

Climate Data Computing: Optimal Interpolation, Averaging, Visualization and Delivery

by

Julien Pierret

Claremont Graduate University

and

San Diego State University

Copyright Julien Pierret, 2018 All rights reserved.

APPROVAL OF THE DISSERTATION COMMITTEE

This dissertation has been duly read, reviewed, and critiqued by the Committee listed below, which hereby approves the manuscript of Julien Pierret as fulfilling the scope and quality requirements for meriting the degree of Doctor of Philosophy in Computational Science with a Concentration in Statistics.

Samuel Shen, Ph.D, Chair
San Diego State University Department of Mathematics and Statistics
Professor

John Angus, Ph.D
Claremont Graduate University Institute of Mathematical Sciences
Professor of Mathematics

Barbara A. Bailey, Ph.D
San Diego State University Department of Mathematics and Statistics
Assistant Professor

Claudia Rangel-Escareño, Ph.D
Claremont Graduate University Institute of Mathematical Sciences
Adjunct Professor of Mathematics

C D Lin, Ph.D
San Diego State University Department of Mathematics and Statistics
Associate Professor, Statistics Associate Chair

Abstract

Climate Data Computing: Optimal Interpolation, Averaging, Visualization and Delivery

by
Julien Pierret

Claremont Graduate University and San Diego State University: 2018

This dissertation solves two important problems in the modern analysis of big climate data. The first is the efficient visualization and fast delivery of big climate data, and the second is a computationally extensive principal component analysis (PCA) using spherical harmonics on the Earth's surface. The second problem creates a way to supply the data for the technology developed in the first. These two problems are computationally difficult, such as the representation of higher order spherical harmonics Y_{400} , which is critical for upscaling weather data to almost infinitely fine spatial resolution.

For the first problem, the 4D Visual Delivery (4DVD) software technology has been created as a web application to provide appealing visualizations of climate data and to easily deliver big climate data to end users quickly and efficiently. The need for such an application is first discussed and is followed up with the initial design requirements. From here, a novel interconnected system is designed that seamlessly integrates a database, web server, and front-end computing. WebGL and JavaScript are used to generate feature rich maps and time series of global climate data. The delivered data can be sorted and their statistical properties can be easily computed. All this is done with minimal processing from the server because the computations are moved instead to the end user's web browser, taking a heavy burden off of the back-end.

The second topic is the calculation of an annual mean global climate of surface air temperature anomalies through the use of optimal interpolation of existing weather stations by using the eigenvalues and vectors calculated via a spherical harmonics representation of the global climate field. An optimal interpolation is made for the weather station data by using a

multivariate regression built on these eigenvalues and vectors which is calculated via an SVD algorithm in the spectral space. The background, mathematics, and code logic are explored for three of the main difficulties: transform of the high order spherical harmonic functions, the calculation of eigenvalues and vectors in complex space in Java, and the calculation of the optimal weights when involving large complex matrices. In the research, code optimizations and numerical verifications of the statistical theory are made and a novel cloud computing approach was explored to find optimal weights over Apache Spark, which is a distributed computing library. Cloud computing was carried out to generate many results of this research, and the cloud computing procedure is documented in this dissertation for other researchers.

DEDICATION

Dedicated to my wife and children. I do it all for you.

ACKNOWLEDGMENTS

I'd first like to acknowledge my adviser Dr. Sam Shen. Without his guidance I would not be where I am today. He completely understood that I had to balance both a family and a full-time job outside of my academic studies. This unfortunately led to me taking a little longer than I would have liked to complete this research but Sam was supportive throughout the whole time. I look forward to continue to work with you on future projects.

I'd also like to thank my wife Carolynn. You supported me in so many different ways! Carolynn would put food on my desk as I worked, took on more than her fair share of parenting duties, and was completely supportive of me throughout this whole period. I'd also like to thank my two boys Maxime and Antoine. Max (3 years old now) would come into the office and mention that he wanted to draw some "math" like I did on the scribbles of paper spread all around my desk. Antoine (almost 2) would also visit me in the office so he could be "bounced" on my knee. They helped force me to take occasional breaks as I worked through the research presented here.

Lastly, I'd like to thank my committee members in alphabetical order:

I thank Dr. Angus who pushed me to the edge of the abyss but kept me from falling in. His classes were some of the toughest, but still enjoyable, that I had taken. I thank Dr. Bailey whom I've known since before I even had a bachelor's degree. She basically taught me the foundation of all I know today! I thank Dr Lin, who first introduced me to the computational side of statistics with matrix manipulations in SAS. He showed me the rabbit hole, and I climbed down to lower level languages from there. I thank Dr. Rangel for showcasing the computational side of things. The direction I took in my research heavily showcases the computational aspect of things.

Thank you all for supporting me!

Support for the Twentieth Century Reanalysis Project version 2c dataset is provided by the U.S. Department of Energy, Office of Science Biological and Environmental Research (BER), and by the National Oceanic and Atmospheric Administration Climate Program Office.

