novel approach to web-based environmental modeling by providing interactive visualizations and user interfaces built with modern client-side web technologies. Existing research in this field has focused primarily on server-side architectures, which limit the interactivity of the user experience. As this thesis will clearly demonstrate, client-side technologies provide a number of new features and capabilities that improve web-based user interfaces by making models easier to both understand and use. Although this methodology will be demonstrated for relatively simple models, the technologies and general approach can be used to create interactive interfaces for virtually any web-based modeling application, whether the model is executed on the server or the client.
Chapter 3

Web-based Interactive River Model

This chapter presents a client-side web application for interactive water quality modeling called the Web-based Interactive River Model (WIRM). WIRM was the first project to be completed for this research. The primary goal was to evaluate the overall feasibility of developing client-side web applications for environmental modeling. The results indicated that not only is this approach feasible, but that it provides for a new kind of interactive user experience that improves the ability to understand model behavior. These results led to new research questions and additional demonstration projects as presented in Chapters 4–6. Specifically, this project led to investigations into how and why interactive visualizations improve model understanding and how additional application features and functionality such as the incorporation of input and observation data could be integrated into this approach. This application is available online at:

http://wirm.walkerjeff.com

The WIRM application resulted in the following peer-review publication. The remainder of this chapter is a reproduction of this publication and was reformatted to maintain consistency throughout this thesis.


54
Abstract

Recent developments in web technologies including evolution of web standards, improvements in browser performance, and the emergence of free and open-source software (FOSS) libraries are driving a general shift from server-side to client-side web applications where a greater share of the computational load is transferred to the browser. Modern client-side approaches allow for improved user interfaces that rival traditional desktop software, as well as the ability to perform simulations and visualizations within the browser. We demonstrate the use of client-side technologies to create an interactive web application for a simulation model of biochemical oxygen demand and dissolved oxygen in rivers called the Web-based Interactive River Model (WIRM). We discuss the benefits, limitations and potential uses of client-side web applications, and provide suggestions for future research using new and upcoming web technologies such as offline access and local data storage to create more advanced client-side web applications for environmental simulation modeling.
3.1 Introduction

Effective management of environmental systems requires understanding the impacts of humans on complex physical, chemical, and biological processes. Engineers and scientists often use simulation models to perform numerical experiments that compare the effectiveness of alternative policies and management strategies for restoring and protecting natural ecosystems. While models have long been an integral part of the environmental decision making process, the abilities of resource managers, policy makers, and interested stakeholders to understand how these models work, what they represent, and their underlying assumptions are limited due to the technical expertise required to develop and use these models (NRC, 1999).

Without direct access to models, decision makers and stakeholders often rely on modeling experts to develop, run, and interpret simulation models on their behalf (Loucks et al., 1985; Booth et al., 2011). This process can create a bottleneck in the flow of knowledge attained through simulation modeling and hinder stakeholder creativity in developing new alternative scenarios because they are unable to directly translate their ideas into the modeling framework (Loucks et al., 1985). As a result, this dependence limits overall stakeholder participation, which is an essential component of the environmental decision making process (Voinov and Bousquet, 2010). Furthermore, this barrier affects not only the ability of stakeholders to directly use and understand models, but also the ability of modeling experts to effectively communicate model results and interpretations with clients and stakeholders.
In the 1980’s, the advent of the personal computer (PC) brought a new era to the development and use of simulation models through widely available and inexpensive computing power accessible through new graphical user interfaces. Loucks et al. (1985) predicted that interactive and easy-to-use interfaces would allow policy makers and stakeholders to better understand and utilize models, which in turn would lead to more informed decision making and wider model acceptance. Chapra and Canale (1987) discussed the application of spreadsheet-based models and animated graphical displays to facilitate the communication of models to stakeholders. While the PC has indeed contributed to advances in environmental modeling over the past 25 years, the barriers between models and non-technical users persist due to insufficient advances in model interfaces, complexity of model software, input data requirements, and difficulties associated with post-processing and interpreting voluminous output data (NRC, 1999; Buytaert et al., 2012). Furthermore, access to desktop-based modeling software can be limited due to licensing restrictions for multiple users as well as incompatibilities between different platforms, operating systems or even software versions. As a result, current desktop-based modeling software is often not only hard to use, but can also be difficult for decision makers and stakeholders to access directly on their own computer.

Over the past 20 years, the World Wide Web has revolutionized human-computer interactions by providing an engaging and accessible platform for users of all technical abilities. Initially, the web was comprised of relatively simple web pages containing text, hyperlinks, and static images. Over time, the evolution of web
standards and development practices gave rise to increasingly sophisticated web applications such as email clients, office suites, photo editors, e-commerce storefronts and social networks. The success and widespread adoption of these applications is attributable in part to the design of intuitive user interfaces that are comprehensible to the average web user, as well as increased utilization of client-side approaches to web application development where more of the application code is transferred from the server to the client. By moving the application into the browser, client-side web applications can provide a more interactive and responsive user experience by directly updating web page content from within the browser, and thus requiring less content be generated by the server. For simulation modeling, this client-side approach allows for not only improved user interfaces to access, visualize and analyze the results of models executed on the server, but also the possibility of performing the model simulations within the browser itself.

The use of web applications for the purpose of simulation modeling is sometimes referred to as web-based simulation (WBS) where the primary model interface is accessed through a web browser (Byrne et al., 2010). The three primary architectures for WBS include the remote architecture where both the model simulations and output visualizations are performed on the server, the local architecture where the simulations and visualizations are performed in the browser, and the hybrid architecture where the simulations are performed on the server and visualizations generated in the browser (Byrne et al., 2010). Traditionally, WBS applications were based on the remote architecture due in part to the computational limitations and lack of development tools for supporting browser-based simulation
and visualization. For many years, the local WBS architecture could only be implemented using browser plug-ins such as Java applets\(^1\) and Adobe Flash\(^2\), which provided the tools and features needed to perform model simulations and visualizations in the browser (Byrne et al., 2010). Today, client-side web applications provide a new approach to the local WBS architecture by facilitating interactive model interfaces built using only web standards.

For environmental applications, web-based simulation has been an area of great interest and active research as investigators explore various ways of harnessing the web for model development, execution, and access (Buytaert et al., 2012; Laniak et al., 2013). Many examples of web applications can be found in the environmental modeling literature for discovering and accessing data (Goodall et al., 2008; Huang et al., 2011), performing web-based simulations (Lim et al., 2001; Bacu et al., 2011; Booth et al., 2011; Feng et al., 2011; Goodall et al., 2013; Sun, 2013) and evaluating model uncertainty (Bastin et al., 2013). The majority of this existing research has primarily relied on server-side approaches where both the application and model are executed on the server. Some recent research has demonstrated the hybrid WBS architecture where a client-side web application is used to provide an interactive user interface for visualizing model results that are generated on the server (e.g. Booth et al., 2011). However, the use of client-side applications in conjunction with the local WBS architecture to perform both the simulation and visualization solely within the browser has yet to be explored.

In this paper, we demonstrate the use of modern web technologies and development practices to create an interactive, client-side web application for a water-quality simulation model of dissolved oxygen in rivers and streams. This work illustrates not only the potential for client-side web applications to provide a modern and interactive user interface, but also the possibility of implementing the local WBS architecture by performing the simulation and visualization entirely within the browser using only web standards. In the next section, we begin with a brief review of web-based simulation and discuss the trade-offs between alternative client-server architectures. We follow with a summary of current web standards and development practices for creating modern client-side web applications. We then demonstrate how these technologies can be leveraged to design and implement a client-side application using the local WBS architecture where the browser performs both the numeric computations and output visualizations, and the server simply acts as a repository for storing model configurations as well as a platform for collaboration. We conclude with a discussion on the benefits, limitations and potential uses of client-side web applications for both local and remote execution of simulation models in general, and suggest ways of incorporating new and emerging client-side web technologies to support more complex simulation models.

3.2 Overview of Web-based Simulation

Web-based simulation (WBS) refers to the use of web technologies to develop, execute, and analyze simulation models where the primary interface is accessed through a web browser (Byrne et al., 2010). Not long after the web was introduced in the early 1990s, simulation modelers began discussing the potential advantages
and limitations of using the web as a platform for simulation modeling (Fishwick, 1996). Byrne et al. (2010) provide a comprehensive review of WBS including the advantages and disadvantages of various client-server architectures. Unlike traditional desktop simulation software where the entire application is installed and executed in a single environment (i.e. the user’s local computer), web applications involve two (or more) physically separate systems: the client and the server. Different architectures for WBS applications are often characterized by how the components of the application are divided between these two systems.

Byrne et al. (2010) define three primary WBS architectures – local, remote and hybrid – based on which system the simulation engine and visualization renderer are each located and executed. A local architecture is primarily client-based whereby both the simulations and visualizations are executed within the user’s web browser; the server simply provides the initial application code to the client and stores model configurations, input and output datasets, and other persistent data in a remote database. In contrast, a remote architecture is primarily server-based with the simulation engine and visualization renderer both executed on the server; the client then plays a minor role in displaying the text, data and static graphics generated by the server. Between these two extremes is a hybrid architecture where the simulation engine resides on the server and visualizations are created in the browser.

The choice of WBS architecture depends in part on the complexity and computational demands of the simulation model and visualizations. While the remote architecture can provide far greater computing power, especially when
coupled with distributed computing resources (i.e. cloud computing), local simulation and visualization in the browser eliminates the latency associated with client-server communications resulting in a more dynamic and interactive user experience. Local simulation and visualization also reduces server demand allowing simpler and less costly server architectures, and improves the scalability of the application for supporting many concurrent users (Byrne et al., 2010).

In the environmental field, web-based approaches to simulation modeling have been primarily focused on server-side approaches where the model is executed on the server and results accessed either through web services or server-generated web pages. Web services are server-side interfaces that accept requests from clients containing input data, perform a computation, and return a response containing output data (Castronova et al., 2013). This service oriented approach is useful for linking models and data repositories to create an integrated modeling platform (Goodall et al., 2011; Goodall et al., 2013). The remote and hybrid architectures are also commonly used when migrating existing simulations models to the web as these models are often computationally demanding and their legacy code is not easily translated to the standard client-side programming language, JavaScript (e.g. SPARROW (Booth et al., 2011), SWAT (Bacu et al., 2011)).

In contrast, there are few, if any, examples of environmental simulation models that harness the high degree of interactivity provided by the local WBS architecture. By performing both the simulation and visualization directly in the browser, changes in model parameter values or algorithms can be evaluated more quickly than if the client must request updated simulations from the server. While this approach is
limited by the amount of computing power available in the browser, it can be used for a variety of real-world applications including rapid screening models, model documentation and education. Further discussion of the limitations and potential uses of the local WBS architecture are provided in the Discussion (Section 3.5).

3.3 Modern Web Application Development

As the web evolved from a collection of static web pages to more sophisticated web applications, developers initially relied on server-side scripting languages such as PHP[^3] or Active Server Pages (ASP[^4]) to dynamically render content in response to a user’s request. In a typical server-side web application, the server would receive a request from a user for some information, extract the relevant data from a database, and then format that data for display in a web browser using Hypertext Markup Language (HTML). The resulting HTML document would then be sent back to the client and rendered by the browser. With server-side web applications, the server would thus perform the application logic by deciding how to translate a user action into HTML content that can be displayed by the client.

As web technologies evolved, developers increasingly turned to client-side web applications to manage more of the application within the browser itself. This approach allows the application to directly control and manipulate the interface without having to load an entirely new page from the server. The development of Asynchronous JavaScript and XML (AJAX) communications allowed client applications to request the raw data directly from the server, format the data using

HTML, and then update the page contents directly, all from within the browser. Using this approach, client-side applications could thus improve the response time of the interface by requesting, formatting and injecting content directly without requiring a refresh of the current web page or navigation to a new page.

For many years, development of client-side web applications was hindered by the lack of features provided by standard web technologies, incompatibilities between different browser versions and vendors, and difficulties in developing well-organized and maintainable application code in JavaScript. Web developers were forced to rely on third-party browser plug-ins such as Java applets or Adobe Flash to create client-side applications with interactive user interfaces that would appear and behave identically across all major browsers. Today, the evolution of standard web technologies and their adoption by major browser vendors are eliminating the use of plug-ins by providing the features and capabilities needed to develop client-side web applications. As a result of this shift towards standards-based web applications, browser plug-ins have decreased in popularity and are no longer widely supported across all Internet-enabled platforms, especially on mobile devices. Together, these advances are driving a general trend in web application development away from server-centric applications, where the browser simply displays content generated by the server, to the concept of “browser as a platform” where the client manages the application and the server simply acts as a central repository for the application code and remote data storage (Hales, 2013).
3.3.1 Advances in Web Technology Standards

Web standards are open, community-driven specifications that define the protocols, languages, and interfaces for powering the web. The primary standards relevant to web application development include:

- **Hypertext Markup Language (HTML)**: a tag-based markup language for storing content in a semantic structure.
- **Cascading Style Sheets (CSS)**: a language that defines the styling and formatting rules to display HTML content.
- **Scalable Vector Graphics (SVG)**: an XML-based language for generating vector graphics.
- **Document Object Model (DOM)**: a tree-based data structure for storing, rendering and manipulating individual elements on a page.
- **ECMAScript (aka JavaScript)**: a client-side scripting language that, when combined with the DOM, is commonly referred to as JavaScript and used to manage events, manipulate elements on a page, perform asynchronous communication (AJAX) with the server, and provide other dynamic and interactive features.

These standards have been critical to the widespread adoption and success of the web, and contributed to its accessibility and interoperability across diverse hardware and software platforms. Evolution of these standards has allowed the web to progress from a collection of hyperlinked documents containing only text and images, to a broad range of applications such as e-mail clients, office suites, photo editors, and social networks. The latest versions of these standards, in particular
HTML version 5 (HTML5) and new specifications for JavaScript Application Programming Interfaces (APIs), provide many of the features commonly found in traditional desktop software such as a variety of user interface controls, elements for data visualization, offline access, file system access, and local data storage (Kessin, 2012) (see Section 3.5.1 for further discussion of these new web technologies). With these new features, the development of modern web applications now relies on effective integration of web standards so that developers can focus on a single code base and interface, which is then accessible across all Internet-enabled devices and platforms (Bostock et al., 2011).

3.3.2 Advances in JavaScript Development Practices and Performance
Following its creation in 1996, JavaScript was long considered an inferior language by programmers and used for little more than simple manipulation of page elements or generating alert pop-ups (MacCaw, 2011). Using JavaScript to develop full-fledged applications was extremely difficult due to its lack of modules and object-oriented classes as well as variations in its implementation by the different browsers, all of which led to code that was difficult to develop, organize, and maintain (so-called “spaghetti code”). Current best practices for JavaScript development focus on employing the “good parts” of the language and avoiding the more troublesome code syntaxes and patterns widely known to cause bugs and errors (Crockford, 2008). As these practices were adopted by the developer community, JavaScript was recognized as a more powerful and pragmatic programming language that could be used for increasingly complex client-side applications.
JavaScript development has also benefited from a diverse ecosystem of free and open-source software (FOSS) libraries that eliminate common cross-browser compatibility issues and provide a high-level interface to common programming tasks. The jQuery\(^5\) library, for example, is perhaps the most widely used JavaScript library and provides a simple interface to many common functions such as selecting elements on a page, applying style classes, and performing asynchronous communication with the server. Without jQuery or similar foundational libraries, these tasks would require multiple versions of the same code to support different browser vendors and versions. Other libraries such as Underscore.js\(^6\) provide a suite of functional utilities for common programming tasks involving various types of objects such as arrays and functions as well as providing implementations of newer JavaScript functions that are not available in older browser versions. These foundational libraries can save considerable time and effort by allowing developers to focus more on their application and less on the details necessary to ensure the application behaves properly across multiple browsers and platforms. Building on these libraries, more advanced application frameworks such as Backbone.js\(^7\), Ember.js\(^8\), and AngularJS\(^9\) have emerged that facilitate the development of complex applications employing well-organized and maintainable code.

JavaScript application frameworks are generally based on variations of the Model-View-Controller (MVC) pattern commonly used in traditional desktop software

---


development (Gamma et al., 1994). The MVC pattern defines a systematic way of separating application data (Model), presentation (View), and logic (Controller) into modular components that can be reused in different contexts and applications. These libraries provide web developers with the tools needed to create complex application workflows through code that is far easier to organize and maintain than when using plain JavaScript. Another major benefit of using these frameworks is that applications can be developed solely using web standard (HTML, CSS, JavaScript) without the need for browser plug-ins. This facilitates more rapid and robust application development and allows developers to follow the “Write Once, Run Anywhere” (WORA) principle whereby a single application code base can be run on virtually any web-enabled device or platform (Hales, 2013).

In addition to the foundational JavaScript libraries and application frameworks, a number of visualization libraries are now available that can generate charts and graphs within the browser. By utilizing the new HTML5 canvas element or the Scalable Vector Graphic (SVG) standard, visualization libraries can create client-side visualizations that are dynamically rendered by the browser itself. This ability to generate in-browser visualizations is an important milestone for supporting scientific web applications, which often require some form of data visualization. Together, these visualization and application libraries provide a more complete development environment for creating client-side web applications that incorporate many of the features found in desktop-based scientific computing environments.

Another long-held criticism of JavaScript was its slow execution speed due to the fact that it is a weakly typed, interpreted language. However, substantial
performance gains have recently been achieved through optimization of the browser engines that translate JavaScript to executable machine code (e.g. Google Chrome’s V8\textsuperscript{10}). One cursory comparison of execution times for seven common numerical algorithms suggested that JavaScript was no more than 10 times slower than compiled C code, while Python, Matlab, Octave and R were orders of magnitude slower than C (Julia Language, 2013). Further performance gains can be attained through the upcoming Web Workers\textsuperscript{11} API, which provides support for multi-threaded web applications, and the WebGL\textsuperscript{12} API, which adds browser support for hardware acceleration using the local computer’s graphics processing unit (GPU). With these performance gains and advances in JavaScript development practices and libraries, client-side web technologies are now capable of supporting more interactive user interfaces using the local WBS architecture to perform numeric calculations and dynamic visualizations within the browser.