TABLE OF CONTENTS

	PAGE
ABSTRACT	iii
ACKNOWLEDGMENTS	vi
LIST OF TABLES.....	x
LIST OF FIGURES	xi
CHAPTER	
1 Introduction	1
1.1 Background.....	1
1.2 Outline of the Dissertation	2
1.2.1 Visualization and Fast Delivery of Climate Data	2
1.2.2 Optimal Interpolation via Distributed Processing.....	4
2 Visualization and Fast Delivery of Climate Data	9
2.1 Introduction.....	9
2.2 Methodology	10
2.2.1 General idea.....	10
2.2.2 Database.....	12
2.2.3 Web Server.....	18
2.2.4 Browser	18
2.3 Results: 4DVD system functions and options	19
2.3.1 View Options: Bump Mapping	19
2.3.2 View Options: Globe	20
2.3.3 View Options: Smooth Grid	21
2.3.4 View Options: Pacific Centered.....	21
2.3.5 View Options: Additional Lines	22
2.3.6 View Options: Color Maps.....	23
2.3.7 Time series display: digital data for the climate history of a given location	23
2.3.8 Data downloads: digital data for a map or time series	24
2.3.9 Adding Example Data - NetCDF	24

2.4	An application example: visual delivery of the NOAAGlobalTemp dataset	25
2.5	Conclusions from the original paper	30
2.6	Additions Since Publication	31
2.6.1	Adding Example Data - NetCDF	31
2.6.2	New features since the publication	31
2.6.3	A new framework - Angular 2	32
2.7	Conclusions and Future Work	38
3	Optimal Interpolation via Distributed Processing	40
3.1	Background	40
3.2	Spherical Harmonic Transform	41
3.2.1	Background	41
3.2.2	Implementation	43
3.2.3	Verification	57
3.2.4	Data and Processing	59
3.2.5	Conclusion	61
3.3	Principal Component Analysis of Re-analysis data	62
3.3.1	Background	62
3.3.2	Implementation	63
3.3.3	Data and Processing	67
3.4	Optimal Weights	69
3.4.1	Background	69
3.4.2	Implementation	75
3.4.3	Validation	82
3.4.4	Data and Processing	92
3.5	Results and Conclusions	96
4	Conclusion and Future Work	104
APPENDIX		
	Notes to the Aspiring Java Numerics Programmer	106
	BIBLIOGRAPHY	112

LIST OF TABLES

	PAGE
3.1 Sum of squared errors for the two formulas at various Q-truncations.	50
3.2 Example of spherical harmonics spectral form.	54
3.3 Example of how data is indexed and stored.	54
3.4 SHTns vs Our implementation	57
3.5 Original $Y_2^1 = 1.0$	58
3.6 Rebuilt $Y_2^1 = 1.0$	59
3.7 Finding the optimal Q - SSE.....	60
3.8 Finding the optimal Q - Integral over the Sphere.....	61
3.9 Example of spherical harmonics spectral form. $Q = 7 (Q + 1)^2 = 49$	62
3.10 95% Variance Explained for each method	68
3.11 Differing “noise” value effects on weights.....	74
3.12 Sum of squared errors.....	74
3.13 Verification of first harmonic calculation only.....	80
3.14 First harmonic issues with normalized data.....	81
3.15 Original random harmonic at $t = 1$	83
3.16 Rebuilt random harmonic at $t = 1$	83
3.17 Original random harmonic at $t = 1$ (spatial domain)	83
3.18 Rebuilt random harmonic at $t = 1$ (spatial domain).....	83
3.19 63 station values SSE	88
3.20 200 station values SSE.....	90
3.21 450 station values SSE.....	91
3.22 SSE of $Q = 10$ Processed	91
3.23 Getting the same value for different Q truncations when forcing the same harmonic of $Q = 0$ on the standard deviation	92

LIST OF FIGURES

	PAGE
2.1 Entity Relationship Diagram (ERD) for GriddedClimateData database	13
2.2 A screenshot of 4DVD menu and sub-menus.	14
2.3 ERD of dataset databases	14
2.4 4DVD menu of options.....	21
2.5 The 4DVD menu selections to visualize the January 2016 temprature data.....	26
2.6 Spatial map of the January 2016's global SATA with respect to the 1971- 2000 mean climatology.	27
2.7 The SST time series at (2.5°S, 132.5°W), a location on the Nino 3.4 region over the eastern tropical Pacific.	29
2.8 “Climate2” main view 3D	34
2.9 Side menu opened.....	35
2.10 Dataset selection window	36
2.11 Colormaps window	37
2.12 “Climate2” main view 2D	38
3.1 Error for harmonic example	51
3.2 Error for function example	51
3.3 Harmonic Plot Q=49	52
3.4 Function Plot Q=49	53
3.5 Cumulative Variance Explained for Non-Normalized Data.	69
3.6 Cumulative Variance Explained for Normalized Data.....	70
3.7 Variance Explained for Normalized Data.	71
3.8 “Hole Punch” algorithm	71
3.9 Numerical Validation - Random time series	84
3.10 Numerical Validation - Random time series, random points	85
3.11 63 stations various Q_s baseline 1960-1990 timeseries	87
3.12 200 stations various Q_s baseline 1960-1990 timeseries	89
3.13 Station Counts	93
3.14 Long term station locations	94

3.15	Long term station locations	94
3.16	Long term station locations	95
3.17	Long term station locations	95
3.18	Optimal Weighting compared to NCEP	97
3.19	Optimal Weighting compared to GISS Surface Temperature Analysis	98
3.20	Optimal Weighting compared to GISS Surface Temperature Analysis - Cor- rected baseline	99
3.21	Spatial Reconstruction 1980 $Q = 102$ side view	100
3.22	Spatial Reconstruction 1980 $Q = 102$ top view	101
3.23	Spatial Reconstruction 1980 $Q = 30$ side view	102
3.24	Spatial Reconstruction 1980 $Q = 30$ top view	103

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND

It's common for academic research to request time to run computationally intensive workloads on a mainframe or a super-computer cluster. Often, you need to prove that your research is of merit, kiss feet, and hope you're allocated enough time to run your whole process. When you finally are given time to run your computations, you have to be prepared in the middle of the night to start it and a single bug in your code, that never occurred during its testing phase, could spell disaster forcing you to repeat this whole process again. There must be a better way.

Thankfully, there is and it's called cloud computing. With services such as Amazon Web Services you can request just enough compute resources that you need and a bug in your code won't be the end of you. Just take the time to fix it, and re-submit your request when you're sure it's been corrected, you pick the schedule and someone else's "higher priority" tasks won't bump you.