3.4 Design and Implementation of a Client-based Web Simulation Model

The Web-based Interactive River Model (WIRM) is a client-side web application for simulating the effect of biochemical oxygen demand (BOD) discharge on dissolved oxygen (DO) concentrations in rivers and streams (available at http://wirm.walkerjeff.com/). By performing all numerical calculations and visualizations within the browser, WIRM demonstrates the ability of client-side web applications to support the local WBS architecture, which has not been as

\textsuperscript{10} http://code.google.com/p/v8/, accessed June 12, 2013.
\textsuperscript{11} http://www.w3.org/TR/workers/, accessed November 27, 2013
\textsuperscript{12} http://www.khronos.org/webgl/, accessed November 27, 2013
widely explored as the remote or hybrid WBS architectures for environmental modeling. The primary design objectives of WIRM were to:

- Use only web standards (e.g. HTML, CSS, SVG, JavaScript) to create a client-side web application that handles the numerical computation, output visualization and user interface control within the browser,
- Use FOSS libraries and frameworks on both the client and server that enable low-cost and rapid application development that adheres to modern best practices in web application development,
- Use a Resource Oriented Architecture (ROA) for retrieving, storing, and updating user-defined data, and
- Support user collaboration through a commenting system and URL-based access to saved model configurations.

3.4.1 Model Application
The mathematical model in WIRM is a non-linear version of the Streeter-Phelps model for simulating the effect of biochemical oxygen demand (BOD) discharge on in-stream dissolved oxygen (DO) concentrations (Streeter and Phelps, 1925). The model is defined by a system of two ordinary differential equations (ODEs) representing the in-stream concentrations of BOD and DO subject to BOD decay and DO reaeration. The equations are solved numerically using the 4th-order Runge-Kutta numerical method to compute the steady-state concentrations of BOD and DO as a function of distance downstream from a point source discharge (Chapra, 1997). The model’s mathematical details are provided in Section 3.7. Note that we deliberately chose a simple model so as to keep the focus of this work on
the design and implementation of the web application and interface rather than on
the complexity of the mathematical model itself.

3.4.2 Application Architecture
The server and client architectures for WIRM were designed to be independent in
order to demonstrate the ability of client-side applications to run separately from
the server. The server primarily acts as a central repository for providing the initial
client application code that runs in the browser, storing user-generated data in a
remote database, and managing user authorization. The client-side application
includes the simulation engine, visualization renderer, and user interface, which are
all managed by a central application controller in the browser. Figure 1 shows a
schematic diagram of the overall architecture for the WIRM web application.
3.4.2.1 Server-side Architecture

The server-side architecture for WIRM was built using the Django web framework, which is an open-source framework based on the MVC design pattern and implemented in Python (Django Software Foundation, 2013). Although it was
originally developed for creating news websites, Django is well-suited for environmental modeling applications as developers can leverage the extensive numerical computing and geographic information system (GIS) libraries available for Python. For example, Sun (2013) used Django to develop a GIS-based web application for the popular nutrient loading model, PLOAD.

Django facilitates the web development process by providing a number of built-in modules containing common web application features such as user account management, relational database integration, and best security practices, which are very difficult and time consuming to create from the ground up. Django can also be extended through open-source plug-ins that provide additional features not available in the core modules such as the django-registration\textsuperscript{13} plug-in for registering new user accounts, and the django-rest-framework\textsuperscript{14} plug-in for passing data between the client and server over a Representational State Transfer (REST) API, both of which are used by WIRM.

To exchange data between the client and server, WIRM uses a Resource Oriented Architecture (ROA) on the server accessed through a REST API. The concepts of ROA and REST are based on the idea that server-side data can be represented as collections of resources and that the common methods of creating, retrieving, updating and deleting these resources can be accomplished using the standard HTTP protocols of POST, GET, PUT, and DELETE, respectively (Fielding, 2000). ROA is an alternative to the Service-Oriented Architecture (SOA), which provides

\textsuperscript{13} https://bitbucket.org/ubernostrum/django-registration/, accessed June 12, 2013.
\textsuperscript{14} http://django-rest-framework.org/, accessed June 12, 2013.
a set of functions that can be accessed using the Simple Object Access Protocol (SOAP) or the XML-based Remote Procedure Call (XML-RPC) (Castronova et al., 2013). Recent examples of environmental modeling applications primarily use SOA to create data discovery and retrieval systems as well as component-based and integrated modeling services (e.g. Goodall et al., 2011; Huang et al., 2011; Goodall et al., 2013). However, ROA is increasingly recognized as a viable alternative to SOA due to its simplicity and ease of implementation (Richardson and Ruby, 2007). Granell et al. (2013) provide a review of the ROA-based approach to integrated environmental modeling where models and associated data are all represented as linkable resources.

The REST API for WIRM provides access to four types of resources: Parameters, Projects, Comments, and Users. Parameters define the set of input parameters needed to compute the model solution and include the default value for each parameter (see Section 3.7). Projects contain user-defined parameter values and associated project metadata such as a title, location and description for storing a system-specific instance of the model on the server. Comments contain short text messages associated with a specific Project and User. Finally, User resources represent user accounts for authentication and authorization and for identifying which user originally created each Project and Comment. All resources are transmitted over the REST API using JavaScript Object Notation (JSON), which is commonly used in modern web applications as it is easier to handle in the client and less verbose than more traditional XML-based formats.
The underlying data for the resources are stored in a MySQL relational database on the server. The object relational mapper (ORM) in Django provides an object-oriented wrapper around the SQL interface allowing the developer to perform the standard CRUD (Create, Retrieve, Update, and Delete) operations using familiar Python syntax instead of complex SQL statements. Access to the REST API is controlled by Django’s user authentication system, which prevents unauthorized users from making changes or deleting data on the server. The authentication system is also used to only allow the original user who created a project to save changes on the server; all other users can view and locally modify an existing project, but they cannot save changes on the server.

### 3.4.2.2 Client-side Application

The client-side application for WIRM includes the user interface, simulation engine and visualization renderer, all of which are executed in the browser. Figure 2 is a screenshot of the user interface following initializing of the client application. To incorporate these components in a well-organized application structure, the client application was developed using the FOSS libraries and frameworks listed in Table 1.
Figure 3-2: Screenshot of the WIRM user interface.

<table>
<thead>
<tr>
<th>Library</th>
<th>Version</th>
<th>Purpose</th>
<th>Website</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bootstrap</td>
<td>2.3.1</td>
<td>CSS styling, responsive layout, user interface components</td>
<td><a href="http://twitter.github.io/bootstrap/">http://twitter.github.io/bootstrap/</a></td>
</tr>
<tr>
<td>jQuery</td>
<td>1.9.1</td>
<td>DOM manipulation, event handling, AJAX communication</td>
<td><a href="http://jquery.com/">http://jquery.com/</a></td>
</tr>
<tr>
<td>jQuery-ui</td>
<td>1.10.1</td>
<td>Interactive user controls</td>
<td><a href="http://jqueryui.com/">http://jqueryui.com/</a></td>
</tr>
<tr>
<td>Underscore</td>
<td>1.4.4</td>
<td>View templating, miscellaneous JavaScript utilities</td>
<td><a href="http://underscorejs.org/">http://underscorejs.org/</a></td>
</tr>
<tr>
<td>Numeric</td>
<td>1.2.6</td>
<td>Element-wise arithmetic operations of numeric arrays</td>
<td><a href="http://www.numericjs.com/">http://www.numericjs.com/</a></td>
</tr>
<tr>
<td>Backbone</td>
<td>1.0.0</td>
<td>Client application architecture</td>
<td><a href="http://backbonejs.org/">http://backbonejs.org/</a></td>
</tr>
<tr>
<td>D3</td>
<td>3.0.8</td>
<td>Data visualization</td>
<td><a href="http://d3js.org/">http://d3js.org/</a></td>
</tr>
</tbody>
</table>

Table 2: FOSS JavaScript libraries used by the WIRM client-side web application.
3.4.2.2.1 Front-end Framework

The overall layout and style of WIRM was developed using Twitter’s Bootstrap\(^{15}\) front-end framework, which is described as a “sleek, intuitive, and powerful front-end framework for faster and easier web development.” Bootstrap includes CSS and JavaScript code that provide baseline style and layout rules for common web page elements such as grids, navigation bars, buttons, and dialog boxes. Bootstrap also incorporates the normalize.css\(^{16}\) style sheet to eliminate many common cross-browser compatibility issues associated with webpage styling and formatting rules.

3.4.2.2.2 JavaScript Application Framework

The client-side application was built using Backbone.js, which, like Django, is based on the MVC design pattern and explicitly separates the application data, presentation and logic. Backbone applications are comprised of four components: models, collections, views and routers. Models define the structure of JavaScript objects that store application data, such as a parameter or a project. Collections are groups of models of the same type (e.g. a parameters collection contains all of the individual parameter models). Views define the elements used to display models on the web page by injecting model data into HTML templates. Finally, the router acts as a central application controller and is used to select the appropriate view based on the browser’s current URL.

In order to synchronize data between the client application and the server, Backbone provides a simple interface for accessing the REST API using


Asynchronous JavaScript and XML (AJAX) communication. The resources provided through WIRM’s REST API are stored locally in the browser as Backbone models and collections. When the user changes parameter values or edits project information, only the local versions of these resources are modified until the user requests that this information be saved to the server. To save data to the server, the local data are sent to the REST API in JSON format using the appropriate HTTP method (POST for creating, PUT for updating).

The user interface is constructed as a series of nested Backbone views. Each view is based on an HTML template that defines how a data object is presented on the web page. By defining a view for each component of the interface, the views can be reused for multiple objects of the same type. For example, a parameter view defines how each parameter is rendered by defining the set of HTML elements and CSS styles to display its title, units, and value as well as an interactive slider. This parameter view can then be reused for each parameter to create the complete set of slider controls for all input parameters.

Another key feature of an application framework like Backbone is the event manager that controls how the application behaves in response to user actions. For example, when the user moves one of the parameter sliders, an event is triggered that is captured by the corresponding view and the underlying parameter model is updated with the new value. The change in parameter value then triggers the simulation engine to re-compute the model solution, which in turn causes the output chart to refresh. Use of the MVC architecture coupled with the JavaScript event
manager allows for highly interactive interfaces built using well-organized and modular code.

### 3.4.2.2.3 Simulation Engine

The simulation engine that computes the numerical solution to the system of differential equations (see Section 3.7) was implemented in JavaScript so that it could be executed in the web browser. The implementation of this algorithm was facilitated by the numeric.js\(^{17}\) library, which provides functions for performing element-wise computations of numeric arrays, a feature not natively supported by JavaScript. These element-wise operations allow the developer to more easily perform array and matrix algebra commonly available in desktop computing environments.

### 3.4.2.2.4 Output Visualization

The results of the model simulation are visualized as a line chart showing the concentrations of BOD and DO as a function of distance downstream from a point source discharge. The charts are generated as SVG images within the browser using the D3 (Data-Driven Documents)\(^ {18}\) JavaScript library (Bostock et al., 2011). D3 provides a functional approach for binding data directly to elements on the page (in this case, the model output arrays are bound to line elements in the SVG chart). Through data binding, D3 can automatically update chart elements in response to

---


changes in the underlying data objects, which facilitates dynamic and interactive visualizations.

3.4.3 Application Testing
To ensure WIRM is compatible across browsers and platforms, it was tested using the BrowserStack\textsuperscript{19} testing suite, which provides access to multiple versions of the major web browsers and operating systems. WIRM appeared and behaved identically in most versions of the major web browsers including Google Chrome\textsuperscript{20} 4+, Mozilla Firefox\textsuperscript{21} 3+, Microsoft Internet Explorer\textsuperscript{22} 9+, Opera\textsuperscript{23} 9+, and Apple Safari\textsuperscript{24} 3.2+ on both Windows and Mac OSX operating systems. The only major browser incompatibility is for Microsoft Internet Explorer versions 8 and below, which do not support SVG graphics.

3.4.4 Application Workflow
In this section, we describe a typical use case for WIRM to highlight the application logic and client-server communications. Figure 3 illustrates the communications between the user, client application, server, and remote database for some of the common tasks performed by the user. For each request from the client to the server, the corresponding HTTP method and URL is shown to demonstrate use of the REST API.

\textsuperscript{22} http://www.microsoft.com/ie, accessed June 14, 2013.  
When the user first visits the client URL (/client/), the server responds with the HTML, CSS and JavaScript files that define the client-side application. While the client-side application is being initialized, the browser requests the default parameter values from the server over the REST API by performing a GET request to the Parameters resource collection (/api/parameters/). The default parameters are then returned as JSON and used to render the parameter controls in the user
interface. Once the application is initialized, the user can create a representation of their specific river by changing the parameter values using the interactive sliders. When a parameter value is changed, an event is triggered causing the client to recompute the model solution and update the output chart. During this process, no data are exchanged between the client and the server as the model solution and visualization are both generated by the browser.

Once all the parameters are defined for the target river, the user can save the configuration to a new Project resource. After clicking the Save button, the user fills out a form containing the project title, location and description, which are then submitted along with the current parameter values to the REST API via a POST request to the Projects resource collection (/api/projects/). When the server receives this request, it validates the data, assigns a unique ID to the project, and saves the information as a new record in the database. With the unique ID, the client redirects the browser to the URL now assigned to the new project, and the user interface is updated to reflect the project data. The user can then continue to make modifications to the parameter values as necessary, saving these values on the server through an HTTP PUT request.

After the model is configured and saved to the server, the user can simply send the unique URL for that project to other collaborators, who will have immediate access to the model without needing to download desktop software or browser plug-ins.

After an existing project is loaded, any registered user can add comments for discussing interpretation of the model results. For example, the model developer could provide comments specifying the source of parameter values and boundary
conditions, and stakeholders could ask questions or provide suggestions for alternative values. The commenting system allows multiple users to create a running dialogue about how the model was created, what conditions it represents and what the results imply for management or decision making. This ability to share saved projects by URL and to support discussion and collaboration through the comment system are key features of the WIRM application that demonstrate the benefits of web-based approaches to simulation modeling for decision support.

### 3.5 Discussion

WIRM demonstrates the use of modern web standards and development practices to create an interactive user interface for web-based environmental simulation modeling. By using a client-side web application with the local WBS architecture, the numerical computations and output visualizations are both generated directly within the web browser, an approach that, to our knowledge, has not yet been presented within the environmental modeling community. We begin this section with a discussion of the potential uses and limitations of the WIRM application itself, and follow with a more general discussion on the benefits, limitations and potential uses of client-side web applications for both the local and hybrid WBS architectures.

For practical applications, WIRM can be used as a rapid screening model for investigating potential water-quality impairments due to BOD discharges. Screening models are often used when developing more sophisticated models, designing monitoring programs, or identifying the potential severity of water-quality impairments (Chapra, 1991; Anderson et al., 2004). Web-based screening
models such as WIRM can provide a quick and easy way to perform a rapid assessment of a system before turning to more sophisticated models or sampling programs. Additional models could be added to WIRM for simulating other common water-quality problems such as toxic releases or nutrient loading. This suite of models would provide a useful toolkit for watershed managers and decision makers who have limited resources or lack the technical expertise for using more complex models.

WIRM could also be a valuable educational resource for students and inexperienced modelers to better understand water-quality simulations and the effects of changing parameter values on the model output. By using the local WBS architecture where both the simulations and visualizations are generated within the browser, WIRM provides a high level of interactivity giving users the ability to visually evaluate the relationships between parameter values and the model output. We consider this interaction a form of visual sensitivity analysis that, although less scientifically rigorous than traditional sensitivity analysis that provides confidence intervals for model predictions, may be more comprehensible and intuitive to inexperienced modelers.

While WIRM demonstrates some potential uses for client-side web applications in environmental simulation modeling, it was created primarily to illustrate the possibility of performing numerical computations and dynamic visualizations within the browser and has a number of limitations for real-world applications. Perhaps most importantly, WIRM currently lacks support for model calibration and validation based on observation data. In this first version, calibration depends on
the user’s understanding of the model parameters such as their typical ranges and estimation methods, information that is currently provided in the documentation but may not be intuitive to all users. Future research will focus on integrating observed data in the application workflow and computing goodness of fit statistics to help guide model calibration.

The design of user interfaces is a critical component to the success and adoption of any web application. Although WIRM was designed to be accessible and intuitive to both technical and non-technical users, further research is needed to better understand what elements and design principles could improve its ability to facilitate model understanding. Extensive research in the field of human-computer interaction provides numerous guidelines and suggestions for creating user-friendly interfaces (e.g. Shneiderman et al., 2009), but identifying the design principles that are most applicable to web-based simulation for environmental management and decision making requires further research. This work could benefit from user studies that test whether web-based user interfaces for environmental applications are more intuitive and understandable than common desktop software interfaces, particularly for non-technical users such as stakeholders and decision makers.

WIRM also does not currently provide the abilities to load model configurations or extract model output in standard text-based file formats. Data input and output is a common problem in many web applications as the ability to access local files from the browser is restricted for security purposes. However, as discussed in the following Section 3.5.1, a number of new and upcoming web specifications are emerging that provide the ability to read and write text files on the local hard drive,
which will allow web applications to load and save data locally without passing files to and from the server. Future research will investigate alternative methods for loading model configuration files containing parameter values and for saving the model output to standard text files so that the results can be used outside of the web application.

Finally, WIRM does not currently provide any measure of uncertainty in the model output, which is an important element for any modeling application (Beven, 2009). Incorporating uncertainty through methods such as first order error analysis or Monte Carlo simulations could improve the utility of WIRM for real-world decision making. Future work will involve finding appropriate methods for estimating uncertainty and incorporating this in the model output to help users better understand the magnitude of various sources of uncertainty.

Despite these limitations, WIRM demonstrates a new approach to web-based modeling using client-side web applications and the local WBS architecture built on modern web standards. The ability to perform numerical computations and dynamic visualizations within the browser provides developers with the tools needed to create more interactive and user-friendly client interfaces for web-based modeling. In the following sections we describe the general benefits and limitations of web applications in comparison to traditional desktop software, and then discuss some potential uses of client-side applications for both local and hybrid WBS architectures.
3.5.1 Comparison of Web Applications and Desktop Software

Compared to traditional desktop software, web applications provide a number of benefits for simulation modeling primarily related to improved accessibility, maintainability, and usability. As the diversity of Internet-enabled computing devices continues to expand, web-based applications have the great advantage of being accessible from virtually any modern device and platform including mobile devices, which are quickly rising in popularity and usage (Hales, 2013). In contrast, desktop software is often only compatible with a single platform or operating system as support for multiple platforms can significantly increase development costs by requiring multiple code bases for the same application.

Web applications are also better suited for collaboration between multiple users. While desktop software is often designed to be used on a single computer and by a single user, web applications are accessible to multiple users on different devices. For instance, if multiple users wish to access the same model using desktop software, they must all have a local installation of the same software as well as local copies of the same input data and model configuration files. Using a web application, multiple users do not need to install any specific software on their computer other than a standard web browser, and also have access to the same input data and model configurations stored on the server. If necessary, developers can provide user authorization capabilities to allow only certain users to make changes to the model while allowing all other users to view model results, which is a feature demonstrated by WIRM. The improved accessibility of web applications can thus better facilitate collaboration between multiple users as well as provide access to
users who might be unable to obtain a copy of the model software due to platform incompatibilities or licensing restrictions.