The business world has enthusiastically embraced this new technology and it's completely changed how they operate. Companies with limited means no longer need to build out a huge network of servers to process their latest and greatest system. They can easily scale both up and down depending on demand and they'll never waste idle CPU cycles. Everyone else is embracing this and academia should too.

The thought of using cloud computing wasn't the initial intent of my PhD research when I first started several years ago. The original plan was to continue my master's research on optimal interpolation using spherical harmonics but as time progressed I noticed room for improvement in many existing solutions, not all completely related to my original intentions.

This dissertation is divided into the following two main chapters:

- Visualization and Fast Delivery of Climate Data (Chapter 2): Creation of a web application using modern technology for browsing, viewing, and downloading large climate datasets.
- Optimal Interpolation via Distributed Processing (Chapter 3): Using cloud computing to calculate "Optimal Global Average Annual Mean Temperature Estimation Using Station Data and Spherical Harmonics"

Both the visualization and optimal interpolation chapters can stand on their own and can be considered as separate projects. The Optimal Interpolation chapter introduces performing the

calculations using cloud computing resources. Numerous choices were consciously made during the course of the optimal interpolation code development to facilitate it so it can work successfully on and off cloud computing resources.

There is also an appendix that is Java oriented in nature. The purpose of it is to aid someone interested in getting into Java from a numerics background. I personally started coding in Matlab first and eventually got pulled into the Java way of doing things. This was partly out of necessity because my work outside of this research led me down this path.

1.2 OUTLINE OF THE DISSERTATION

1.2.1 Visualization and Fast Delivery of Climate Data

The visualization portion of this research was born out of frustrations of sifting through the numerous climate datasets that are available. Anytime a new dataset needed to be inspected, I had to go through laborious and manual processes to get the data into a form my personally written tools could deal with. If I wanted to plot a single date out of a particular dataset, I was forced to download the entire dataset. There was no easy way to “browse” what a dataset looked like. While I was able to find some datasets that did offer some plotting services, they were rudimentary at best and hard to glean the desired insights from the data. There really is no “standard” for visualization of data prior to downloading.

There are services like Earth System Grid Federation (ESGF) that try to make this process easier, but their primary function is as a search engine to find available datasets and to aid in their download, not to visually browse through them. The service is rather impressive and supports more than 700,000 datasets from universities and both national and international laboratories. These datasets range in sizes from a few small kilobytes to petabytes, for a total combined size of over 5 petabytes of data [52]. ESGF is also the only gateway to access data from the “Coupled Model Inter-comparison Project” (CMIP) phase 5, which by itself takes over 4.2 petabytes of data.

The problem is, that’s just too much data to go through and inspect. What if a user simply wants to browse these available products and see what they are; it’s simply impossible. Since I struggled with this process, others must be equally frustrated.

Thus work began on finding a way to showcase the results put out by the San Diego State’s Climate Informatics Lab. Out of the many different datasets the informatics lab had released, we began with “The Weather History Time Machine” as this was a very popular dataset. It was my belief that having datasets available on-line to view interactively and download would be a great way to share and spread the lab’s research. Sharing CSV files is one thing, but in reality that’s not enough. Researchers don’t have time to download a dataset, write code to be able to parse it for use with their systems, only to finally view the data and realize it’s completely useless

for their research needs. There are also so many datasets available that they wouldn't know which one, if any, would be useful to their research. Downloading every single dataset until they find "the one" is both time and cost prohibitive.

Whatever system we ended up using, it must allow users to easily browse through the numerous datasets that the Climate Informatics Lab has. Users need a visual representation of the data and they must have the ability to download parts of the dataset because they may only be interested in a particular date or region. The system must also be fast enough to run on relatively inexpensive hardware to keep costs low, giving the data provider the option of either hosting the hardware themselves or uploading it into the cloud through a service hosting provider.

This project began by searching to see if software already existed that met our needs. National Oceanic and Atmospheric Administration (NOAA) uses the Web-based Reanalyses Intercomparison Tools (WRIT) [47]. WRIT allows users to generate charts and download sub-samples of datasets through their website. However, these generated charts offer no interaction and do not have much flexibility. Science on a Sphere (SOS) (<http://sos.noaa.gov/>) is another NOAA application. This is an educational tool to visually show Earth system science to people of all ages, but it provides no data services and requires a large download be installed, which is something we wanted to avoid.

Cameron Beccario's Earth (<https://earth.nullschool.net/>) can be visually appealing, but again does not offer useful data support and functions such as visualizing historical time series at different spatial locations and downloading the corresponding data. It also doesn't take advantage of a user's graphics card; exploiting the graphics hardware, could potentially be a game changer in this space.

NOAA View Global Explorer (<http://www.nnvl.noaa.gov/view/globaldata.html>) is mainly an image viewer. It uses static images that are download dynamically as users zoom into different regions (similar to Google maps). It can load datasets with high counts of grid points since it's using images instead of actual numerical data. While this route is feasible, it would require building a separate downloading system, as it would be impossible to accurately get values from an image file. A time series component also wouldn't work with this method; a separate system would need to be written for that as well.

There is also an abundance of paid GIS systems that are web based; all of these were omitted because they were not cost effective. Open source or bust!

Since no off-the-shelf systems fit the requirements building a system from scratch was necessary. Basing the system on open source technologies offers flexibility in sharing with other research groups who wish to implement it in their own processes.

With that decided, several key features were needed in new the system:

- Give a user the ability to look at both time series and spatial data.

- Contain enough visual appeal so users would have no need for other software to generating graphics from the data.
- Create an easy to use interface that could switch through numerous datasets (the interface should also be relatively easy to customize if another group wishes to use the system).
- Above all, the system must be fast. If the system isn't responsive, people won't bother using it.