Web applications are also easier to maintain than desktop software in terms of fixing bugs and applying software updates. With desktop software, update patches must be applied by each user on their own local computer, while web-based applications can be instantaneously updated with a single deployment to the server, which is then immediately available to all users. When updating web applications, however, developers should be cautious and notify users in advance when making changes that may result in unexpected changes to model results. Support for multiple versions of a web application should be considered if major changes to the underlying algorithms are needed. A version control system would allow users to upgrade to a new version of the model at their convenience and ensure the results of their model do not change unexpectedly. But for bug fixes and other minor updates, web applications provide a more rapid and robust approach for routine software maintenance.

In terms of usability, web applications are executed in an environment that is familiar to most users who will already be accustomed to using web browsers. Desktop software, on the other hand, often requires considerable training as interfaces can vary widely between different applications even on the same operating system. Although training may also be required to properly use a web application, the fundamental application components including hyperlink-based navigation and common user interface controls will be familiar to the average web user. Despite this familiarity, however, it is the developer’s responsibility to ensure
that the model interface is intuitive and comprehensible for the intended users, a responsibility that applies as much to web applications as to traditional desktop software.

Although web applications provide a number of benefits over desktop software, there are also limitations that are important to consider. Perhaps the most commonly cited limitation of web applications is that they require a connection to the Internet. While this is still largely true, this limitation will soon apply primarily to server-side web applications where the application is controlled on the server. Client-side applications have increasing support for offline access through the new Offline Application Caching25 API (Kessin, 2012). Using this feature, client-side applications can be configured for offline access by instructing the browser to store a local copy of the files needed to run the application when the user first visits the application URL. Once the application files have been obtained, the user will be able to access the application through their standard browser even when a connection is not available.

In addition to offline access, client-side web applications can also have the ability to save and load data directly from the user’s local hard drive through the new Web Storage26 and File27 APIs (Kessin, 2012). The Web Storage API provides the ability to store data within the browser and was created as a replacement for cookie-based storage, which can pose security risks and is also severely constrained by the

amount of data that can be stored. This storage mechanism can be used to save input data, parameter values and model configurations locally within the browser, which can then be synchronized with a remote server at the user’s request or even automatically once a connection is available. The File API provides further data management possibilities through the ability to read and write local text files from within the browser, a task that traditionally depended on server-side file handling. A user could thus save the output data or model configuration to a text file, which can then be loaded back into the application in a manner similar to traditional desktop applications. Together, these new standard web APIs will allow developers to create client-side web applications that can even more closely resemble the functionality of desktop software by providing offline access as well as the ability to load and save data within the browser or through local files directly on the user’s computer without requiring access to the server.

Client-side web applications are also limited by the amount of computing power and memory available in a browser-based environment. While desktop applications can take full advantage of multiple-core processors and extensive hard drive storage space, web applications have historically not had complete access to the local processor or hard drive; although, this limitation is becoming less significant as browsers continue to evolve and provide increasing computational power for web applications. The exact limitations on client-side processing are neither clearly defined nor easy to determine as it depends on the hardware, operating system, and browser being used. Mobile devices, especially, provide limited computing power, although the evolution of mobile hardware is continuing at a rapid pace. Our
research suggests that client-side web applications are able to handle a much greater computational load than previously believed thanks to optimization of JavaScript engines in modern web browsers and the availability of FOSS libraries to support scientific computing.

As mentioned in Section 3.3.2, additional performance gains in client-side web applications could be investigated through the use of the new Web Workers API, which provides support for multi-threaded web applications in the browser. With Web Workers, the simulation engine could be placed into its own thread, freeing memory and computing resources for the user interface and output visualization. In addition, the upcoming WebGL API provides direct access to the local computer’s graphics processing unit (GPU), allowing a client application to execute computationally demanding numerical algorithms or render detailed 3D visualizations of model output with high performance. Much research is needed to evaluate the benefits and limitations of these new client-based APIs for performing more complex simulations using multithreaded or GPU-accelerated applications.

In addition to the computational limitations of running models within the browser, there are also practical limitations requiring that all client-side application code be written in JavaScript. Although JavaScript is often thought of as a web programming language, it is a general purpose language that is fully capable of performing numerical algorithms such as solving ordinary differential equations as demonstrated with WIRM. As discussed in Section 3.3.2, the optimization of JavaScript engines within modern web browsers enables high speed execution of common numerical algorithms and there is no inherent limitation in the types of
algorithms that could be implemented, as would be true for any other general purpose programming language. Although there are relatively few numerical libraries currently available for scientific computing in JavaScript, we expect to see more libraries become available over time as developers create new toolkits for supporting scientific web applications.

In summary, web applications provide a number of benefits compared to traditional desktop software by improving the accessibility, maintainability and usability of simulation models. While there are also a number of limitations to web applications, the rapid evolution of web technologies and standards continue to provide a variety of features and capabilities traditionally found only in desktop software. Considerable research is needed to investigate alternative methods for accessing local and remote data, performing increasingly complex simulations, designing intuitive and user-friendly interfaces, and providing offline access to client-side web applications.

3.5.2 Potential Use Cases for Client-side Web Applications
Client-side web applications provide a variety of new and exciting capabilities for web-based simulation models. The fundamental characteristic of client-side web applications is the transfer of computational load from the server to the client, which can allow for a more interactive and dynamic user interface in addition to reduced demand on the server. This general approach of client-side processing and visualization could be used in conjunction with either the local or hybrid WBS architectures for creating interactive user interfaces to simulation models, whether the models themselves are executed in the browser or on the server.
Using the local WBS architecture, client-side web application could be used to create rapid screening models similar to WIRM (as discussed in the beginning of Section 3.5), as well as web-based documentation for more complex models and educational modules for understanding fundamental modeling concepts and theories. Traditionally, model documentation has been composed of narrative descriptions, equations, and static diagrams, which can exceed hundreds of pages for large models. However, understanding how the components fit together and how the parameters affect various processes can be a difficult task, even for experienced modelers. Web-based documentation using small, interactive visualizations representing the underlying equations of individual model components could help users understand how a model works, and how the components relate to one another. Interactive visualizations that show the effects of changing parameter values on model output in real-time could be a more effective learning platform than the traditional approaches to model documentation (Liu and Stasko, 2010). While more research is clearly needed regarding the benefits of interactive visualizations for improving model understanding, client-side web applications using local WBS architectures provide the tools needed to develop and test this new approach to model education and documentation.

Client-side web applications are also not strictly limited to the local WBS architecture and could be used to create improved user interfaces for visualizing and analyzing the results of models that are executed on the server through the hybrid WBS architecture. Once a simulation is completed on the server, the client application could be used to retrieve the output data and perform a variety of
analysis or post-processing tasks such as filtering, aggregating, and visualizing the model output. Booth et al. (2011), for example, created an interactive, map-based client-side application to visualize the results of the SPARROW nutrient loading model, which is executed on the server.

A client-side web application could also be used to inspect model calibration by computing goodness of fit statistics between the model output and observed data. The interface could allow the user to analyze model performance under various conditions such as low (or high) flow conditions or specific seasons by providing tools for interactive data filtering and aggregation. Similarly, a client-side application could be used to compare the results for alternative scenarios generated by a server-side model. The client-side approach to post-processing model results would not only improve the interactivity of the application by generating visualizations in the browser, but also relieve the computation load on the server by performing these computational tasks locally. The flexibility and interactive features afforded by client-side applications could thus improve model transparency and access as decision makers and stakeholders would not have to rely on modeling experts to generate customized visualizations or summaries of model results.

### 3.6 Conclusions

Web-based simulation has been a topic of great interest throughout the environmental modeling community for many years. Only recently, however, have the technologies and tools emerged to develop client-side web applications that provide an interactive and easy-to-use interface accessible to both technical and
A client-side web application, the Web-based Interactive River Model (WIRM), was developed to demonstrate modern web development techniques using free and open-source software libraries and frameworks built using current web standards. WIRM also demonstrates the capability of client-side web applications to support the local WBS architecture where the browser performs both the numeric computation and visualization, tasks that traditionally required server-side approaches or browser plug-ins such as Java applets or Adobe Flash. Future research will focus on adding additional features to WIRM such as methods to save and load model configuration files, write output to local text files, integrate observation data to support model calibration, and provide uncertainty estimates to better facilitate the decision making process.

Although the client-side and local WBS approach may be currently limited to simple screening models such as WIRM or interactive model documentation, the evolution of web standards and development techniques is continuing at a rapid pace and will continue to provide increasing support for more complex models. Furthermore, new and upcoming client-side technologies will provide a number of features traditionally found only in desktop software such as offline access, local data storage and file access, and support for multi-threaded or GPU-accelerated computations. A significant amount of research is needed to discover innovative ways of using these new technologies in client-side web applications, and to create improved and accessible user interfaces for inspecting and analyzing web-based simulation results. With continuing advances in web technologies, we expect client-side web applications will someday play a major role in improving model
accessibility and usability for decision makers and stakeholders who often lack direct access to the models commonly used in environmental management and decision making.

3.7 Appendix A: Mathematical Theory

The mathematical model in WIRM is a nonlinear version of the Streeter-Phelps model for simulating the effect of biochemical oxygen demand (BOD) discharge on in-stream dissolved oxygen (DO) concentrations (Streeter and Phelps, 1925; Chapra, 1997). The model is represented by a system of two ordinary differential equations (ODEs):

\[
\frac{dL}{dt} = -F_{ox} k_d L \
\frac{do}{dt} = -F_{ox} k_d L + k_a (o_s - o)
\]

where \( L \) is the BOD concentration (mg/L), \( o \) is the DO concentration (mg/L), \( t \) is the downstream travel time downstream (d), \( F_{ox} \) is the DO inhibition factor (unitless), \( k_d \) is the first-order BOD decay rate (1/d), \( k_a \) is the DO reaeration rate (1/d), and \( o_s \) is the DO saturation concentration (mg/L).

Inclusion of the DO inhibition factor, \( F_{ox} \), is a modification to the original Streeter-Phelps model to limit the rate of BOD decay when DO concentrations are low. The inhibition factor is defined according to Michaelis-Menton kinetics:

\[
F_{ox} = \frac{o}{k_{so} + o}
\]
where $k_{so}$ is the DO inhibition half-saturation concentration (mg/L) such that when $o = k_{so}$ the inhibition factor is $F_{ox} = 0.5$ and the rate of BOD decay is reduced by 50%.

The BOD decay rate, $k_d$, is computed as a function of temperature using the standard Arrhenius-based temperature correction factor:

$$k_d = k_{d,20} \theta_{BOD}^{T-20}$$  \hspace{1cm} (3-4)

where $k_{d,20}$ is the standard BOD decay at 20°C (1/day), $T$ is the water temperature (°C), and $\theta_{BOD}$ is the temperature correction factor (unitless), typically 1.047 (Chapra, 1997).

Similarly, the reaeration rate, $k_a$, is corrected for temperature by:

$$k_a = k_{a,20} \theta_{DO}^{T-20}$$  \hspace{1cm} (3-5)

where $k_{a,20}$ is the DO reaeration rate at 20°C (1/day), and $\theta_{DO}$ is the temperature correction factor (unitless), typically 1.024 (Chapra, 1997). The DO reaeration rate at 20°C is estimated using the Covar chart, which selects among three empirical formulas of DO reaeration based on mean stream depth and velocity (Covar, 1976; Chapra, 1997). The oxygen saturation concentration, $o_s$, is computed from the water temperature using the saturation equation in Chapra (1997) assuming no salinity and that the system is at standard atmospheric pressure.

The system of equations can be solved numerically given the initial conditions:
\[ L(t = 0) = L_0 \]  
\[ o(t = 0) = o_0 \]

where \( L_0 \) and \( o_0 \) are the initial in-stream concentrations of BOD and DO, respectively, at the point of discharge \( (t = 0) \). The system of equations is solved using the 4th-order Runge-Kutta numerical method with a step size of 0.1 days to compute the concentrations of BOD and DO as a function of downstream travel time (Chapra, 1997).

Finally, in order to plot the concentrations spatially as a function of downstream distance, which is a more useful metric for interpreting the results, the travel time is converted to distance using the stream velocity:

\[ x = 86.4Ut \]

where \( x \) is the distance downstream (km), \( U \) is the velocity (m/s), and 86.4 is a unit conversion factor.
Chapter 4

Interactive Visualizations of Model Theory

This chapter demonstrates the use of interactive visualizations to better understand model theory and behavior. The chapter begins with the motivation for this research—the need to improve how models are explained, communicated, and understood. Next, research from the field of cognitive science is presented as a basis for understanding how and why interactive visualizations are powerful tools for improving model communication and understanding. A novel visualization called the mass balance diagram is then introduced and demonstrated using a simple hydrologic model called the abcd water balance model. This visualization provides a concise and intuitive representation of the underlying equations representing the individual water balance storage terms and flows. Finally, a web-based interface containing interactive versions of this visualization is presented. Through this interface, one can develop a better understanding of how the model works by interactively exploring the effect of each parameter on the various mass balance terms. The goal of this chapter is to illustrate the potential for the web to serve as a platform for interactive model documentation that improves communication and understanding.

4.1 Introduction

Mathematical models are commonly used to study and manage environmental and water resources systems. Models represent scientific theories using mathematical
notation in the form of equations. These theories and equations represent processes and interactions occurring in natural systems. Mathematical models are often used to confirm or reject theories by comparing model predictions to observation data. If the theory is confirmed, then the model can be used to make future predictions or perform numerical experiments for comparing alternative boundary conditions or management strategies.

The use of equations has been central to the advancement of mathematical, scientific, and engineering theory. Equations provide a compact and precise means of describing theories. They allow researchers to communicate more efficiently and unambiguously than through natural language. Equations can also be used to combine multiple theories, with each theory and equation representing an individual process. New equations, and thus new theories, can be derived by integrating or analyzing existing equations. Analytical techniques for manipulating equations provide insight into the behavior of a system, such as evaluating the limits or steady-state conditions of a system. However, despite the utility of mathematics for formalizing and developing theory, equations are not always the most effective means of explaining and communicating the theory itself.

In environmental and water resources engineering, model communication is a major challenge (Friedman et al., 1984; NRC, 2007). Models are widely used to support management and decision making. For models to be used and interpreted correctly, managers and decision makers need to understand the underlying theory—what the model represents, how it behaves, and what its assumptions and limitations are. Scientists and engineers often equate understanding model theory
with understanding the equations representing that theory. However, equations themselves are only one representation of theory. Other representations of theory can provide similar, if not greater, insight compared to equations.

Visualization is another common approach to communicate model theory. Visualizations represent information in the form of visual elements such as lines, circles, polygons, etc. Textbooks, journal articles, and model documentation often include visualizations to help the reader understand how models work and what they represent. Stock-and-flow diagrams show the individual components of a model and how those components are related. Plots of model equations show the shapes and characteristics of the equations representing relationships between variables and parameters. However, visualizations are often limited by the static nature of paper and ink. Traditional model documentation (i.e. articles, manuals, reports, and other types of documents) limits the amount of information that can be conveyed through these static visualizations. Reducing complex, multi-dimensional relationships down to a two-dimensional canvas can be a challenge. Multiple visualizations are sometimes used to show how model output changes as inputs or parameters change. But this approach cannot reflect all possible parameter combinations and thus the overall behavior of the model theory.

Interactive visualizations provide a powerful method for allowing users to explore the behavior of equations and theory at their own pace. By changing boundary conditions or parameter values, a single visualization can dynamically reflect how those changes affect the model computations. Interactive visualizations can thus provide more information than static visualizations because they are not limited to
a pre-defined set of conditions. Furthermore, they can provide additional insight into model behavior by showing the dynamics of the model through animation.

But are interactive visualizations really useful for understanding model theory? What does it mean to have an intuition about model behavior? To better understand the potential benefits of interactive visualizations, it is useful to consider the cognitive aspects of understanding model theory. The next section addresses these questions by drawing on theories and research from cognitive science and related fields. These theories provide a framework for discussing the trade-offs between alternative representations of model theory, and for supporting the use of interactive visualizations to better understand models.

4.2 Understanding Model Theory

The proper use of a model requires understanding the model theory, which is often equated with understanding the model equations. But equations are simply a representation of theory in the form of abstract mathematical symbols. Expressed with different symbols, two equations can look very different but reflect the same underlying theory and generate the same model output. This is not uncommon in the literature where different authors may describe the same model using different symbology (e.g. compare Martinez and Gupta (2010) and Sankarasubramanian and Vogel (2002)). Therefore, it is not the equations themselves but the theory and behavior represented by those equations that is important to understand.

When developing or using a model, an experienced modeler will often be able to predict, at least to some degree, the effect of changing a parameter on the model
output. But typically, the modeler is not actually solving the equations in their head. Rather, they have an intuition about how the model behaves and how the parameters and inputs affect that behavior. Although this intuition can be limited, especially for highly non-linear models, it nevertheless represents intrinsic understanding of model behavior.

As an example, consider the following equation:

\[
\frac{dY}{dt} = -kY \quad \text{where} \quad k > 0 \quad \text{and} \quad Y(t = 0) = Y_0 > 0
\]  

(4-1)

Engineers will likely recognize this equation as an ordinary differential equation describing first-order decay. This equation represents a theory stating that the rate of change of some quantity \( Y \) over time is linearly proportional to the value of that quantity, and that the initial value of \( Y \) is \( Y_0 \). This equation is commonly used to represent theories for various processes such as radioactive decay, chemical kinetics, and reservoir releases. The solution to Equation (4-1) can be derived analytically using simple integration to yield another equation, which again represents the same underlying theory:

\[
Y(t) = Y_0 e^{-kt}
\]  

(4-2)

Having studied these equations in various contexts, engineers instinctively know the shape and behavior of Equation (4-2). Even without specific values for \( Y_0 \) and \( k \), the general shape of the solution can be visualized as a mental image that might look similar to Figure 4-1. Although the mental image may not be as accurate or precise as the exact solution shown in Figure 4-1, it can capture the general shape
and characteristics of the equation. The value of $Y$ begins at $Y_0$ with a steep decreasing initial slope. As $t$ increases, the curve levels off and asymptotically approaches a value of $Y = 0$.

![Figure 4-1: Solution to First-order Decay Equation](image)

In addition to representing the general shape of the solution, this mental image can also be used to predict the effect of changing $k$ or $Y_0$. Increasing the rate parameter ($k$) results in a steeper curve that declines more quickly, but still approaches $Y = 0$. Decreasing the initial value ($Y_0$) causes the starting point of the curve to shift downwards, but at the same time the tail of the curve continues to approach the horizontal axis. The general behavior of this model can thus be understood by manipulating the mental image.