With these set goals in mind work began on SDSU's 4D Visual Delivery (4DVD) web application, and after a lot of hard work we now have completed working prototypes of the system available in two flavors:

- Pure HTML and JavaScript: <http://climate.mrsharky.com>
- Angular 2: <http://climate2.mrsharky.com>

Through SDSU, we have also dual licensed this work. There is an open source variant available for non-commercial purposes using the [GNU General Public License v3](#). This copyleft license will ensure that any work derived from here will remain completely open source. We are also going to have a commercial license available for those that want to use it for commercial purposes. Both the HTML (climate) and Angular 2 (climate2) versions of the project can be downloaded from [GitHub](#). This system, and hopefully other systems derived from it, will make both the spreading and assimilation of climate data easier.

We are already beginning to see the significance of this work; a few parties whom Dr. Shen has showed the work, have already expressed interest. The scope of the project has also increased since it's unveiling due to this interest and to unforeseen expansions in complexity of the project. The current interested parties are:

- [NOAA-CREST](#) - Would like to use it to aid in educating people about climate.
- [The Collider](#) - A nonprofit innovation center focused on catalyzing market-driven climate solutions.
- Harvard Medical School - We have begun a collaboration with them to refactor the web application to work for brain imaging. While I myself have not done the work directly, I outlined needed to be done, mentored the students working on it, and helped them with issues as they came up.

A paper related to this work has been submitted, accepted, and published in "Advances in Data Science and Adaptive Analysis" [39]. The first sections of this chapter contains the results from that paper and it is followed with the additions that have been done on the system since it's publication.

1.2.2 Optimal Interpolation via Distributed Processing

The second part of the research is a continuation of the work that I began for my master's thesis at SDSU. That original work ran into a number of issues that, at the time, I didn't know how to address. While we will go into the details of the problem in Chapter 3, I will give a brief synopsis of what the road blocks were.

The premise of what we're trying to do is use a derived dataset that contains gridded information for the whole globe to figure out the climates spherical harmonic frequencies. Using these frequencies we will figure out how to optimally weigh individual weather stations to determine a global trend for the annual temperature. The original solution, worked on during my masters, and the optimally weighing logic failed to give consistent results. The process was also computationally expensive causing us to reduce the scope of what needed to be processed.

Since the overall theme for this dissertation proposal is to not only show that valuable research can be done in the cloud, I wanted to continue tackling this problem with the aim of using distributed computing to solve these sorts of issues.

Computationally expensive problems shouldn't deter research; we will prove that it doesn't have to. A research team with limited funds and hardware can still compete with "The Big Boys" and on tough problems if they organize their problems wisely.

The problem is that the original code base was written in Matlab, a language that isn't the friendliest when one needs to run their calculations in the cloud. The logic needed to be switched to a language that would facilitate the usage of distributed computing system resources more easily, specifically resources available via cloud computing providers. After a few sideways hops, we settled on Java because it is supported by Apache Spark (a distributed computing system that can run on Amazon Web Services). Spark is heavily used in the Data Science community and is available on numerous cloud providers. Using this system, researchers will be able to do as many calculations as they want in "the cloud"; this opens up massive computing potential to them.

Another benefit of porting the code to Java would be that the process would no longer require Matlab. Removing Matlab from the equation alleviates the following difficulties:

- Sharing the code requires that other users also have Matlab and subsequently, all the extra add-on libraries that we used are also required.
- Matlab isn't cheap; a completely open source solution would be easier for other researchers to use, change, etc...
- With an eye to distributed computing, Matlab won't work on an Apache Spark cluster. While not part of the research of this section, one of the overall goals for our research is to eventually process everything on a distributed cluster in the cloud.

- It isn't easy to modify the Matlab code so any dataset can work with it. Matlab is good for quick prototyping and one offs, but not good at creating processes that need to run on different datasets.

The above issues are why we decided that development of the code couldn't continue in Matlab. We would first need to port the completed code out of Matlab and continue it's development in a new language that comes with a lot less baggage. Java was selected because it is both portable and, as was mentioned earlier, can be used with Apache Spark. While it is true there are other more popular languages that can be used in conjunction with Spark, Java affords us a great deal of flexibility, and has close to native support in Spark.

Before we actually settled on Java, I originally began porting the code to Python but was left unimpressed with the language. Only after we had actually switched to Java did we discover that Python did not have access to all of Spark's features, features that were required to implement this custom process to work successfully. If I would have stuck with Python, I eventually would have been unable to move forward. I would have needed to port my code again to either Java or Scala to have access to those features. In the end, I was lucky that I found Python's indentation syntax horrendous, as it was one of the main reason's that I decided to ditch the language entirely.

During the porting and subsequent development process I ran into numerous obstacles, one of which actually was the choice of using Java. Java is great, but since the language is compiled to machine code at run-time, it isn't nearly as fast as if we ported the code over to C/C++. Java isn't necessarily a language associated with numerics, so in many cases libraries didn't exist for some of the calculations we needed to do. Other times we had issues with the available libraries for certain numerical tasks that had to be accomplished.

The final code base is an extremely numerically complex piece of machinery. It deals with numerics such as: spherical harmonics, eigenvalues, vectors, matrix factorizations, angular momentum coupling, etc. Showing that it's possible to do these calculations successfully through a framework like Spark is proof that practically anything could be coded up to take advantage of cloud computing.

Using services like Amazon Web Services (AWS) that provide Apache Spark as a services comes with other advantages too. NOAAs has already started a Big Data Initiative, where they have made their datasets available freely available on AWS. One could theoretically never do any computations on their own systems and never waste bandwidth on a single dataset download. All of the calculations could be done in the cloud as the data is already available there!

Furthermore, results from said computations could be automatically made available to a web based application, like the one we already developed in chapter 2, when the computations are finished. Researchers could code and run their manipulation logic in the cloud, have them placed in a web application and then browse through their results without downloading a single large file.