The ability to perform these mental manipulations is almost effortless, or at least far less difficult than mentally solving the equations. This observation suggests that the mental and visual representation of theory is a source of intuition about the
model behavior. With this intuition, the effects of changing parameters or inputs on the system behavior can be predicted without having to compute the solution or analyze the equations. Although a visual representation of theory was used in this example, there are other non-visual representations or sources of intuition as well.

To better understand model intuition and understanding, the field of cognitive science provides some theories and empirical evidence about the potential mechanisms involved in understanding model behavior. Although there are no clear or definitive answers explaining how humans think, some generally accepted theories provide a useful framework for discussing the topic. This discussion will focus on two of these theories: mental models for explaining how humans understand the representation and behavior of a model and cognitive load theory for explaining what makes some things harder or easier to learn than others. These two theories are described in the following sections and then used to explain why interactive visualizations can help us understand model theory.

### 4.2.1 Mental Models

The original concept of a mental model is often attributed to Craik (1943) who suggested that people possess an inner representation of the world around them (Johnson-Laird, 1980; Jones et al., 2011). This inner representation can be used to evaluate present situations and anticipate future situations based on past experiences. Johnson-Laird (1983a) later used this idea of a mental model as a basis for studying human reasoning and cognition in the field of psychology. Since then, mental models have been used to explain cognitive processes in a number of other fields including decision science (Courtney, 2001), system dynamics (Doyle and
Ford, 1998), science education (Rapp, 2005), and human-computer interaction (Norman, 1986) to name just a few.

Rouse and Morris (1986) reviewed the definitions and applications of mental models in different fields and identified three common themes regarding their purpose: describing, explaining, and predicting systems. Based on these themes, Rouse and Morris (1986, p. 351) define mental models as “the mechanisms whereby humans are able to generate descriptions of system purpose and form, explanations of system functioning and observed system states, and predictions of future system states.” By this definition, mental models can provide a basis for explaining how people develop an understanding of mathematical theory. In order to describe a model, explain how it works, and predict how it will respond to changes in boundary conditions or parameters, the user must possess a mental model of the theory.

Mental models can be formed in a variety of ways, including through instruction or direct experience. Norman (1983) argues that one of the primary ways mental models are formed is through direct interaction with a system. For example, learning to drive a car requires sitting in the driver’s seat and interacting with the controls, rather than simply reading the instruction manual. Norman uses this idea as a basis for his principles of user interface design in the field of human-computer interaction. In order to effectively utilize software, users must have some understanding of how the software will behave—they must have a mental model of the software. The design of software interfaces should then focus on ensuring the development of an accurate mental model.
In scientific research, Nowak et al. (2013) argue that the primary benefit of developing computer simulations is to develop a mental model of the system being simulated. Through the process of writing programming code and interacting with the simulation, researchers can develop a mental model of how the system behaves. This mental model then allows them to predict how the model will change in response to varying inputs or parameters. The end result is a greater intuition and understanding about the model and how it represents the system. This may explain why some environmental modeling software is often hard to use. Because the model user is not involved in the development of the model or software, they do not have the same mental model as the model developer of how the software works.

A central feature of mental models is that they can be used to perform mental simulations (Liu and Stasko, 2010; Johnson-Laird and Khemlani, 2013). Mental simulations are used to compare potential outcomes for different actions or conditions. This process is believed to be a foundation for human reasoning, problem solving, decision making, and other forms of thinking. The creation and simulation of mental models is sometimes called model-based reasoning (Nersessian, 2002).

Although mental models may play a central role in human thinking, they also have well-known limitations. Mental models are often inaccurate, incomplete, and biased by previous experiences (Jones et al., 2011). Mental models are influenced by a person’s preconceived ideas and perspectives. This may explain confirmation bias, where a person only accepts new information if it conforms with their existing mental model (Jones et al., 2011). For example, the differing opinions on climate
change can be attributed to different mental models (Sterman, 2008). Because of these subjective aspects of mental models, one person’s mental model of a system often differs from that of another person based on their unique experiences. These limitations of mental models may lead to conflicts when different people possess different mental models of a system and thus believe in different behaviors of the system.

Kolkman et al. (2005) propose a methodology for addressing water resources conflicts based on analyzing the mental models of the participants. They argue that conflicts arise from differences between the mental models of the individuals. They suggest that the primary role of scientific experts is to help refine those mental models using simulation models. Participants can then make decisions based on the same understanding of the system behavior through a so-called shared mental model. Jones et al. (2011) describe a similar process of analyzing mental models to improve natural resources management.

Although mental models seem similar to conceptual models, there are important differences (Greca and Moreira, 2000). Mental models are internal representations of a system or theory, while conceptual models are external representations. In other words, mental models exist in the mind of an individual, while conceptual models can be written down and shared between individuals. One might assume that by learning from a conceptual model, one would form a mental model that would be consistent with the conceptual model. However, there is evidence to suggest this is not the case and that mental models often differ from conceptual
models due to the individuals unique experiences and previous knowledge (Greca and Moreira, 2000).

In summary, mental models provide a useful construct for understanding how humans reason and understand the world. Mental models are an internal representation of a system that can be mentally simulated to predict behavior and anticipate the outcomes of alternative actions. For mathematical modeling, mental models appear to play an important role in understanding model theory. Clearly, we do not solve or manipulate equations in our mind with ease, and yet we can gain an intuition about the behavior of these equations. This intuition may be the result of a mental simulation, which suggests we possess some form of mental model. Therefore, improving our understanding of model theory requires facilitating the formation of mental models representing that theory. To understand how mental models are formed, it is useful to consider another theory called cognitive load theory that focuses on the learning process and explains why some things are easier to learn than others.

4.2.2 Cognitive Load Theory
Cognitive load theory provides a basis for understanding how humans learn (Sweller et al., 1998). This theory is based on the existence of two types of memory: working memory and long-term memory. Working memory is the part of the brain that actively processes information, what is commonly called consciousness. After the information is processed in working memory, it is stored in long-term memory in the form of schemata. Schemata are mechanisms of storing and organizing information (i.e. knowledge). The purpose of cognitive load theory is to understand
how information is processed by working memory and then stored as schemata in long-term memory, and what factors affect this process.

Although schemata are thought to be related to mental models, the distinction is unclear and unresolved (Jones et al., 2011). One theory is that mental models use schemata as a source of information, which is then processed in combination with new information to perform mental simulations (Merrill, 2000). Although a mental model may be based on information stored in long-term memory, the simulation of a mental model is believed to occur in working memory (Nersessian, 2002).

The central premise of cognitive load theory is that working memory is limited (Sweller et al., 1998). When there is too much information to process, working memory becomes overloaded and the brain is unable to store the new information as schemata. The amount of information being processed by working memory is called the cognitive load. Cognitive load is primarily associated with new information and is not affected by the use of existing schemata in long-term memory. If the learner is reading about a familiar topic, then they can use existing schemata to process new concepts. But if the subject is completely unfamiliar, then there are more elements of new information to process in working memory resulting in a high cognitive load.

There are three types of cognitive load: intrinsic, extraneous, and germane (Sweller et al., 1998). The intrinsic cognitive load is the complexity of the information itself regardless of the form in which it is presented. Intrinsic cognitive load is considered immutable, meaning it cannot be changed. It is generally related to the number of
individual elements of information and to what degree those elements interact. Extraneous cognitive load is the additional load associated with the form of the information (i.e. how it is presented). When information is presented in a clear and easy to understand way, it has a lower extraneous cognitive load. The germane load is the processing required to store new information as schemata in long-term memory. Together, these three types of cognitive load are useful for studying the effectiveness of different forms of learning materials.

Because the intrinsic cognitive load is immutable, the design of learning materials focuses often on minimizing the extraneous cognitive load (Sweller et al., 1998). The subject matter cannot be made any more or less complicated, only its presentation can be changed. Information should thus be presented as clearly as possible and in such a way that the learner can store that information in their long-term memory. When the presentation of information is too complex, it imposes a high cognitive load that hinders the learning process.

In applications of environmental modeling, cognitive load theory may explain why equations are often harder to understand than visualizations, even when both representations are based on the same underlying theory. Equations may have a high extraneous load because they contain many individual and interrelated elements. To understand a new equation (or set of equations), the learner must keep track of the associations between different abstract symbols and the physical processes they represent. If the learner is already familiar with the equations and symbols, then they can utilize existing schemata and do not need to store these associations in working memory. As an example, the symbol ‘Q’ is often used to
represent the volumetric flow rate for a stream (i.e. streamflow). For water resources engineers, this association between ‘Q’ and streamflow may be stored as a schema and thus does not impose a cognitive load on working memory. But for someone unfamiliar with this symbology, the association between ‘Q’ and streamflow must be stored in working memory, which adds to their cognitive load.

In addition to the symbols used to represent variables and parameters, there are symbols representing mathematical operations that may contribute to cognitive load. Through training and experience, engineers become familiar with common operator symbols (e.g. the integral or derivative symbols), as well as the behavior of common equation forms (e.g. the exponential, $e^{-kt}$). Existing schemata representing these symbols and operations allow an engineer to more easily process new equations in their working memory. But other audiences may not have existing schemata about equations and thus processing all of this information imposes a higher cognitive load.

Visualizations can facilitate learning by imposing lower cognitive loads on working memory. One of the primary mechanisms by which visualizations reduce cognitive load is through the spatial integration of information (Sweller et al., 1998). For example, by placing text labels directly beside visual elements, there is less cognitive load associated with making the connection between the visual elements and the things they represent. For example, Larkin and Simon (1987) found evidence that diagrams of physics problems improved learning by grouping related information together spatially. This spatial proximity of related information reduces
the need to search for or remember associations between symbols and their meaning.

Visualizations also reduce cognitive load by shifting some of the demands to the visual perception system, which effectively expands working memory (Lohse, 1997; Card et al., 1999). Visual perception is very efficient and effective at identifying patterns (Few, 2012). For example, it is often more difficult to find patterns in a table of numbers than it is through a visual representation of those numbers. For model theory, a visualization of an unfamiliar equation clearly illustrates its shape and characteristics, which results in less cognitive load than if the equation had to be mentally processed in its symbolic form.

However, visualizations can also be detrimental to learning. Poorly designed visualizations impose a high cognitive load and make it difficult to understand the information being presented (Huang et al., 2009). Animated visualizations can be overwhelming when many independent elements are simultaneously changing, making it difficult to focus on any one element or understand how the entire system changes as a whole (Lowe, 2014). However, by taking the cognitive effects into account and using principles and best practices for effective design (e.g. Tufte (1983) and Few (2012)), visualizations can be designed to minimize cognitive load and thus enhance cognitive functioning.

4.2.3 Interactive Visualizations

Although data visualization and human-computer interaction have been studied for many years, only recently have researchers begun investigating the dynamics between human cognition and interactive visualizations (Yi et al., 2007; Liu and
Traditionally, visualization research has focused on the external representation of information, that is developing different kinds of visualizations (Liu and Stasko, 2010). However, researchers are now recognizing the importance of understanding the dynamics between these external representations and the internal representations associated with cognition.

The combination of cognitive science, data visualization, and human-computer interaction is the focus of information visualization (InfoVis), which is defined as “the use of computer-supported, interactive, visual representations of abstract data to amplify cognition” (Card et al., 1999, p. 7). Visual analytics is another relatively new field focusing on combining information visualization with powerful data analysis techniques to develop insight from large datasets (Thomas and Cook, 2005; Keim et al., 2008). Pike et al. (2009) describe some key areas of research needed to better understand the relationships between interactive visualizations and human cognition, which they call the science of interaction. However, despite the need for more research in this area, there are a few common theories describing how interactive visualizations improve our understanding of data and information. Two of the primary theories are that interactive visualizations 1) facilitate mental model formation, and 2) amplify cognition by expanding working memory (Liu and Stasko, 2010).

Unlike static visualizations, interactive visualizations allow the user to directly manipulate the visualization and explore the problem space. The process of interaction is believed to be one of the primary mechanisms of mental model formation (Norman, 1983). By forming a mental model, the user can later perform
mental simulations based on that model. Expert modelers may be able to predict and understand how parameters and inputs affect model behavior because they possess a mental model of the theory with which they can perform mental simulations. Whether the resulting mental model itself is a visual representation is debatable. The form of mental models can be visual, linguistic, abstract, or even some combination of the three (Johnson-Laird, 1998). Nevertheless, the process of interaction can stimulate the formation of an internal representation in whatever form can be used for mental simulation.

In addition to facilitating mental model formation, interactive visualizations are also believed to amplify cognition (Card et al., 1999). Sometimes referred to as cognitive offloading, interactive visualizations reduce the cognitive load on working memory by utilizing the visual perception system. In other words, instead of thinking about what shape an equation will take, one can simply look at it. Some consider this an extension of cognition. As Liu and Stasko (2010) describe: “Interaction is not merely a delegate or executor of the cognitive processes in the head, it is a central part of the cognitive process.” For learning model theory, interactive visualizations allow the learner to simply look at the shape and relationships between equations, instead of keeping track of those equations (and variables) in working memory.

Another related theory is distributed cognition in which interactive visualizations do not simply amplify cognition, but actually change the set of cognitive skills being used (Liu et al., 2008). For example, when thinking about the effect of changing a model parameter, one might think about the analytical properties of the
equation. But with an interactive visualization, the cognitive processes focus on pattern recognition and identifying how the shape of the equation changes. Although both forms of thinking focus on the same end result, they are very different kinds of thinking.

In summary, there are a number of different theories explaining why interactive visualizations are useful for understanding complex systems and information. These theories tend to focus on the formation of mental models for performing mental simulations, and the amplification or extension of cognitive functioning by reducing cognitive load. However, more research is clearly needed to obtain empirical evidence that supports these theories. Nevertheless, the use of interactive visualizations for enhancing understanding and cognition is worth exploring to address the challenges of model understanding and communication in environmental and water resources engineering. The remainder of this chapter demonstrates the use of interactive visualizations for understanding the theory of a simple water balance model.

4.3  abcd Water Balance Model

The abcd water balance model is a watershed model used to compute streamflow based on climatic inputs of precipitation and potential evapotranspiration. It was originally introduced by Thomas (1981) as a simple approach for computing water balances in support of the U.S. National Water Assessment. The abcd model is a parsimonious model having minimal data requirements and as few parameters as possible while still being able to reproduce different hydrologic behaviors. The
model is named for its four primary parameters (called $a$, $b$, $c$, and $d$, described in the following section).

Over the past thirty years, the $abcd$ model has been used for a variety of watershed hydrology studies. It has been compared to other parsimonious water balance models such as the Thornthwaite-Mather and Palmer models, and found to perform as well under a variety of conditions (Alley, 1984; Vandewiele and Xu, 1992). It has been used as part of prediction systems for forecasting streamflow based on precipitation (Alley, 1985; Block et al., 2009). Fernandez et al. (2000) developed an approach for regional calibration of the $abcd$ model using multiple observation stations in an effort to improve estimates of streamflows in ungauged basins. Sankarasubramanian and Vogel (2002) used the model to study the empirical relationships between actual evapotranspiration, potential transpiration, and precipitation. (Vogel and Sankarasubramanian, 2003) show that the $abcd$ model often reproduces watershed dynamics better at a monthly time-scale compared to an annual time-scale. Martinez and Gupta (2010) evaluated the general performance of the $abcd$ model in representing 764 catchments across the United States. They concluded that it is only reliable under certain conditions and more work is needed to identify appropriate model structures. Finally, Liu et al. (2013) recently used the $abcd$ model to evaluate the effect of human activities and climate variability on streamflow in the Yiluo River, China.

For this research, the $abcd$ model is used to demonstrate a new approach to explaining model theory using interactive visualizations. This model was chosen
for its simplicity and because it incorporates fundamental hydrologic processes common to most watersheds.

4.4 Model Equations

The \textit{abcd} model is comprised of two storage compartments: soil moisture and groundwater (Figure 4-2). These compartments are subject to gains and losses through precipitation, evapotranspiration, direct surface runoff, groundwater recharge, and groundwater discharge. The total streamflow is the sum of direct surface runoff and groundwater discharge. All flows and storages are represented in units of depth (e.g. millimeters or inches). These depths can be converted to water volumes by multiplying by the drainage area of the watershed. The model can be applied on various time scales, but is most often run with a monthly time step and generally should not be used at any greater time step (see Vogel and Sankarasubramanian (2003)).
For the soil moisture compartment, the fluxes and storage are computed using a quantity called the available water ($W_t$). For time step $t$, the amount of available water is the sum of soil moisture from the previous time step ($S_{t-1}$), and the amount of precipitation during the current time step ($P_t$).

$$W_t = S_{t-1} + P_t \quad (4-3)$$

The available water is partitioned between the amount of soil moisture at the end of time step $t$ ($S_t$) and the amounts of water lost to evapotranspiration ($E_t$), groundwater recharge ($GR_t$) and direct runoff ($DR_t$) over time step $t$.

$$W_t = S_t + E_t + GR_t + DR_t \quad (4-4)$$
These terms are computed using a quantity called the evapotranspiration opportunity \( Y_t \), which is the sum of the remaining soil moisture and the amount of actual evapotranspiration.

\[
Y_t = E_t + S_t \tag{4-5}
\]

The evapotranspiration opportunity is computed as a non-linear function of available water \( W_t \) and two parameters \( a \) and \( b \):

\[
Y_t = \frac{W_t + b}{2a} - \sqrt{\left(\frac{W_t + b}{2a}\right)^2 - \frac{W_t b}{a}} \tag{4-6}
\]

The change in soil moisture over each time step is assumed to be linearly proportional to the rate of potential evapotranspiration \( PET_t \):

\[
\frac{dS}{dt} = -\frac{PET_t}{b} S \tag{4-7}
\]

Assuming the amount of soil moisture at the start of time step \( t \) is equal to the amount of evapotranspiration potential \( Y_t \), Equation (4-7) can be solved for the amount of soil moisture at the end of time step \( t \):

\[
S_t = Y_t e^{-\frac{PET_t}{b}} \tag{4-8}
\]

The amount of actual evapotranspiration over the current time step \( E_t \) is then computed from Equations (4-5) and (4-8):
\[ E_t = Y_t - S_t \]
\[ = Y_t - Y_t e^{-\frac{P_{ET_t}}{b}} \]
\[ = Y_t \left(1 - e^{-\frac{P_{ET_t}}{b}} \right) \quad (4-9) \]

Finally, the remaining available water is allocated between groundwater recharge \((GR_t)\) and direct runoff \((DR_t)\) by parameter \(c\).

\[ GR_t = c(W_t - Y_t) \quad (4-10) \]
\[ DR_t = (1 - c)(W_t - Y_t) \quad (4-11) \]

For the groundwater compartment, the change in storage is computed as the net gain from the soil moisture storage as recharge \((GR_t)\) and loss from groundwater discharge \((GD_t)\).

\[ G_t - G_{t-1} = GR_t - GD_t \quad (4-12) \]

The amount of groundwater discharge is assumed to be linearly proportional to the amount of storage by parameter \(d\):

\[ GD_t = d \cdot G_t \quad (4-13) \]

Substituting Equation (4-13) into Equation (4-12), the groundwater storage at time \(t\) \((G_t)\) can then be computed by:

\[ G_t = \frac{G_{t-1} + GR_t}{1 + d} \quad (4-14) \]

Finally, the total streamflow \((Q_t)\) is computed as the sum of direct runoff and groundwater discharge:
These equations describe the theory of the \textit{abcd} model. However, the equations themselves provide little insight into the behavior of the model. Analytical techniques such as taking the limits or derivatives of these equations could yield some insight into the model behavior, but it is difficult to understand how these equations behave individually and also affect one another. Experienced modelers may be familiar with the forms of some of these equations and thus able to rely on existing knowledge (e.g. the exponential decay of soil moisture shown in Equations (4-7) and (4-8)). But the other equations such as Equation (4-6) describing the relationship between evapotranspiration opportunity and available water may be less familiar and less intuitive. Furthermore, the effects of changing the parameters on the overall behavior are not intuitive, especially for parameters $a$ and $b$ due to the non-linearity of Equation (4-6). As an alternative approach to explaining this model, a visualization called the mass balance diagram was developed to represent the model theory in a simple and easy-to-understand way.

\subsection*{4.5 Mass Balance Diagram}

Visualizations provide an alternative representation of model theory. As discussed in Section 4.2, visualizations facilitate learning and understanding by reducing cognitive load and enhancing cognitive processes through pattern recognition and the power of visual perception (Card et al., 1999). For the \textit{abcd} model, the entire model theory described in Section 4.4 can be represented by just two visualizations, one for each storage compartment. These visualizations are called mass balance diagrams and reflect the magnitude of each storage and flux term in the model for...
any given set of parameter values. In this section, the theoretical basis of the mass balance diagram is first described. Static versions of this diagram are then presented for the groundwater and soil moisture compartments of the abcd model. After describing the basis for this diagram and how it can be applied to the abcd model, web-based interactive versions of the diagram are presented in Section 4.6.

4.5.1 Generalized Mass Balance Diagram
The mass balance diagram is based on the continuity principle: the change in storage must be equal to the sum of the inflows minus the sum of the outflows.

\[
\Delta \text{Storage}_t = \text{Storage}_t - \text{Storage}_{t-1} = \sum \text{Inflows}_t - \sum \text{Outflows}_t \quad (4-16)
\]

Note that for the storage term, the subscripts \( t \) and \( t - 1 \) denote the amounts of storage at the end of time steps \( t \) and \( t - 1 \), respectively. For the flux terms, the subscript \( t \) denotes the sum of inflows and outflows over time step \( t \). This equation can be rearranged as

\[
\text{Storage}_t + \sum \text{Outflows}_t = \text{Storage}_{t-1} + \sum \text{Inflows}_t \quad (4-17)
\]

This alternative form states that the sum of the remaining storage at time step \( t \) and the total outflows must equal the sum of initial storage for that time step and the total inflows. This equation can be plotted with the left hand side (LHS) on the ordinate (y-axis) and the right hand side (RHS) on the abscissa (x-axis) (Figure 4-3). Because the LHS equals the RHS, this equation is simply represented by the 1:1 line of equality.
4.5.2 Groundwater Storage Compartment

The mass balance diagram for the groundwater storage compartment is based on the mass balance equation in Equation (4-12), which can be rearranged in the form of Equation (4-17):

\[ G_t + G_D_t = G_{t-1} + G_R_t \]  

(4-18)

This equation states that the sum of the remaining groundwater storage \( G_t \) and the groundwater discharge \( G_D_t \) is equal to the sum of the initial groundwater storage \( G_{t-1} \) and the amount of groundwater recharge \( G_R_t \) for time step \( t \).

Recall from Equation (4-14) that the remaining amount of storage \( G_t \) is computed as a function of \( G_{t-1} + GR_t \):

\[ G_t = \frac{G_{t-1} + GR_t}{1 + d} = \left( \frac{1}{1 + d} \right) \cdot (G_{t-1} + G_R_t) \]  

(4-19)
By substituting Equation (4-19) for $G_t$ in Equation (4-13), the groundwater discharge $GD_t$ can also be computed as a function of $G_{t-1} + GR_t$:

$$GD_t = d \cdot G_t = d \cdot \left( \frac{G_{t-1} - GR_t}{1 + d} \right) = \left( \frac{d}{1 + d} \right) \cdot (G_{t-1} + GR_t) \quad (4-20)$$

Therefore, both terms on the LHS of Equation (4-18) ($G_t$ and $GD_t$) can be computed as a function of the RHS of Equation (4-18) ($G_{t-1} + GR_t$). Figure 4-4 shows the relationship between each term on the LHS and the sum of the RHS for a parameter value of $d = 0.3$.

![Figure 4-4: Groundwater Storage and Discharge Components.](image)

The dotted line represents the 1:1 line of equality. Parameter $d=0.3$.

The two panels in Figure 4-4 can then be combined into a single figure as the stacked area chart shown in Figure 4-5. Because the sum of the LHS terms ($G_t + GD_t$) equals the sum of the RHS terms ($G_{t-1} + GR_t$) from the continuity principle of Equation (4-18), the top edge of the combined areas will always lie on the 1:1 line of equality. For any given value of ($G_{t-1} + GR_t$), the corresponding values of $G_t$ and $GD_t$ can be inferred from the heights of the two vectors shown in Figure 4-5.

125
Figure 4-5 is thus a single visual representation of the two components in Equations (4-19) and (4-20) arranged according to the mass balance in Equation (4-18).

![Groundwater Mass Balance Diagram](Image)

**Figure 4-5: Groundwater Mass Balance Diagram**

The effect of parameter $d$ on the groundwater model can be illustrated by comparing multiple versions of this diagram. Figure 4-6 shows the groundwater mass balance diagram for three values of parameter $d$.

![Effect of Changing Parameter d on Groundwater Mass Balance Diagram](Image)

**Figure 4-6: Effect of Changing Parameter d on Groundwater Mass Balance Diagram**
Figure 4-6 shows that by increasing the value of $d$, the amount of groundwater discharge increases and the remaining groundwater storage decreases for any given amount of available groundwater ($G_{t-1} + GR_t$).

Finally, the mass balance diagram can also be used to understand the algorithm for computing the model solution. Beginning at an initial point, one can follow the trajectory of the model over a sequence of time steps. Figure 4-7 shows the computation of groundwater storage over five time steps assuming no recharge ($GR_t = 0$), parameter $d = 0.3$, and an initial value of $G_{t=0} = G_0$.

![Figure 4-7: Example of Groundwater Algorithm](image)

The solution of the groundwater storage as a function time is represented by the heights of the vectors shown in Figure 4-7. By plotting these vector heights over time, the solution to the model can be represented as a time series. Figure 4-8 plots the solution for this example as the groundwater storage relative to the initial value.
The trajectory of the groundwater storage thus illustrates the exponential decay behavior one expects from a linear reservoir in the absence of any inflows.

![Figure 4-8: Time series of Groundwater Storage](image)

**4.5.3 Soil Moisture Compartment**

The soil moisture compartment of the *abcd* water balance model can also be represented in a mass balance diagram. By setting Equation (4-3) equal to Equation (4-4), the mass balance equation for the soil moisture component in the form of Equation (4-17) is:

\[
S_t + E_t + GR_t + DR_t = S_{t-1} + P_t
\]  

(4-21)

As with the groundwater compartment, the equations in Section 4.4 can be rearranged such that each term on the LHS of Equation (4-21) is a function of the RHS, which is also equal to the available water term \(W_t\) from Equation (4-3). Note that the evapotranspiration potential itself is a function of available water.
\( Y_t = \int (W_t) \) as defined by Equation (4-6). The individual terms in Equation (4-21) are represented by the following equations described in Section 4.4.

\[
S_t = Y_t(W_t) e^{-\frac{PET_t}{b}} \tag{4-22}
\]

\[
E_t = Y_t(W_t) \left(1 - e^{-\frac{PET_t}{b}}\right) \tag{4-23}
\]

\[
GR_t = c(W_t - Y_t(W_t)) \tag{4-24}
\]

\[
DR_t = (1 - c)(W_t - Y_t(W_t)) \tag{4-25}
\]

The sum of these four terms can be represented as a stacked area chart with the horizontal axis representing the available water, \( W_t \). Figure 4-9 shows the mass balance diagram for the soil moisture component with parameter values of \( a = 1, b = 5, c = 0.5, \) and \( PET_t = 1 \). Note that although potential evapotranspiration is an input variable to the model, it can be treated as a parameter for the purpose of using this diagram to understand the model theory.

![Figure 4-9: Soil Moisture Mass Balance Diagram](image-url)
Figure 4-9 thus represents the model theory for the soil moisture compartment as defined by Equations (4-21) - (4-25). From this diagram, one can see how each storage and flux term is computed for any given amount of available water. Furthermore, the assumptions and theory of the model can be identified. For example, with these parameters values, groundwater recharge and direct runoff only occur when there are at least 5 units of available water. Below this threshold, the available water is partitioned only between evapotranspiration and remaining soil moisture indicating that the soils are under-saturated. Above this threshold, the soil is saturated such that the amounts of soil moisture at the end of the time step and evapotranspiration remain constant; any excess water is routed to groundwater recharge and direct runoff. This threshold is defined by parameter $b$, which represents the saturation level of the soils.

Figure 4-10 shows an example calculation where the initial soil moisture is $S_{t-1} = 4$, and the precipitation over the current time step is $P_t = 2$. The total available water for the current time step is then $W_t = 6$. The 6 units of available water results in 0.5 units of groundwater recharge, 0.5 units of direct runoff, 0.9 units of evapotranspiration, and 4.1 units of soil moisture.
The effects of different model parameter values can be illustrated by comparing multiple versions of the diagram. Figure 4-11 shows the effect of changing parameter $a$. As $a$ decreases from 1.0, the direct runoff and groundwater discharge components extend below the saturation point, which has the effect of allowing direct runoff and groundwater recharge to occur when soils are under-saturated.

Parameter $a$ is often described as “the propensity of runoff to occur before the soil is fully saturated” (Thomas et al., 1983). However, without this diagram, the
interpretation of this description is somewhat vague. It is not clear from Thomas’ description, nor the equations themselves, exactly how the “propensity of runoff” is reflected in the mass balance. However, the diagram shows that as $a$ decreases, a greater fraction of the available water is routed to direct runoff. Furthermore, the diagram shows that decreasing $a$ also increases the “propensity” for recharge to occur, not just runoff. This example demonstrates the benefits of visualizing the mass balance to understand the model theory. This approach can thus provide insight into the model behavior that could not be attained by analyzing the equations or reading the parameter descriptions.

The effect of changing parameter $b$ is shown in Figure 4-12. The value of $b$ defines the maximum value of the evapotranspiration potential variable $Y_t = S_t + E_t$. As $b$ increases, greater amounts of available water are necessary to create runoff and recharge when $a = 1.0$.

![Figure 4-12: Effect of Changing Parameter $b$ on Soil Moisture Mass Balance](image)

The effect of changing parameter $c$ is shown in Figure 4-13. As $c$ increases, the amount of groundwater recharge also increases and the amount of direct runoff decreases. Parameter $c$ thus defines the ratio of direct runoff to groundwater
recharge when the available water exceeds the sum of the actual evapotranspiration and remaining soil moisture.

Figure 4-13: Effect of Changing Parameter $c$ on Soil Moisture Mass Balance

Finally, the effect of changing the potential evaporation rate, $PET_t$, is shown in Figure 4-14. As potential evapotranspiration increases, a greater portion of the evapotranspiration potential ($Y_t = S_t + E_t$) is routed to actual evapotranspiration and less water remains as soil moisture. This change, however, has no effect on the amounts of groundwater recharge or direct runoff.

Figure 4-14: Effect of Changing Potential Evapotranspiration on Soil Moisture Mass Balance

Although the examples shown in this section provide some insight into how the model behaves, they are limited to only a small subset of the possible combinations of parameter values. For example, the effects of changing $b$, $c$, and $PET$ are only

133
shown for $a = 1.0$. Furthermore, the transitions between parameter values in each figure are difficult to interpret (e.g. are the transitions linear or non-linear?). These are common limitations due to the static nature of traditional documents. In the next section, a web-based interactive version of this diagram is presented that provides a more powerful approach to explore the model theory.

4.6 Web-based Interactive Visualization

Web-based interactive visualizations of the $abcd$ mass balance diagrams were created using the methodology described in Section 2.4 of Chapter 2 (Figure 4-15 and Figure 4-16). Because this methodology is based on using only standard web languages (HTML, CSS, and JavaScript), these visualizations are compatible across platforms and browsers and do not require installing browser plug-ins. With these interfaces, users can explore different combinations of parameter values by moving the sliders. When a slider is moved, the equations are re-computed and the visualization immediately refreshes. This interaction allows the user to see how the diagram changes in real time as each parameter is increased or decreased.

Figure 4-15: Interactive Mass Balance Diagram for Soil Moisture
In addition to changing parameter values, these visualizations also allow the user to inspect the values of each component at any position along the horizontal axis. When the user moves the mouse over the diagram, vertical line segments are drawn corresponding to the x-coordinate of the mouse position (Figure 4-17). The heights of these line segments correspond to the value of each component similar to the vectors shown in Figure 4-5 and Figure 4-10. This interaction provides a mechanism for visually comparing the relative values of each variable.

Figure 4-16: Interactive Mass Balance Diagram for Groundwater

Figure 4-17: Hover Interaction of the Interactive Mass Balance Diagram
The interactive versions of these diagrams have a number of benefits over the static versions shown in Section 4.5. They allow the user to explore virtually unlimited parameter combinations. Although the ranges of the sliders are currently fixed, additional settings could be added allowing the user to customize the range of each parameter. Another benefit is the animation of the diagram that results from dragging the slider. This animation reflects the linear and non-linear relationships between the parameter values and the mass balance terms, which cannot be easily inferred from the static versions of these diagrams. The animation also reflects how the equations are related according to the continuity of the system. Because the upper edge of each diagram must lie on the 1:1 line, the decrease in one area must correspond to an increase in another area. This visual consistency reflects the continuity of the system and the relationships between the equations.

The mass balance diagram was designed to minimize cognitive load by utilizing principles of effective visual design (e.g. Tufte, 1983; Few, 2012). First, the labels for each visual element were placed directly beside the corresponding area rather than in a legend. This reduces the cognitive load associated with searching for and remembering the association between the visual elements and the processes they represent. Second, the number of colors was minimized by using different versions of each color (one bright and one faded). Finally, grid lines, borders, and other extraneous elements were excluded to allow the user to focus on the important aspects of the visualization. Together, these design principles were used to make the visualization as clear as possible and minimize cognitive load.
In addition to minimizing the cognitive load, these visualizations were designed to facilitate the formation of a visual mental model. By interacting with each visualization, the user can quickly develop the ability to mentally manipulate the diagram as a mental image. This ability to perform mental simulations provides an intuition and understanding of how changing parameter values affects the mass balance terms. This ability can then be utilized when applying and calibrating the model to a specific system, as will be discussed in Chapter 5.

4.7 Discussion and Conclusions

The interactive mass balance diagrams demonstrated in this chapter illustrate a powerful approach to explaining model theory. Although equations are useful and necessary for developing model theory, they often do not provide sufficient insight into how the model works, especially among non-technical audiences and inexperienced modelers. Interactive visualizations provide an alternative representation of model theory that has lower cognitive load and can be designed to facilitate mental model formation. With a mental model based on these visualizations, the user can perform mental simulations of the model theory and predict how changing parameter values affects the model theory.

Although no user studies were performed in this research, it is clear that the geometric shapes and patterns reflected in these visualizations are easier to process and remember than the equations themselves. By exploring the parameter space using the interactive visualizations, users can more easily develop an intuition about how the parameter values change the model behavior. For example, the non-linear effect of changing parameter $a$ on the model behavior is much easier to infer from
Figure 4-11 than from Equation (4-6). Furthermore, it is clear from this figure that changing parameter $a$ affects all four components of the soil moisture compartment even though it only appears in a single equation. This knowledge is then useful when applying and calibrating a model for a specific system. By having this simple and concise representation available in their mind, users can focus attention on other inquiries such as understanding the relationship between the model output and the observation data.

Although the mass balance diagram is based on the principle of continuity, which applies to environmental models in general, it requires that the model use a specific algorithm for computing the solution. Specifically, the remaining storage and outflows of a model compartment be computed from the sum of the initial storage and total inflows. Although the $abcd$ model uses this type of solution algorithm, other models may involve bidirectional flows or inflows that depend on the amount of available storage and thus cannot be represented using this diagram. However, other kinds of visualizations based on continuity or other general principles could be developed to explain the structure of models with other characteristics and algorithms.

Another important point is that these visualizations were designed to understand the model theory and not the model output. Clearly, using this diagram to compute a time series solution would be difficult. These visualizations are instead intended to understand how the output is computed. In other words, they allow the user to see inside the black box and understand the effect of changing parameters on the model computations for each storage and flux term. In the next chapter, these
visualizations will be integrated with a client-side web application for performing simulations and calibration to better understand both the model theory and the model output.

Finally, the web-based interface for interacting with these visualizations demonstrates how the web could serve as a useful platform for interactive model documentation. The ability to generate these interactive visualizations using only standard web languages is a relatively new advance in web technologies. The traditional paper-based documentation formats are inherently static, which limits the ability to use visualizations to enhance cognitive processing. Although similar interactive visualizations could be created using desktop software, web-based documentation is more accessible and does not require any specific software other than a modern web browser. New approaches to web-based model documentation using these interactive visualizations will ultimately improve how models are used, communicated, and understood for supporting environmental management and decision making.
Chapter 5

Web-based Interactive Watershed Model

This chapter presents a client-side web application containing multiple interactive visual interfaces for simulating and calibrating a dynamic watershed model. Unlike most web-based models, this application was designed to run independently from the server by storing input data and model parameters within the browser and maintaining application state across multiple pages and between browser sessions. The application provides a series of interfaces for loading input data, performing simulations, evaluating the goodness of fit, and calibrating parameter values using observed data. These interfaces demonstrate the use of multiple interactive visualizations designed to improve understanding of dynamic model behavior. This application is available online at:

http://abcd.walkerjeff.com

5.1 Introduction

Simulation models are important tools for supporting environmental research, management, and decision making (Friedman et al., 1984; NRC, 2007). Models are commonly used to confirm theories, perform experiments, and compare the effectiveness of alternative management strategies. However, many models are often used as black boxes with the user providing input data and the model returning output data. Understanding how the model responds to changes in the input data or parameters is a common and major challenge. Although the model theory and
underlying equations can provide some insight into the model behavior, it is
difficult to understand the behavior of dynamic simulation where boundary
conditions and inputs change from time step to time step.