The advantage of processing this particular work on a cluster is that there is a computationally expensive step that runs too slowly on a single workstation. Even when exploiting multi-threading of the application, some calculations still take too long to calculate reasonably. Thankfully, these computations can be decoupled from one another and can be run in parallel which makes distributing the computations to a cluster feasible.

The goal of all of this cloud computing is to produce two different outputs:

- A time series representation of the global anomaly based off limited available station data.
- A time series gridded “map” of this global anomaly from this same limited set of available station data.

The first goal can potentially be generated by a single work station, but calculating it via Spark can speed the process up. The second goal requires the use of distributed processing as there are too many computations the higher the resolution we wish to generate.

The original work done during my masters was able to partially solve the first goal, but those results didn’t agree with previous research, but the second goal was impossible to solve at the time, but using modern tools now it is.

As a first step, we must first be confident that the Matlab, Java, and subsequent Spark code get the same results before moving forward. Once we know the code is correct we can then work towards completing the optimization calculations.

Like the other web based application, this code also has be open sourced. The reason to open source the code is to allow others to integrate their own research code to work with Apache Spark. While some people may be interested in running our code on their own datasets, sharing a template that many can use and learn from will be valuable in many different fields of research. The entire Java library can be found on github under my personal directory <https://github.com/dafrenchyman/mrsharky>. Just like our web application project, this code is open source as well. This will allow other researchers to be able to apply our code to their projects as well.

Our work here shows it is possible to use software such as Apache Spark as a viable alternative to requesting time on a mainframe; we hope others will follow in our footsteps. Since we’ve open sourced the code, we will be providing researchers with a “cut and paste” example that they can use as a starting point for their own Apache Spark distributed computing problems.

Finally, I have one very honest admission to make: While it is definitely possible to run these computations in the cloud through a service like AWS, I however did not. During the course of the summer of 2017, my employer, Experian, tasked me with building out a small Spark cluster because we were doing too much small processing on AWS. This was an effort to move some of the smaller processing tasks back to our internal systems as we happened to have some machines

on hand. Experian graciously allowed me to use this small cluster in my research when it was sitting idle and not in use.

CHAPTER 2

VISUALIZATION AND FAST DELIVERY OF CLIMATE DATA

2.1 INTRODUCTION

While work has been done to make climate data available to users, it is not easy to read the large data files and to get desired information from them. An average user, who only knows about CSV and excel files, cannot easily use these downloaded datasets, which are in netCDF or other formats. Although professional users of climate data may have the ability to read the data, they have to download the entire dataset first and may have to convert the data into a new format familiar to the users. Only then, can the data be visualized. Average users would be unable to do this and would need a system in which they can browse through available data visually, identify special features (such as El Nino's influence on land surface air temperature anomalies over the U.S.), and then download the desired data. This kind of convenient, fast, and focused access to big climate data is a global need to many different people ranging from scientists to high school students, from business corporations to government agencies, and from tourists to farmers. The traditional digital data download is like searching for a needle in a haystack, requiring professional climate and computer scientists to download entire datasets and then manually sifting through them searching for desired information. In this era of cloud computing, a new delivery mechanism is required that enables users to visually scan through data products available before downloading only a subset of the data that is actually needed. Instead of binging on the whole "data buffet" at once, the buffet should allow users to only "eat" what they actually want. This could significantly save users' time and the bandwidth of the "data buffet."

Some visualization tools, such as NOAA's Web-based Reanalyses Intercomparison Tools (WRIT) [47], have been developed and have their own merits. WRIT allows users to generate charts and download sub-samples of datasets through its website. While WRIT has the merit of allowing users to do some custom calculations, its generated graphics offer no interaction and do not have much flexibility. Another example is NOAA Science on a Sphere (SOS) <http://sos.noaa.gov/>. This is an educational tool to visually show Earth system science to people of all ages, but it provides no data services and requires a large download be installed. Cameron Beccario's Earth (<https://earth.nullschool.net/>), can be visually appealing, but does not offer useful data support. It lacks functions such as visualizing historical time series at different spatial locations and downloading the corresponding data. It also does not take advantage of a user's

graphics card. Exploiting the graphics hardware potential could be a game changer in big-data delivery.

The purpose of this paper is to describe a new system for 4-Dimensional Visual Delivery (4DVD) of big climate data, developed at San Diego State University (SDSU) in collaboration with NOAA. The 4DVD system offers four services: (i) visually appealing maps, (ii) animations, (iii) time series, and (iv) CSV data download. In this way, the easy-to-use 4DVD technology can help bring commonly used climate data, such as the NOAA National Centers for Environmental Prediction (NCEP)'s surface air temperature, precipitation, and atmospheric pressure into classrooms, museums, and households. It can also serve as an interface of a climate history time machine (CHTM) [46] that allows people to visit the past climate and weather, back to 1900 or earlier. The 4DVD system makes it very easy to browse through many different datasets visually, download parts of the data in an easy to use CSV format, and have a visually beautiful and appealing tool that people can use to generate charts for a wide variety of applications.

The next section (Section 2) will describe the general idea and methodology of the system. Section 3 includes 4DVD technology description and user guidance. Some application examples are presented in Section 4. Conclusions and discussion are in Section 5.

2.2 METHODOLOGY

2.2.1 General idea

4DVD integrates many different pieces of technologies together into a single cohesive package. While the individual parts themselves are not all that complicated, the challenge is to get these discrete parts to work together efficiently as one cohesive system that runs quickly. The system's design and development satisfy the following criteria:

- The system can allow users to view spatial maps and time series, to animate, and to download data;
- The system should be visually appealing so that users would normally have no need for other software to generating graphics for their applications;
- The interface should be easy to use and customize;
- Adding additional datasets to the database should be an easy task;
- The system must be fast since an unresponsive system would not attract users.