One approach to understand dynamic model behavior is through interactive
visualizations and computational steering. Marshall et al. (1990) introduced a visual
interface for controlling a 3D hydrodynamic model of Lake Erie. The interface
allowed the user to change the parameter values and see the model output being
generated in real time, a process known as computational steering (van Liere et al.,
1997). Unlike other forms of interactive visualizations that focus on post-
processing model results, computational steering allows the user to directly interact
with the model and not just the model output. This approach thus provides a direct
link between the model parameters and the output visualization. Marshall et al.
(1990) suggested that computational steering increases the researchers’
productivity and understanding of the model when compared to the traditional post-
processing techniques. Although the use of computational steering methods for
performing interactive modeling has been explored for over two decades, it has yet
to be integrated into web-based approaches to environmental modeling and
decision support.

The ability to perform interactive modeling on the web has been limited in part by
the traditional reliance on server-side architectures where the model is executed on
the server and results displayed in the browser (see Section 2.3 of Chapter 2).
Although there have been some advances in web-based interfaces such as
interactive maps that allow the user to view model results in a spatial context, the
model results are often presented as static images generated on the server. These interfaces may improve the ability to discover, access, and retrieve model results, but they do not allow the user to directly interact with the model. Changing model inputs or evaluating alternative scenarios often requires filling out text boxes through a web-based form, submitting the form to the server, and then waiting for the results to be generated and returned to the user. This lack of interaction limits the user’s ability to explore how changing parameters or scenarios affect the model results.

Client-side web applications provide a new approach to interactive modeling as discussed in Chapter 2. The performance of modern JavaScript engines and the ability to generate visualizations within the browser provide the features needed for performing numerical simulations that can be coupled with interactive visualizations. This chapter builds on the previous demonstrations in this thesis to demonstrate how client-side technologies can be used to understand the behavior of dynamic simulation models. Using computational steering, the user can directly manipulate the model parameters and see the effects on output time series and model calibration in real time. This chapter also demonstrates a number of additional client-side web application features including persistent data storage, the ability to link multiple visualizations to create coordinated views of the model, and the ability to load and save input and output data using simple text files.

In the following section, the theory, input data requirements, and parameters of the watershed model are described. Next, the application architecture is presented focusing on the core components of the application and how those components are
linked to create a single application with multiple interfaces. The user interfaces of the model are then described for loading input datasets, performing model simulations, comparing model output with observed data, searching for optimal parameter values that minimize model error, and exporting the model as a text file. Finally, the limitations and potential applications of this application and methodology are discussed, followed by some concluding remarks and suggestions for future research.

5.2 Model Description

The Web-based Interactive Watershed Model (WIWM) is based on the abcd water balance model originally developed by Thomas (1981) and introduced in Chapter 4. To briefly review, the abcd model computes streamflow based on observed precipitation and potential evapotranspiration (PET). Although the abcd model is often run on a monthly or annual time step, it is implemented on a daily time step for this application. Figure 5-1 shows the storage compartments and flows represented in the model.
The model includes two storage compartments: soil moisture and groundwater. The soil moisture gains water from precipitation and loses water through evapotranspiration, direct surface runoff and groundwater recharge. The groundwater compartment gains water from the soil as recharge and loses water as groundwater discharge. The total streamflow is computed as the sum of direct runoff from the soil and discharge from the groundwater. The history, theory, and equations of this model are described in Chapter 4.

**5.2.1 Potential Evapotranspiration**

Daily PET rates are estimated using Hargreave’s equation as suggested by Shuttleworth (1993, p. 4.18):

\[
PET = 0.0023 S_0 \sqrt{T_{max} - T_{min}} (T + 17.8)
\]  

(5-1)

where \( PET \) is the daily potential evapotranspiration (in/day), \( S_0 \) is the water equivalent of extraterrestrial solar radiation (in/day), \( T_{max} \) and \( T_{min} \) are the daily
maximum and minimum temperatures (ºC), and $\bar{T}$ is the daily average temperature computed as the arithmetic mean of $T_{max}$ and $T_{min}$ (ºC).

The water equivalent of extraterrestrial solar radiation ($S_0$, in/day) is computed as a function of latitude and Julian day (number of days since January 1 of a given year) as described by Shuttleworth (1993, p. 4.31):

$$S_0 = 15.392 \, d_r (\omega_s \sin \phi \sin \delta + \cos \phi \cos \delta \sin \omega_s) \cdot \frac{1 \text{ in}}{25.4 \text{ mm}}$$  \hspace{0.5cm} (5-2)

where $d_r$ is the relative distance between the earth and the sun (unitless), $\omega_s$ is the sunset hour angle (radians), $\phi$ is the latitude (radians), and $\delta$ is the solar declination (radians). With the latitude defined by the user, the remaining variables are computed as:

$$d_r = 1 + 0.033 \cos \left( \frac{2\pi}{365} J \right)$$  \hspace{0.5cm} (5-3)

$$\delta = 0.4093 \sin \left( \frac{2\pi}{365} J - 1.405 \right)$$  \hspace{0.5cm} (5-4)

$$\omega_s = \cos^{-1}(- \tan \phi \tan \delta)$$  \hspace{0.5cm} (5-5)

where $J$ is the Julian day. More details on these equations can be found in Shuttleworth (1993).

### 5.2.2 Input Data Requirements

The input data requirements for the model include the watershed latitude and daily time series of precipitation, minimum and maximum air temperature, and observed streamflow. The precipitation data are a direct input to the model. The latitude and air temperature data are used to estimate PET as described in Section 5.2.1. The
observed streamflow data are used for model calibration and must be represented in units of depth by dividing the volumetric flow rate by the watershed drainage area. All water fluxes are specified in units of inches/day and all temperatures in degrees Celsius.

5.2.3 Parameter Descriptions
The model includes two initial conditions and four parameters for a total of six user-defined values. The two initial conditions represent the initial depth of water contained in the soil moisture and groundwater storage compartments. The four parameters include:

- \(a\): propensity for runoff and recharge to occur when soils are undersaturated \((0 \leq a \leq 1)\)
- \(b\): an upper limit on the sum of evapotranspiration and remaining soil moisture for each time step \((b \geq 0)\)
- \(c\): the fraction of streamflow arising from groundwater discharge vs. direct runoff \((0 \leq c \leq 1)\)
- \(d\): base flow recession constant assuming the groundwater behaves as a linear reservoir \((d \geq 0)\)

The effect of these parameters on the model behavior can be explored using the interactive mass balance diagram described in Chapter 4. Additional discussion of these parameters, their relationships to physical properties of the watershed, and their typical ranges across the United States can be found in Thomas (1981), Thomas et al. (1983), Alley (1984), Vandewiele and Xu (1992), Fernandez et al.
5.2.4 Example Dataset
For the purpose of demonstration, an example input dataset for the West Branch of the Westfield River watershed was compiled. The West Branch of the Westfield River has a 94-mi² watershed and is located in western Massachusetts (42.24 ºN, 72.90 ºW). Daily observed streamflow data were downloaded from the U.S. Geological Survey (USGS) National Water Information System (NWIS) (Station ID: 01181000) (USGS, 2014a). The daily precipitation and air temperature data were obtained from the Global Historical Climatology Network – Daily (GHCND) dataset through the National Climatic Data Center (NCDC) Climate Data Online (CDO) web portal (Station USC00199972 in Worthington, MA) (Menne et al., 2012). Missing streamflow and air temperature data were filled by linear interpolation. Missing precipitation measurements were assumed to be zero. The input dataset spans a 5-year period from 2000 through 2004.

5.3 Application Architecture
The Web-based Interactive Watershed Model (WIWM) is a client-side web application designed to run independent of the server. The WIWM application is written in the standard web languages (HTML, CSS, and JavaScript) and executed entirely within the browser. The server simply provides the files necessary to run the application. The application data are stored within the browser allowing the user to access the same data from multiple pages and to maintain application state between sessions. In other words, after loading an input dataset and calibrating the
model, the user can close the browser and later return to the same model without having to re-load the input data or re-define the parameters. The application can also export the model data as a text file, which can then be re-loaded by the same user or shared with another user.

5.3.1 Application Components
The overall architecture and components of the WIWM application are shown in Figure 5-2.

The application was designed around two central components: the Application Model and the Application View. The Application Model provides a data structure that stores the current state of the application including the parameter values, input data, and name and latitude of the watershed. Note that the word “model” in this case refers to a “data model” and not the simulation model, which is a separate
component. The Application View is the primary presentation layer that connects the various components of the application. When the user navigates to any of the pages, the Application Model and Application View are initialized and used to construct the interface, load the model data (if available), and connect the components. By using this design, the Application Model and Application View are re-used across the different pages indicated in Figure 5-2.

In order for the application data to be accessible from each page, the data are stored in the browser’s LocalStorage system. LocalStorage is an HTML5 Application Programming Interface (API) that is part of the Web Storage specification and was created to replace the traditional cookie-based storage system (W3C, 2013d). The LocalStorage API provides a key-value storage mechanism that allows web applications to store data within the browser on the user’s computer. The data in LocalStorage can then be accessed across multiple pages and between browser sessions. By using this approach, a client-side web application can maintain application state without sending the data to the server. The Application Model is synchronized with LocalStorage so that parameter values and input data can be used across the different pages and stored even after the browser is closed.

When the user navigates to one of the pages, the Application View determines which page is currently being loaded and initializes the interface for that page. For example, if the user visits the Setup page, then the Setup view is loaded. By using the centralized Application Model and Application View, the different pages all share the same core logic and model data. The Application View also creates an Event Dispatcher, which coordinates the user interactions and application behavior.
For example, when the user changes a parameter, an event is triggered to recompute the model solution and update the visualizations.

### 5.3.2 Frameworks and Libraries

The WIWM application was developed using a number of free and open source software (FOSS) frameworks and JavaScript libraries. The data model, views, and event dispatcher were built using the Backbone JavaScript web framework (Backbone, 2014). Backbone, which was also used for the Web-based Interactive River Model (WIRM) described in Chapter 3, is based on a variation of the Model-View-Controller (MVC) design pattern. MVC is a common approach to software development that separates the roles of data (Model), presentation (View), and logic (Controller). Chapter 3 provides more details about Backbone and the MVC pattern. The visual aesthetics and layout of the application were built using the Twitter Bootstrap front-end framework (Twitter, 2014). Bootstrap includes CSS and JavaScript files for defining a grid-based visual layout of the pages. Bootstrap also provides interface controls and other features such as dialog boxes, buttons, and tables, among others.

In addition to Backbone and Bootstrap, the application uses additional JavaScript libraries including JQuery, underscore.js, RequireJS, and D3. JQuery and underscore provide a number of features and functions for manipulating the elements on the page, handling objects and arrays, and ensuring cross-browser compatibility of common JavaScript functions (jQuery, 2014; Underscore.js, 2014). RequireJS is a dependency management system that provides a structured approach to loading individual JavaScript libraries and dependencies (RequireJS,
RequireJS also optimizes the application code by joining all individual JavaScript files into a single file and compacting the source code to minimize the file size. This optimization drastically reduces the number of requests needed to obtain the application files from the server, which improves the loading time of the application. Finally, the Data Driven Documents (D3) JavaScript library is used to create visualizations of the model output using the scalable vector graphics (SVG) format (Bostock et al., 2011). More information about D3 can be found in Section 2.2.2 of Chapter 2.

5.4 Application Interfaces

5.4.1 Theory Page
The theory page provides an overview of the model theory, structure, input data requirements, and model parameters (Figure 5-3). The theory page allows to the user to explore the effect of changing parameter values using the mass balance diagrams described in Chapter 4.
Description

The ABCD water balance model is a simple hydrologic model for simulating streamflow in response to precipitation and potential evapotranspiration developed by Thomas (1989). The model is comprised of three storage compartments: soil moisture, groundwater, and surface flow. The soil moisture component stores water from precipitation and loses water to evapotranspiration (ET), surface runoff, and groundwater recharge. The groundwater compartment gains water from recharge and loses water as discharge. The total streamflow is the sum of surface runoff from the soil moisture and groundwater discharge.

Input Data

The model runs on a daily time step and requires input time series of precipitation, minimum and maximum air temperature, and observed streamflow. The soil temperature data are used to compute PET using the method described by Shultzworth (1981). More information about the input data can be found on the Setup page.

Parameters

There are four parameters governing the model behavior:

- $\alpha$: controls the amount of soil and recharge that occurs when the soils are under-saturated.
- $\beta$: controls the saturation level of the soils.
- $\gamma$: defines the ratio of groundwater recharge to surface runoff.
- $\delta$: controls the rate of groundwater discharge.

Mass Balance Diagrams

The model equations are represented by the interactive visualizations shown below. Each visualization demonstrates how the model computes the storage terms and fluxes for each time step. The diagrams are based on a mass balance:

$$\text{Storage}_i = \sum \text{Outflows}_i - \text{Inflows}_i$$

This equation means that the sum of remaining storage and total inflow must equal the sum of initial storage and total inflow.

For the soil moisture compartment, the mass balance equation is:

$$\text{Soil Moisture}_i = \text{ET}_i + \text{Precip}_i$$

For the groundwater compartment, the mass balance equation is:

$$\text{Groundwater}_i = \text{Discharge}_i + \text{Recharge}_i$$

References


Figure 5-3: Theory Page
5.4.2 Setup Page
On the setup page, the user loads the input data and sets the name and latitude of the watershed. Initially, the setup page provides instructions on how to load an input dataset and set up the model (Figure 5-4). The input data must be in a comma-separated value (CSV) file containing daily time series of precipitation, minimum air temperature, maximum air temperature, and observed flow. To load the input data, the user simply drags and drops the CSV file from their desktop onto the drop area. When an input file is dropped, the application reads and validates its contents, and then stores the data in the Application Model. The input data are automatically displayed as a set of time series charts for the user to verify the completeness of the dataset and ensure that the loading process worked correctly (Figure 5-5). In addition to loading an input dataset, the user must specify the name of the watershed and the latitude, which is used for estimating PET. After loading the input data and setting the watershed name and latitude, the user can click the Save button to store the current data in LocalStorage. When this process is complete, the input data will then be available from the other application pages and the user can proceed to the simulation page.
5.4.3 Simulation Page
The simulation page provides an interface for performing model simulations. This interface provides a set of controls, parameter sliders, mass balance diagrams, output visualizations, and a mass balance summary (Figure 5-6). The controls include a set of buttons for saving and deleting the model, exporting the model output as a CSV file, and displaying a Help window. Below the controls are a set of sliders for changing the model parameters and initial conditions. These sliders are grouped by storage compartment with one group for the soil moisture compartment and the other for groundwater. For each group of sliders, the corresponding mass balance diagram is displayed to reflect the underlying model theory based on current parameter values. Initially, the simulation page provides a single output visualization showing the time series of observed and simulated streamflow. The user can add additional charts to show any combination of input or computed variables. This allows the user to create customized views of the model inputs and components. For example, Figure 5-7 shows the simulation page after adding a chart displaying the direct runoff and groundwater discharge fluxes. Finally, the mass balance summary indicates whether the simulation is gaining or losing water based on the sum of total inflows, outflows and change in storage.
The simulation view includes three types of interaction. First, when the parameter values are changed, the simulation is automatically re-computed and the output
charts updated. This interaction allows the user to explore how the parameters affect the simulated streamflow and any other variables shown in additional charts. Because all charts are linked to the same underlying Application Model, changes in parameter values are reflected simultaneously on all the charts as well as the mass balance diagram. This interaction is useful for understanding how the parameters affect not only the simulated streamflow, but the individual components within the model.

The second interactive feature is the ability to change the time period shown on the output charts by zooming and panning over the top chart. The user can zoom in and out of the top chart using the mouse scroll wheel, and pan horizontally by dragging the chart to the left or right. Changing the time period on the top chart also changes the period of all other charts. This interaction allows the user to explore the model behavior during specific time periods such as individual storm events, and to understand why the model might deviate from the observed streamflow over certain periods. For example, Figure 5-8 shows the simulation during September 2000 when the observed flow indicates a series of storm events but the simulated flow shows no events. By adding a chart showing precipitation during this period, the user can see that there were no storm events reflected in the precipitation data during this period, which explains why the simulated flow deviates from the observed flow. The lack of precipitation during this period is likely due to missing values in the input dataset.
The final interactive feature is a link between the time series charts and the mass balance diagrams. In order to understand how the model computes each value in the time series, the user can move the mouse cursor over any of the time series charts to show the corresponding calculations on the mass balance diagram. For the date corresponding to the mouse position, the available water, outflows, and remaining soil moisture are highlighted on the mass balance diagram. Figure 5-9 shows an example where the mouse is positioned on a day in June when the available water is 6.2 inches. Based on this amount of available water, the corresponding location on the mass balance diagram is highlighted to show the computed values of recharge, runoff, evapotranspiration, and remaining soil moisture. The values of these four variables are also shown in the chart at the bottom of Figure 5-9. This linkage between the time series charts and the mass balance diagrams can be used to create an animation of the model computations by simply moving the mouse from left to right over a time series chart. As the mouse cursor moves from one day to the next, the highlighted position on the mass balance
chart moves to reflect the sequence of computations. This interaction thus provides a means of visualizing the algorithm used to compute the model solution.

![Mass Balance Diagram Interaction on Simulation Page](image)

**Figure 5-9: Mass Balance Diagram Interaction on Simulation Page**

### 5.4.4 Calibration Page

The calibration page provides diagnostic plots and summary statistics for evaluating the goodness of fit of the simulation (Figure 5-10). This view includes a time series chart, scatterplot, and cumulative-distribution-frequency (CDF) plot of the observed and simulated daily streamflow. The calibration view also shows two goodness of fit statistics: the root mean square error (RMSE) and the Nash-Sutcliffe efficiency coefficient (NSE). All charts and both statistics are based on the log10 transform of the observed and simulated flows. The RMSE and NSE are computed by:
\[ RMSE = \sqrt{\frac{1}{n} \sum_{t=1}^{n} (\log_{10}(Q_{obs}^t) - \log_{10}(Q_{sim}^t))^2} \] (5-6)

\[ NSE = 1 - \frac{\sum_{t=1}^{n}(\log_{10}(Q_{obs}^t) - \log_{10}(Q_{sim}^t))^2}{\sum_{t=1}^{n}(\frac{1}{n} \sum_{t=0}^{n}(\log_{10}(Q_{obs}^t)))^2} \] (5-7)

where \(Q_{obs}^t\) and \(Q_{sim}^t\) are the observed and simulated streamflows for time step \(t\), respectively, and \(n\) is the number of time steps.

Figure 5-10: Calibration View

As with the simulation page, the calibration page allows the user to change parameter values using the interactive sliders. When a parameter is changed, the model solution is automatically re-computed and the visualizations and goodness of fit statistics updated. This interaction allows the user to quantitatively explore how the different parameters affect the relationship between simulated and
observed flow. For example, the user can see how changing parameter $d$ (the groundwater recession coefficient) primarily affects the low flows of the simulation. Using the statistical measures of error shown on this page, the user can calibrate the model by adjusting the parameters to minimize the model error. This page thus provides both visual and quantitative representations of model error.

However, the calibration page only shows the model output and errors for a single set of parameters. Because it does not track the goodness of fit over different combinations of parameter values, it does not provide a direct means of minimizing the model error over the entire parameter space. The optimization page described in the following section provides another view of the model designed for searching the parameter space and finding the set of parameter values with the lowest error.

5.4.5 Optimization Page
The optimization page provides an interface for searching the parameter space to minimize model error. This interface includes a time series chart of observed and simulated streamflow and scatter plots of RMSE versus parameter value for each of the four model parameters. These scatter plots are commonly known as “dotty plots” (Beven, 2009). Unlike the previous interfaces, the optimization interface tracks the simulation history by storing the parameter values and corresponding error for each run. When a parameter is changed, a new run is added to the history and another point is added to each dotty plot. The interface also keeps track of the “best” run having the lowest RMSE over the current history.

The optimization view can be used to search the parameter space using a simple Monte Carlo simulation. After clicking the Start button, the interface will begin
generating random sets of parameter values one at a time until the Stop button is pressed. The parameter values are sampled from uniform distributions with ranges corresponding to the minimum and maximum values of each slider. Figure 5-11 shows the optimization interface after 500 iterations of the Monte Carlo simulation. The red line on the time series chart and the red dot on each dotty plot represent the current simulation based on the values set by the parameter sliders. The blue line and blue dot represent the best run with the lowest RMSE. The user can load the parameter values corresponding to the best run by clicking the “Best” button or clicking on the blue dot on the dotty plots. The user can also load other previous runs by clicking one of the black dots. When a dot is clicked, the parameter values corresponding to that run are loaded. This allows the user to inspect the differences between the runs and understand why some runs have higher or lower error than others.

![Figure 5-11: Optimization Page after 500 Monte Carlo Simulations](image)
In addition to performing a random Monte Carlo search, the optimization page can also be used to perform a manual gradient-descent search. By dragging one of the parameter sliders, the user can generate a curve showing the change in RMSE over a range of values for each parameter. For example, Figure 5-12 shows the effect of changing parameter $a$ from 0.9940 to 0.9998 with all other parameters remaining constant. The dotty plot for parameter $a$ shows a minimal value at 0.9988. The user can then load that value of $a$ by clicking the “Best” button and move on to another variable. However, because of the interaction between parameters, changing one parameter will affect the optimal value of another. The user must therefore sequentially change each value until the optimal run converges on a local minimum. Figure 5-13 shows an example after performing this sequential and iterative search method. It is important to note that because of the non-linearity of the model, this minimum is not guaranteed to be a global minimum.

Figure 5-12: Optimization Page showing RMSE Curve for Parameter $a$
5.5 Discussion and Conclusions

The application presented in this chapter demonstrates the potential for client-side web applications to support highly interactive and visual simulation modeling on the web. Unlike traditional model software that requires a series of steps to set the parameters, run the simulation, and generate visualizations of the model output, this application links these steps into an interactive and visual user interface. The user can explore not only the sensitivity of the output data to changes in parameters, but also the interaction between the individual components and processes within the model. The ability to perform data input and output using standard text files and maintain application state without any dependence on the server are also new and unique features of this application.
For practical purposes, this web application could be used as an educational tool for teaching the modeling process and the dynamics of a simple watershed model. Students and inexperienced modelers can use this application to learn about model calibration and parameter optimization, and how the various components of the model behave and interact. Because the model is built using web standards, it is easily accessible to anyone with a modern web browser and Internet connection. However, for real-world use cases, additional research is needed to improve the underlying hydrologic model and provide additional features to support decision making. The addition of a snow accumulation and snow melt module could improve the model performance in northern latitudes where snow dynamics affect the watershed hydrology. The addition of human-induced flows such as groundwater pumping and surface water withdrawals could be integrated to evaluate the effect of human activities on streamflow. Additional non-linear or genetic optimization methods could also be added to search for optimal parameter values with minimal model error. Other goodness of fit statistics and mass balance summaries could also be incorporated.

The use of client-side web applications for supporting web-based environmental modeling requires further research to identify the computational limitations of this approach. Although this particular application performs well for simulations on the order of 1-10 years, larger datasets result in delayed interaction. Alternative approaches such as using the HTML5 canvas element could improve the performance of the interface by using a raster-based graphics format instead of the vector-based SVG format (see Section 2.2.2 of Chapter 2). Additional performance
enhancements might also be achieved through the use of the standard Web Workers API, which provides support for multi-threaded web applications. With the Web Workers API, the simulation engine could be executed in its own thread, thus freeing computational resources for the user interface. Although client-side web applications limit the size and complexity of the model, as well as require that all code be written in JavaScript, the exact limitations and potential uses of this approach are not yet clear and require more research. Nonetheless, the results of this research suggest this limitation is far less restrictive than previously believed.

Another potential improvement to this application would be to use a server-side database for remotely storing the model input data and configurations as was demonstrated in Chapter 3. Instead of saving and loading model configurations using text files, the user can store their models on a centralized server, which can promote model sharing and collaboration. The application could also be configured to obtain input data directly from external data sources such as USGS and NCDC using web services. This would allow the user to easily update model input data as new observations are available. These features will be demonstrated in the following Chapter 6.

Finally, although this application was designed to promote model understanding, more research on the cognitive benefits of interactive visualizations and computational steering is needed. User studies could be performed to compare the effectiveness of alternative visualizations and views of the model. The arrangement of the interface controls and the general workflow of the application could also be improved through user feedback. However, the benefits of interactive visualization
in enhancing cognition are well known and suggest web-based interactive model interfaces provides a viable means of improving model understanding (Card et al., 1999; Yi et al., 2007; Liu et al., 2008; Liu and Stasko, 2010).

In conclusion, the WIWM application demonstrates a novel web-based approach to dynamic environmental modeling designed to facilitate model understanding and accessibility. Although the application itself is meant to be a demonstration, the concepts, architecture, and functionality could be adapted to a wide variety of web-based environmental applications. The interactive user experience provided by this approach improves model understanding and can ultimately lead to more effective use of web-based environmental modeling and decision support systems.
Chapter 6

The Living Model

This chapter demonstrates the concept of a Living Model. A Living Model is defined as a web application that provides an interactive visual interface for continuously monitoring and refining a simulation model by linking the model to input data sources. By utilizing web services of external data providers, the simulation period and observation dataset for evaluating model performance are automatically extended as new data become available.

The Living Model concept is implemented in this chapter using the abcd water balance model described in Chapters 4 and 5. Unlike the demonstration in Chapter 5 that included only a client-side web application, the demonstration in this chapter includes a server-side web application that serves as a data integration and management system for updating input and observation datasets retrieved from external data providers. This chapter also includes a modified version of the client-side web application presented in Chapter 5 that provides an interactive visual interface for exploring the model behavior, updating the calibration, and comparing model output to observation data over different time periods. The application also allows the user to save changes to the model configuration and parameter values on the server in order to promote the long-term persistence and availability of the model. This application is available online at:

http://living.walkerjeff.com
6.1 Introduction

The development, application, and refinement of environmental models is often a cyclical process (USEPA, 2009). The model life cycle begins with the need to support some decision, action, or management plan. Significant financial and human resources are spent collecting data, developing theory, and incorporating local knowledge to create a representation of the system for comparing alternative decisions or management strategies. However, after the initial project is complete, the model is frequently left sitting idle until the next phase of the cycle when the model is resurrected, updated, and extended using new observation data. This period of idleness can span years, even decades, and leaves the model generally inaccessible to stakeholders, decision makers, resource managers, and the general public. Therefore, although the modeling process may be cyclical, it is also sporadic and disjointed, and reduces the return-on-investment for the resources and knowledge that went into the original model development.

Modern web technologies provide a new infrastructure for automating much of the model update process. Major data providers such as the U.S. Geological Survey (USGS), U.S. Environmental Protection Agency (USEPA), and the National Climatic Data Center (NCDC) now offer web services designed to retrieve observation data in standardized data formats. The Application Programming Interfaces (APIs) provided through these web services are designed specifically for enabling machine-to-machine communication as opposed to the traditional web-based interfaces designed for humans to retrieve data. These web services can be used to automatically fetch, parse, process, and integrate observation data, and thus
eliminate much of the manual effort required to update model input datasets and extend simulation periods.

Through the research described in this thesis, the potential for utilizing these new web technologies to link models with their input data sources led to the concept of a ‘Living Model.’ The Living Model is a web application designed for continuously using, evaluating, and refining a simulation model through an interactive visual interface. By linking the application to input data sources, the model’s input and observation datasets can be continuously and automatically updated as new data become available. Compared to traditional model life cycles that involve long periods of time when the model is idle and inaccessible, the Living Model provides great potential for increasing the model utilization and improving how we understand and manage environmental systems. The Living Model could be used for adaptive management, evaluating model performance during extreme events, improving the model theory, and identifying changes or trends in the system behavior. Furthermore, the model could be continuously re-calibrated and validated over time as new data become available.

However, the automation of the model update process raises a number of issues regarding data quality assurance, data provenance, and the ability to keep track of which data are used for model calibration and validation. Observation datasets are often incomplete and contain erroneous measurements and outliers. The implementation of a Living Model must therefore provide a means for supervising the process of obtaining new data, and ensuring that the user is aware of how missing or abnormal data are handled by the automated system. The provenance of
the input data must also be available allowing the user to trace the input dataset back to its original source. Furthermore, the Living Model must provide some way of tracking which data were used for calibration as the input dataset is updated over time. However, with proper consideration of these issues, the Living Model opens new opportunities for improving the cost-effectiveness and utility of environmental models to understand and manage environmental systems.

In this chapter, a web application is presented to demonstrate the concept of a Living Model. In the following section, the overall architecture of the Living Model application is presented focusing on the primary roles and relationships of the server-side and client-side application components. Next, the design and implementation of the server-side web application for managing and evaluating the input datasets is described. Following the server-side application is a description of the client-side web application, which provides an interactive visual interface to access, evaluate, and refine the simulation model as new input data become available. Finally, the benefits and limitations of this approach for implementing the Living Model concept are discussed, followed by some conclusions and suggestions for future research.

### 6.2 Living Model Architecture

The Living Model is demonstrated using the *abcd* water balance model described in Chapters 4 and 5. The Living Model application is comprised of two components: a server-side web application for managing input datasets and model configurations using a relational database and file system, and a client-side web application for providing an interactive visual interface to the simulation model.
(Figure 6-1). The two web applications exchange information using Asynchronous JavaScript and XML (AJAX)-style communication, which are described in Section 2.1.4.2 of Chapter 2 (Garrett, 2005). By using AJAX-style communication, the client-side application can retrieve and update the model data without refreshing the page, thus improving the user experience (see Section 2.1.4.2 of Chapter 2).

The input data are retrieved by the server-side application from two data sources: the U.S. Geological Survey (USGS) National Water Information System (NWIS) Water Services and the National Climatic Data Center (NCDC) Global Historical Climatology Network – Daily (GHCND) database (Menne et al., 2012; USGS, 2014b). The following section describes the design and implementation of the server-side application including its components and the process of fetching, updating and managing the input datasets.
6.3 Server-side Web Application

The server-side application is comprised of three primary components: a web application, a relational database, and a file system (Figure 6-1). The web application was built using the Flask web framework, which is implemented using the Python programming language (Flask, 2014). Flask is a web framework designed for customization through extension modules, which will be described in the following sections. The relational database is stored using the SQLite database engine (SQLite, 2014). The database stores metadata related to the various input datasets as well as model configuration data and the relationships between various
datasets. Finally, the input data are stored as text files on the server’s file system.

The following section describes the database schema and implementation.

6.3.1 Database Architecture
The server-side database contains four separate tables for storing the application data (Figure 6-2). Each table corresponds to an individual data model, which will be referred to as a data ‘object’ to avoid confusion with the word ‘model.’ The four data objects are defined and managed using SQLAlchemy and the Flask-SQLAlchemy extension (Flask-SQLAlchemy, 2014; SQLAlchemy, 2014). SQLAlchemy is a Python module that provides an Object Relational Mapper (ORM) for specifying the database schema and performing standard CRUD (create, read, update, and delete) operations using object-oriented Python syntax instead of the Structured Query Language (SQL). Each table in the database is defined by a Python class, which contains attributes and methods. The attributes are used to automatically generate the database schema by defining the fields of each table and the relationships between the tables. The class methods provide additional functions for processing and managing the associated database records. Figure 6-2 shows the attributes, methods, and relationships of the four data object classes used in the Living Model application.
As shown in Figure 6-2, the Living Model database contains four data objects classes called Model, Watershed Dataset, USGS Dataset, and GHCND Dataset. The Model object stores data for defining an instance of the simulation model. The Model attributes include a name for the model, created and updated timestamps, the values of each hydrologic parameter ($a$, $b$, $c$, and $d$), the start and end dates of the calibration period ($cal_{start}$ and $cal_{end}$), and the start and end dates of the input dataset when the model was last saved ($last_{start}$ and $last_{end}$). The Model class also includes an attribute that defines a relationship between the Model instance and a Watershed Dataset, which stores information about the input dataset for each model. The Watershed Dataset attributes include a name, start and end
dates, and two foreign keys that define relationships to one USGS Dataset and one GHCND Dataset.

The USGS Dataset and GHCND Dataset objects represent individual USGS streamflow gauges and GHCND meteorological stations, respectively. Both dataset objects include attributes for the station ID, station name, latitude, longitude, start and end dates, number of missing values, and created and updated timestamps. Each USGS Dataset also stores the drainage area associated with the gauging station for converting the observed flows from volumetric units (ft³/s) to depth units (in/day) that are used in the abcd model (see Chapter 5).

The Watershed, USGS, and GHCND Dataset objects include methods for fetching, processing, updating, and loading observed data and site information. The `fetch_site` and `fetch_data` methods retrieve site information and raw observation data, respectively, from the associated data provider and store the results as text files on the server. The USGS Dataset fetches site information and observation data from the USGS NWIS Site Service and Daily Values Service, respectively (USGS, 2014b). These two web services provide a Representational State Transfer (REST) API that accepts requests and returns data in standardized data formats.

The GHCND Dataset fetches site information and observed data from the NCDC File Transfer Protocol (FTP) server, which provides text files containing the entire observation dataset for each GHCND station (Menne et al., 2012). Although the GHCND data could be accessed from the NCDC Climate Data Online (CDO) web services, which provide a REST API similar to the USGS web services, the CDO
services are better suited for retrieving smaller (e.g. daily or monthly) subsets of the data. Therefore, the FTP server was used to obtain the complete data record for each station in order to retrieve multi-year input datasets.

In summary, the Living Model database stores the relationships between each instance of the simulation model (i.e. a Model object) and each input dataset (i.e. a Watershed Dataset object). The input dataset is then comprised of a USGS Dataset and a GHCND Dataset. Using these related data objects, the Living Model application can automatically update the input dataset for a given simulation model instance with a single request as described in the following section.

6.3.1.1 Update Data Process

Figure 6-3 shows a sequence diagram of the update process. The process begins when the application receives a request from the user to update the Watershed Dataset. This request is then passed to the associated USGS Dataset followed by the GHCND Dataset. To update the USGS Dataset, the raw data are first fetched from the USGS NWIS web services. The raw data are then converted to CSV format and saved to the local file system in a folder specified by the USGS Station ID. Next, the raw data are processed to fill missing values using the methods explained in Section 6.3.1.2 below. The processed data are saved as a CSV file in the same folder as the raw data file. The start date and end date of the processed dataset and the missing count attribute are updated in the relational database. After the USGS Dataset is updated, the same process is applied to the GHCND Dataset. After both the USGS and GHCND datasets are updated, the Watershed Dataset merges the two processed datasets into a single input dataset. The merged dataset
is stored as a CSV file in a folder specified by Watershed Dataset ID attribute. Finally, the start date and end date attributes of the Watershed Dataset are updated in the database.

![Sequence Diagram for Updating a Watershed Dataset](image)

**Figure 6-3: Sequence Diagram for Updating a Watershed Dataset**

### 6.3.1.2 Dataset Processing

Because the simulation model requires a continuous input dataset (i.e. a dataset without gaps due to missing values), the raw data obtained from the USGS and GHCND datasets are processed to identify and fill missing values. For the
demonstration purposes, the application uses two simple approaches to fill gaps in
the observed datasets. For the precipitation data, missing values are set to zero. For
the streamflow and air temperature data, missing values are filled by linear
interpolation based on the nearest available observations.

A variety of alternative methods could also be implemented for estimating missing
values such as through regressions against observation data from other near-by
stations, cubic spline interpolation, or stochastic processes, to name a few.
Furthermore, additional processing steps such as detecting outliers or abnormal
values could be incorporated to improve the quality of the input dataset. For the
purpose of demonstrating the Living Model concept, the simple approaches for
filling data gaps described above are sufficient. In any event, the actual differences
between methods for filling gaps are often inconsequential when there are few
missing values and no long-term gaps in the dataset.

6.3.2 Database Management Interface
The server-side application provides a database management interface for creating,
retrieving, updating, and deleting instances of the four data object types described
in Section 6.3.1. The interface is automatically generated using the Flask-Admin
extension, which creates for the user a series of webpages corresponding to each of
the four data objects (Flask-Admin, 2014). For each data object, the interface lists
the existing database records and provides forms for creating, editing, and deleting
individual records. Figure 6-4 shows an example of the list page for the Watershed
Dataset object class. The database record list includes icons for editing and deleting
each record. Clicking the edit button opens a form for editing the individual
attributes of the record. Figure 6-5 shows an example of the edit form for a GHCND Dataset corresponding to the station USC00196783 in Reading, MA.