The whole system was prototyped on an Ubuntu 16.04 LTS Server virtual machine running on top of Xen Server [10]. Virtualization was a key in aiding the application development and made it easy to rapidly prototype out changes and to start the system over again from scratch when large changes had to be implemented. If desired, the complete application has also been

written as a docker container, for an even easier deployment [32]. Docker is not only a great tool for deployment, but also has aids in making research reproducible [9].

The deployed system has basically three parts:

- (a). User's Web Browser: Client side code is run on the users machine to take work off of the server;
- (b). Web Server: Http service to provide the browser with everything needed to run the application; and
- (c). Database: The data storage for all the datasets to be viewed and to be added.

A simplified order of execution is that when a user comes to the website, the web server will serve up both static content from files and dynamic content generated from the database. Any and all requests for data submitted by the user's web browser must pass through the web server. The web server then goes to and communicates with the database to gather whatever is requested, puts it into a readable format, and then serves it back to the web browser. Although the web server has to do all of this work, it is relatively easy work, and the real workhorse in this whole system is the database. A lot of data needs to be looked up for a single data request and then needs to be given back to the web server in a short time. If the data is not returned fast enough, the user's experience will be degraded and the user may accidentally make multiple requests through the database, which could backlog the system. Thus, it is critical for the database to run very fast and quickly return back the necessary data to the web server.

Even if the database is designed correctly, it is at risk of becoming a bottleneck if the system is heavily accessed. However, for scaling purposes, a distributed file system [59] can be applied with multiple database servers, to be provided by NOAA or other service agencies, with the same data. This standard big data technology can be easily setup to alleviate this risk. A system can also be setup to have different physical databases hosting different contents, putting more popular datasets on more dedicated hardware.

In order to keep the web server and database as lean as possible, all of the actual data processing is done by the individual user's browser. This enables 4DVD to use the user's local computing resources. All the data conversions to the graphical preparation and visualizations are done at this level. This also makes it possible to have many different options available for the user to manipulate the data and change the visual style without putting any added pressure on the web application. The website also uses WebGL for its graphical abilities so an OpenGL capable graphics card is a requirement to view the website correctly [31].

The following subsections will describe 4DVD's dissected parts of applications, starting from the back (i.e., the database) and working forward to the end user.

2.2.2 Database

The system's current database runs on MariaDB [60] with tables optimized to make it fast enough to handle larger datasets while avoiding solid state drives. Every attempt has been made to try and keep data footprint as small as possible. One benefit of 4DVD is that the database and website are decoupled to the point where it would be easy to swap out MariaDB with another database of owner's choice.

MariaDB is being run with multiple databases on one server. The main database GriddedClimateData keeps a lookup of all the available data products available. The most important part of this main database is a single stored procedure (g_Dataset) that is used to get a list of all the available datasets. This stored procedure requires no inputs and gives back a table of information about each of the available dataset. This "lookup" contains various pieces of summary and reference information, and more importantly gives the name of a separate database and location for each of the individual datasets available. When the website first loads, it loads this lookup information populating the website with information about each of the available datasets.

2.2.2.1 G_DATASETS

This procedure is called by the web server via PHP by the following command:

`http://website/php/g_datasets.php` and returns a json file similar to the following:

```
Datasets = {
  Dataset_ID: [ 1 , 2 , ... ] ,
  Name: [ "Menu Item a | Menu Item 1 | ... | Dataset 1" , "Menu
    Item b | Menu Item 2 | ... | Dataset 2" , ... ] ,
  DatabaseStore: [ "Database1" , "Database2" , ... ] ,
  OriginalLocation: [ "ftp :// locationOfData1.nc" , "ftp ://
    locationOfData2.nc" , ... ] ,
  StartDate: [ "2010-01-01" , "2001-01-01" , ... ] ,
  EndDate: [ "2015-12-01" , "2009-12-01" , ... ] ,
  Units: [ "degK" , "J/kg" , ... ] ,
  DefaultLevel: [ "Default level 1" , "Default level 1" , ... ]
};
```

The layout for the GriddedClimateData database can be found in the Entity Relationship Diagram (ERD) shown in Fig. 2.1.

In the current implementation, the location to download the datasets is stored in the Dataset table; this information drives the automated loading script that will be discussed later on. The Name field in the Dataset table is the location of where the dataset will appear in the

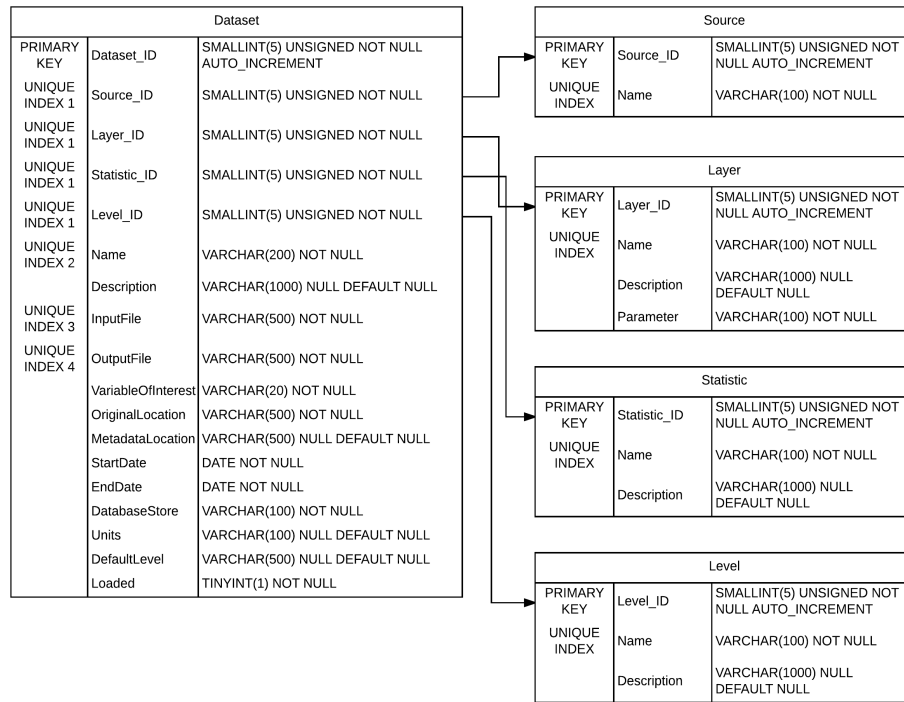


Figure 2.1. Entity Relationship Diagram (ERD) for GriddedClimateData database

website's loading menu. Here, one of the NOAA datasets is used to illustrate how this works. The entry looks like the following:

NOAA Twentieth Century Reanalysis (V2c) | Pressure Level | Non-Gaussian | Air Temperature | Monthly Mean (1000–10mb)

The “|” (pipe) characters are used as delimiters to specify under which menu or sub-menu the dataset will appear. So, the above example would be under a menu cascade: NOAA Twentieth Century Reanalysis (V2c) ⇒ Pressure Level ⇒ Non-Gaussian ⇒ Air Temperature ⇒ Monthly Mean (1000-10mb).

This is intentionally nested to allow many individual datasets to be available within the same menu. Each sub-section can also have many different versions. For the above example, the NOAA | Pressure | Non-Gaussian | Air Temperature section has four different choices:

- Long Term Monthly Mean (1000-10mb)
- Long Term Monthly Mean Ensemble spread (1000-10mb)
- Monthly Mean (1000-10mb)
- Monthly Mean Ensemble Spread (1000-10mb)

Sub-sectioning makes it possible to have many datasets available on a single menu (see Fig. 2.2). All of the parsing and menu creation takes place on the user's end. This makes it easy to add new datasets to the system without ever altering any code or the web server. The new datasets just need to be added into the database properly, and a lookup placed into this Dataset table. Absolutely no code change is required to add in a new dataset.



Figure 2.2. A screenshot of 4DVD menu and sub-menus.

When a particular dataset is selected out of the menu, the system then uses the DatabaseStore field from the Dataset table. This field gives the location of another database that holds the actual data for that particular chosen dataset. Each available dataset in the system is required to be in its own database. This was intentionally done to make it easier to add more datasets over time and to keep a uniform setup for all the datasets.

When the system switches to one of these “sub” databases, it gains access to all the raw data shown in Fig. 2.3. This figure is the ERD representation of what these tables look like in our current implementation.

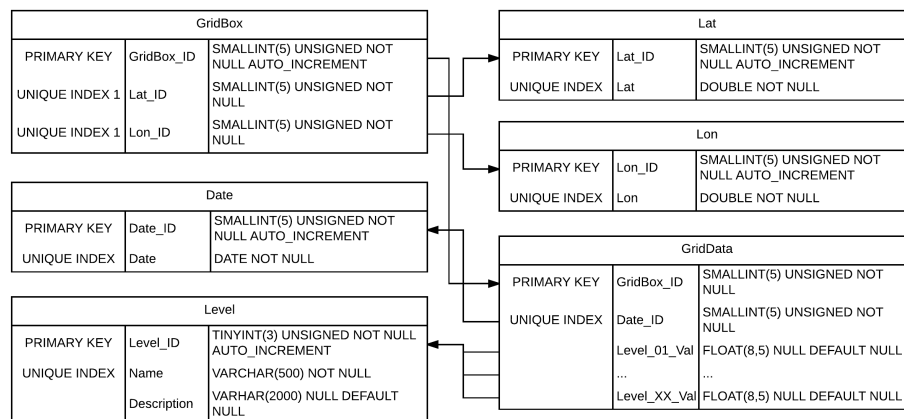


Figure 2.3. ERD of dataset databases

For these “sub” databases to work correctly in 4DVD they must contain three different stored procedures: `g_GetLevel`, `g_GetGridData`, and `g_GetTimeseriesData`. Each is described below.

2.2.2.2 G_GETLEVEL

The first of these stored procedures provides a list of the different “levels” (or layers) available for a dataset. Although not all datasets have different levels, for the ones that do, this stored procedure gives back that list along with their accompanying `Level_ID`. The system always sets the `Level_ID` = 1 the first time a new dataset is selected.

This stored procedure is accessed via the website from the following link http://website/php/g_GetLevel.php?dbname=DATABASE, replacing DATABASE with the desired DatabaseStore received from the previous PHP results. This returns a json similar to the following:

```
Levels = {
    Level_ID: [ "1", "2", ... ],
    Name: [ "1000.0 millibar", "950.0 millibar", ... ]
};
```

2.2.2.3 G_GETGRIDDATA

The second stored procedure serves the global gridded data back to the user with the help of two inputs: a `Level_ID`, which would have been gathered from the above, and a date. The available dates are all provided by the main “lookup” stored procedure mentioned previously, and the data is restricted to monthly readings only for the time being. The future versions of 4DVD will reach daily and hourly level. This procedure returns a table containing an ID for each grid point (or grid box), the latitude and longitude of each grid point, and finally the numerical value at that location.

The website requests this information from the following link:

```
http://website/php/g_GetGridData.php?dbname=DATABASE&date=DATE&
level=LEVELID
```

DATABASE is again replaced with the DatabaseStore value. DATE should be replaced with a date within the ranges of the `StartDate` and `EndDate` that came from the main “lookup.” Finally, LEVELID should be replaced with the `Level_ID` received from the previous stored procedure. The procedure will return something similar to the following, except for every grid point on the globe:

```
RawData = {
    GridBox_ID: [ 1, 2, 3, ... ],
```

```

    Lat: [−90, −90, −90,...],
    Lon: [−178, −176, −174,...],
    Value: [266.79843, 266.79843, 266.79843,...]
};

```

2.2.2.4 G_GETTIMESERIESDATA

The third stored procedure gives the time series data of a particular location and requires only one new input: the GridBox_ID provided by the previous procedure. This procedure simply returns the ordered dates and values associated with an individual GridBox_ID.

The system requests the time series data via the following link:

```

http://website/php/g_GetTimeseriesData.php?dbname=DATABASE&
gridboxId=GRIDBOXID&level=LEVELID.

```

The DATABASE field is the same as before, while the GRIDBOXID is the GridBox_ID just mentioned; the LEVELID is the same value used previously. The return will look like

```

RawTimeseriesData = {
    Date: ["2000-01-01", "2000-02-01",...],
    Value: [248.58548, 246.27234,    ]
}

```

Although different datasets could potentially be stored in a single database, and even in a single table, all the datasets in the 4DVD have been split up for the purpose of delivering the data as fast as possible. Separating the datasets into different databases helps keep the indexes from growing too large, and also helps organize the data in a distributed way that is faster to look up an individual dataset.

As was mentioned previously, keeping every dataset in its own database has another major advantage: it is easier for the 4DVD system to add other datasets later; hence the 4DVD system is expandable.

In summary, the system is employing four optimizations to make the data quickly accessible:

- Data for individual datasets resides in their own databases. This has the added advantages of keeping the tables smaller and making lookups faster.
- Tables have been indexed on the inputs required to look up specific data.
- Data has been organized in the tables in a way to make the lookup via indices even faster.
- Stored procedures have been forced to use the best index for the desired lookup.

For the last item, MariaDB does not always choose the best query optimization when performing a lookup [21]. While 4DVD was in development, this problem plagued the system until the realization that stored procedures could be configured to force the application to use a specified index, instead of letting the system choose it. This forcing sped up the data requests by a healthy margin.

The above optimizations have their side effects, including:

- The MariaDB server is populated with a large number of databases. This can seem disorganized when browsing through the list of available databases on the server.
- Creating indexes for faster lookups requires additional storage space.
- Storing the data in a database is not as size-efficient as using something like NetCDF [41]. When running, our datasets take up a lot more room in a database.
- To increase the speed for the two different data lookup types requires the data be duplicated with one table for the global map and one for the time series.

A dataset that starts out around 2GB in a NetCDF file may take up approximately 9GB of space when the above processes are being performed. This almost five fold increase is large. The data size increase is one of the reasons the prototyped 4DVD system does not include the larger daily or hourly datasets. While it is possible to put very large datasets into the system, it may require dedicating a whole hard drive for just one dataset. With current and future hard drive prices, this may not be a deal breaker, but for prototyping it is not ideal.

While the size increase is unfortunate, this is what makes the system fast. When a user uses the website, they are not downloading the entire dataset, but only looking at one layer and one date at a time. So, the data transferred for each individual result is minor.

Due to the sometimes large dataset sizes, 4DVD is not the best platform to use for downloading the entirety of a dataset. The system would instead prefer referring the user back to the original datasets' NetCDF files. NetCDF may not be great for a web based application, but it is excellent at making it possible to store a lot of data in a small package.

A distinguished advantage of the 4DVD system is its flexibility and adaptive capability to new technologies. The data does not need to be stored in the database in the same way as described above. The system could be modified to completely forgo a database if desired. Since the data that is ultimately being used is a json file, the system could easily be modified to use something else as long as the output result is still a json file. If a particular data product would be better served by storing it in a different way, the system could be modified to do so. One could also stick with using the database system, but decide to change the structure as long as the necessary stored procedures are created and behave in the same way as originally designed. The data can thus be setup in any way one desires.

2.2.3 Web Server

The web server is a fairly basic Apache web server with the addition of PHP and its database access libraries. The whole website is served up by completely static html and js (JavaScript) files. The only dynamic files are the PHP queries to the database. PHP also only has access to the limited number of stored procedures mentioned previously. PHP's job is simply to convert the dynamic data into a json file that can easily be loaded up by the end user's browsing session. The server also is setup to compress all text data to be transferred to the user, and hence to save bandwidth. All the data the system generates is text data except for the few images that are used in the system.

In the end, 4DVD does not have anything special about its web server in the system's development pipeline. In the future, PHP may be replaced with node.js as this could potentially yield higher efficiency [30]. In the meantime though, the system performs very fast.

2.2.4 Browser

Most of the work that the web application does actually takes place on the end user's browser. Once the raw data has been loaded from the database to the user's workstation, the data is organized and prepared locally. All of the mathematics and matrix graphical processing is done by the browser with the aid of JavaScript and WebGL shaders. There are many reasons why this is beneficial to the application; a few are listed below.

- Less processing needs to be done on the server side, so the server spends more time handing out data rather than doing complex processing of its own.
- WebGL is considerably faster than processing graphics via canvas or other browser drawing methods. By offloading the graphics processing to the video card the visual quality can be improved so that movements and animations are smoother (i.e., rotating the globe), and the data loads faster.
- New features can be added to the system without updates to the database. Only static files delivered by the web server would require an update. These changes can be easily prototyped without breaking the existing system.

For example, every single option available in the "Graphical Options" section is processed locally. While options like the adding of additional coastal resolutions does load a new file from the website, all of the processing of correctly displaying a file is done by the browser. All graphical settings, like re-organizing the data to switch from a 3D view to a 2D one, is processed in JavaScript. There is no need to load new data from the database, or to communicate with the server during any of these steps.

This has made it extremely easy to add more “bells and whistles” to the application over time without wasting time repopulating or changing the database, which is by far the lengthiest process.

The application thus provides the user a robust menu system that can easily be augmented with more features. The current main applications menu is laid out as follows:

- Dataset Selection: Where you can select one of the available datasets.
- Layer Selection: If a data set is multi-layered, the different layers available for selection are here.
- Time