![Database Record List for the Watershed Dataset](image1.png)

**Figure 6-4: Database Record List for the Watershed Dataset**

![Database Edit Form for the GHCND Dataset](image2.png)

**Figure 6-5: Database Edit Form for the GHCND Dataset**

In addition to the list of records and the edit form, the database management interface provides a form for creating new records in each table. Figure 6-6 shows an example of the creation form for a USGS Dataset. Note that only the Station ID

![Database Creation Form for the USGS Dataset](image3.png)
and start date attributes are required because the remaining attributes are automatically populated from the USGS Site Service API using the `update_site()` method.

![Database Create Form for the USGS Dataset](image)

**Figure 6-6: Database Create Form for the USGS Dataset**

The database management interface therefore provides a collection of pages for creating, editing, and deleting records in the database. However, because the actual datasets are stored as text files on the server, another interface was needed for accessing and viewing the observation datasets associated with each database record.

### 6.3.3 Dataset Interface

The dataset interface provides a series of pages for inspecting, updating, and downloading the observation datasets. For each of the three types of datasets (Watershed, USGS, GHCND), the dataset interface provides a list of the existing datasets, which include hyperlinks to the detailed view of each dataset. Figure 6-7 shows an example of a list of four USGS Datasets.
In addition to the list of existing datasets, the dataset interface also provides a detailed view of each dataset. The detailed view provides 1) a link to the external data provider site for the corresponding station (i.e. the USGS or GHCND station homepage), 2) the attributes and metadata for the dataset, 3) links to download the dataset in CSV or JSON formats, 4) buttons to request updates of the dataset or site information, and 5) visualizations of the dataset itself. Figure 6-8 shows an example of the dataset interface for a USGS streamflow gauge on the Aberjona River in Winchester, MA. The visualization shows the complete time series of the observation dataset. This visualization also indicates which values were recently updated and which values were originally missing and then filled during data processing. The time-series charts are zoomable, allowing the user to inspect subsets of the dataset. Figure 6-9 shows the USGS dataset interface for the Aberjona River with the time-series chart zoomed to July 2013–March 2014 showing the updated and filled values in more detail.
The GHCND dataset interface provides a similar detailed view, but includes three time-series charts, one for each of the three variables (precipitation, minimum and maximum air temperature). Figure 6-10 shows the GHCND dataset interface for the Reading, MA meteorological station. This figure indicates missing values that were filled in mid-2002 and recent updates occurring in 2014.
Finally, the watershed dataset interface shows the combined USGS and GHCND datasets, which comprise the input dataset for an instance of the simulation model. This interface also provides a single update button that will trigger updates of both datasets using the process described in Section 6.3.1.1.
The dataset interfaces therefore provide a means of updating each dataset and inspecting which values were filled during data processing and which were recently updated. The ability to review the source datasets is an important feature of the Living Model. Without these detailed views of the datasets, it would be difficult to identify which data were filled and which data were recently updated. These detailed views also provide an easy way of updating the datasets or downloading the raw or processed data with a single button click.
6.3.4 REST API
The final component of the server-side application is a REST-based API for accessing the simulation model objects on the server. This API provides a way for the client-side web application to retrieve model parameters and input datasets and also save changes to the parameter values on the server. The simulation model REST API was implemented using the Flask-Restless extension, which automatically generates a set of URLs and functions for retrieving the model parameters using the HTTP GET method and updating those parameters using the PUT method (Flask-Restless, 2014). Figure 6-12 shows a sequence diagram illustrating the process of loading the model, saving the model, and resetting the model to previously saved values.

The model is loaded when the client-side application is first initialized. The client-side application sends a GET request to the server, which retrieves the model data from the database and returns the data in JSON. Within this model data is a link to the input dataset for the requested model. The client-side application submits a second request for the input dataset, which is retrieved from the server’s file system. The client-side application then initializes the interface and the user can interact with the model.

To save the model to the server, the client-side application sends the parameter values using a PUT request. The server uses those values to update the corresponding attributes in the database. The server then sends a response back to the client with a status message indicating if the update was successful. The client-side application then notifies the user.
Using the reset model feature, the user can reset the parameters on the interface using the values stored on the server. This feature is useful for allowing the user to explore how the model responds to changes in the parameter values. After experimenting with different parameter values, the user is then able to return to the original set of parameters that were last saved to the server (presumably determined by prior calibration). When the user requests a reset of the parameters, a GET request is sent to the server, which fetches the current values saved in the database.
and returns those values to the client. The client then updates the parameter sliders on the interface to return to the previously-saved parameter values.

### 6.4 Client-side Web Application

The client-side web application is a simplified version of the Web-based Interactive Watershed Model (WIWM) interface described in Chapter 5 for the *abcd* water balance model. The theory and background of the *abcd* model can be found in Chapters 4 and 5. The implementation of the interface uses the same frameworks, libraries, and toolkits as described in Chapter 5.

For this demonstration, the application provides an interactive interface for calibrating the model, evaluating the model results based on updated input datasets, and inspecting the goodness of fit over different time periods. Figure 6-13 shows the living model interface for the Aberjona River in eastern Massachusetts. With reference to the numbered labels in Figure 6-13, the interface includes 1) a link to the input watershed dataset interface where the user can inspect the input dataset (in this case, the interface shown in Figure 6-11), 2) buttons for saving and resetting the model and downloading the input and output datasets as CSV files, 3) sliders for changing the model parameters, 4) an interactive time-series chart for selecting subsets of the period, 5) a time-series chart of the observed and simulated daily streamflow, 6) a scatterplot of simulated vs. observed daily streamflow, and 7) a summary table listing the start and end dates, number of days, and two error statistics. The error statistics include the root mean squared error (RMSE) and percent bias for the selected period, calibration period, and period of record.
Figure 6-13: Living Model Interface for the Aberjona River

The upper chart on the interface shows the time series of observed streamflow and can be used to select different periods by brushing. Brushing refers to a type of interaction involving simply clicking and dragging the mouse to select a period, which will create a shaded rectangle showing the selection. The selected period can then be dragged to the left or right. The start and end dates of the selected period can also be changed by clicking and dragging the left or right edges of the shaded region. After a period is selected, the time scale on the lower chart and the scatter plot are updated to show only the data within the selected period. The summary table is also updated to show the start and end dates and error statistics for the selected period. Figure 6-14 shows an example where the two-year period 2010–2011 is selected.
After selecting a period, the user can calibrate the model to the data contained in only that period by moving the parameter sliders. Changes in the parameters automatically update the model computation and visualizations in real time using the same approach described in Chapter 5. Figure 6-15 shows the interface after adjusting the parameters to minimize the RMSE and reduce the magnitude of the % Bias for the period 2010-2011. Compared to the error for the default parameters shown above in Figure 6-14, the RMSE for this two-year period was reduced from 0.68 to 0.30 and the % Bias increased from -25% to 4.1%.
After calibrating the model to a selected period, the user can save the parameters by clicking the Save button. When the model is saved, the calibration period is automatically set to the current selected period. The updated parameter values and calibration period are then sent to the server, which updates the attributes of the model in the relational database as described above in Section 6.3.4. If the update is successful, the server returns a success message and the user is notified that the model is saved and the interface is updated to reflect the new calibration period (Figure 6-16). After the model is saved, the calibration period is indicated by an orange bar across the top chart. The summary table is also updated to show the start and end dates, number of days, and error statistics for the calibration period. Note
that in this figure, the selected period is identical to the calibration period but that is not always necessarily the case.

![Living Model Interface for the Aberjona River after Saving the Model]

Figure 6-16: Living Model Interface for the Aberjona River after Saving the Model

After the model is calibrated, the user can compare the model fit between the calibration period and any other time period. This allows the user to perform model validation. If the model fit is the same (or better) over a period independent of the calibration period, then it is considered validated. The user can easily compare the model fit by moving the selected period outside the calibration period. Figure 6-17 shows a comparison between the model fit for the 2-year calibration period (2010–2011) against the subsequent period from January 2012–October 2013. The summary table indicates that the both the RMSE and % Bias are lower during the selected period (i.e. the validation period) with values of 0.24 and 1.8%,

<table>
<thead>
<tr>
<th>Selected Period</th>
<th>Calibration Period</th>
<th>Period of Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>2010-01-01</td>
<td>2010-01-01</td>
</tr>
<tr>
<td>End</td>
<td>2012-01-01</td>
<td>2013-10-31</td>
</tr>
<tr>
<td>No. Days</td>
<td>721</td>
<td>1400</td>
</tr>
<tr>
<td>RMSE</td>
<td>0.2996</td>
<td>0.2734</td>
</tr>
<tr>
<td>% Bias</td>
<td>4.1%</td>
<td>2.9%</td>
</tr>
</tbody>
</table>
respectively, compared to the values during the calibration period of 0.30 and 4.1%. In this case, the calibration may be considered validated depending on the user’s requirements for model validation.

![Image of living model interface for the Aberjona River comparing validation period](image)

*Figure 6-17: Living Model Interface for the Aberjona River comparing Validation Period*

Finally, in addition to tracking the calibration period, the living model interface also indicates updates to the input dataset. After creating and calibrating a model, the user can return to the interface at some point in the future to inspect the simulation after updating input data. The data are updated by first visiting the watershed dataset interface for the corresponding input dataset (e.g. Figure 6-11) and clicking the Update button. After updating the dataset, the user can return to the model interface, which will indicate the period of updated data with a blue bar.
over the top chart. By clicking this blue bar, the updated period is selected. Figure 6-18 shows the interface after updating the input dataset, which added three months of input data over November 2013 – January 2014. The lower time-series chart and error statistics indicate the model fit is worse, likely because the model does not include snow-accumulation and snow-melt dynamics.

![Living Model Interface for the Aberjona River with the Updated Period Selected](image)

The living model interface thus provides a means of interacting with the model, saving calibrated parameter values to the server, reviewing updated simulation periods, and comparing the model fit across different periods. The ability to identify new data and track the period used to calibrate the model are important features of
the Living Model. The interface also allows the user to update the calibration period as new data become available.

### 6.5 Discussion and Conclusions

The web application presented in this chapter demonstrates the concept of a Living Model using the *abcd* water balance model. This application provides an interactive visual interface to access, monitor, and refine the model over time by automatically updating the input and observation datasets. The application provides interfaces for managing and supervising the input datasets, allowing the user to track the provenance of input data back to the original sources. The interactive model interface was designed to facilitate model understanding and the ability to evaluate model fit under different time periods.

Although this demonstration illustrates the basic functionality and potential of the Living Model concept, it has some limitations that require further research. One of the primary limitations is the processing of raw observational data. The raw datasets often include missing or erroneous values that require identifying and estimating appropriate values for creating continuous input time series (i.e. filling gaps in the data). In this demonstration, the data processing method uses simple linear interpolation of the observed streamflow and air temperature datasets and assigns missing precipitation values to zero. However, alternative methods as described in Section 6.3.1.2 such as regression against other observation stations, cubic spline interpolation, or stochastic processes could improve the estimation of missing values. Furthermore, quality assurance methods for identifying outliers or abnormal data could be incorporated. More research on implementing these alternative
estimation methods and providing interfaces for managing these processes is needed.

The client-side web application for evaluating and calibrating the model could also be improved by providing additional diagnostic plots and error statistics. Additional parameter optimization algorithms such as those described in Chapter 5 could assist the user in calibrating the model. The interface could also be improved by allowing the user to add additional charts showing the input data and individual storage and flux terms as demonstrated in Chapter 5. As this thesis has demonstrated, client-side web technologies provide great potential for creating novel interactive interfaces designed to improve the use and understanding of simulation models. However, for the purpose of this demonstration, the model interface was designed simply to demonstrate the core functionality of linking input data sources to the simulation model.

Another limitation of this application is the lack of user authentication. Currently, the input datasets can be updated and the model calibration can be changed by any user. With the addition of user accounts, the application could be configured to allow only authorized users to make changes while still allowing any user to view the model. However, this functionality was already established in Chapter 3 and not deemed necessary for the purpose of this demonstration.

The application presented in this chapter illustrates the possibility of using the Living Model concept to improve the modeling life cycle. The ability to automatically update model input data over the web can fundamentally change how
models are used to study and understand environmental systems. Once a model is developed, it can be used to support adaptive management, monitor for changes or trends in the system dynamics, and evaluate the model performance under extreme conditions. For instance, if the model begins deviating from the observed data, this might suggest some fundamental change in the system. A water quality model, for example, might begin under-predicting the concentration of a pollutant. This behavior might suggest the introduction of a new source in the system. Similarly, a hydrologic model may begin over-predicting streamflow, which may indicate increased groundwater pumping or extractions from the basin. A Living Model could thus provide a tool for evaluating both the system dynamics as well as the model performance under varying conditions as the system evolves.

Finally, by being continuously available on the web, the accessibility of the model for non-technical audiences is greatly improved. Stakeholders and decision makers are no longer dependent on modelers to perform manual extensions of the simulation period. Although some human oversight will always be required to ensure the system works appropriately, the role of the model expert would shift from manual data processing to system oversight. This new role would allow the modeler to focus more on evaluating and improving the model theory rather than processing input datasets.

In conclusion, this research has demonstrated a novel approach to web-based modeling by utilizing the web services of common data providers for linking simulation models and input data sources. The incorporation of server-side data management with client-side web applications provides an accessible and highly
interactive application for evaluating and refining model performance and calibration. This approach thus presents new opportunities for the continual use of simulation models to better manage and understand environmental systems.
Chapter 7

Conclusions

This thesis presents a novel approach to web-based environmental modeling. This new approach focuses on the use of client-side technologies to provide interactive visualizations and user interfaces of environmental simulation models. By using client-side technologies to perform the numerical computations and output visualization within the browser (as opposed to the server), the user can directly interact with the model and explore how it behaves in response to changing parameters and input data. Traditional approaches to web-based modeling have focused almost exclusively on server-side approaches where the model is executed on the server and output data are simply displayed in the browser as static text and images. Although these existing approaches improve the accessibility of model results, they rarely provide a way of interacting with the model and understanding how the model behaves.

Advances in web technologies and development practices have enabled the web to evolve from a collection of static hyperlinked documents to interactive web applications (see Chapter 2). Recent advances in client-side web languages (HTML, CSS, and JavaScript) and performance optimizations of modern web browsers are allowing the web to become a new software platform. Modern web applications are increasingly built using client-side technologies to provide rich user experiences that rival traditional desktop software. This shift towards a more
interactive user experience provides great potential for improving how we access, use, and understand environmental models.

In this thesis, a series of web applications are presented to show how client-side technologies and development practices can be applied to environmental simulation models. By using this approach, these demonstrations illustrate the benefits and great potential for creating interactive visualizations of model theory and output designed to improve model understanding. These interactive visualizations and user interfaces allow users of all technical abilities to easily explore the model behavior and see the effects of changing parameters in real time.

The first application created for this research was the Web-based Interactive River Model (WIRM) described in Chapter 3. WIRM began as an experiment to determine whether it was even possible to perform numerical calculations and output visualization within the web browser using standard web languages. The results clearly demonstrated that not only is this possible, but the performance of modern JavaScript engines supports highly interactive visualizations of model output. Additional features were added to WIRM to demonstrate how a client-side web application could be coupled to a server-side application for storing model parameters and facilitating collaboration between multiple users. The WIRM project led to new research questions about whether web-based interactive visualizations improve understanding of model behavior and the underlying theory.

In Chapter 4, theories and research from the field of cognitive science were used to explain why interactive visualizations are useful for improving model
understanding. Two of these theories—mental models and cognitive load theory—provide a framework for understanding the cognitive effects of model interaction and visualization. In comparison to equations or static representations of the model results, interactive visualizations reduce cognitive load and promote the formation of mental models. Through the process of interacting with a visual representation of the model, the user can gain greater understanding and intuition about the model behavior.

The cognitive benefits of interactive visualizations for understanding model theory are demonstrated using a novel visualization called the mass balance diagram presented in Chapter 4. To demonstrate the use of this diagram, it was applied to a simple water balance model, the $abcd$ model. A web-based, interactive version of this diagram was created to illustrate the potential for using client-side technologies to create interactive model documentation. The results of this chapter suggested that although model equations are necessary for developing theory and proving model behavior, interactive visualizations provide a more accessible and easier-to-understand representation of model theory.

In Chapter 5, a client-side web application called the Web-based Interactive Watershed Model (WIWM) was presented. WIWM demonstrates how the client-side approach can improve understanding of model behavior. Unlike the previous demonstrations, this application focuses on dynamic simulations driven by observation data. WIWM provides multiple interfaces containing coordinated interactive visualizations for exploring model behavior as well as calibrating the model parameters. This application demonstrates the ability of client-side
applications to not only perform simulations and create customized views of the model, but also compare model output to observation data using traditional diagnostic plots and error statistics. WIWM also demonstrates the abilities to maintain application state by storing model data within the browser and load input data and save output data using text files without any dependence on the server.

Finally, Chapter 6 demonstrates the concept of a Living Model where a client-side web application is coupled with a server-side data management system for automatically updating input and observation datasets. Although the ability to couple models with data sources through the web has been previously demonstrated, the Living Model provides an interactive interface for evaluating and refining the model as new data become available rather than simply accessing the results in some static format. The Living Model improves the model life cycle by making it more continuous and by reducing the time, effort, and resources needed to extend simulation periods and update model datasets.

Although this research has focused on relatively simple environmental models, the technologies and methodology can be applied to more complex models and decision support systems. Clearly, it is not practical, or in some cases even possible, to migrate many existing environmental models to client-side platforms. However, the approach demonstrated in this thesis can improve the user interfaces for managing and evaluating any model, even those executed on the server. For example, an interactive interface of a server-side model would allow the user to explore the model results and compare those results to different datasets. Interactive
interfaces also improve how users develop and configure models by providing direct feedback on how parameters affect the underlying theory.

This approach to interactive web-based user interfaces is also not strictly limited to simulation modeling. It can also be used for data exploration and analysis. Although many environmental datasets are already accessible through interactive map-based interfaces, there are few, if any, web applications that provide an effective way of exploring and understanding observation datasets. These interfaces would provide the ability to create customized views of the data, perform statistical analyses, and manipulate the data by sorting, filtering, and aggregation.

In summary, this research provides a new approach for web-based modeling designed to improve how we use and understand environmental models and observation data. Although the web has long been recognized as a platform that can improve model accessibility, the applications and interfaces needed to interact and explore model simulations on the web have yet to emerge. The approach demonstrated in this thesis can be adapted for a variety of environmental applications including decision support systems, simulation models, data exploration and analysis tools, among many others. The central aspect of this methodology is the ability to improve the user experience through greater interaction and visualization. This new approach to web-based user interfaces has the potential to fundamentally change how we access, understand, and interact with environmental data and models. This improved accessibility and interactivity will ultimately lead to better understanding of environmental data, models, and systems and thus more informed management and decision making.
References


Nersessian, N. J. (2002). The cognitive basis of model-based reasoning in science (pp. 133-153).


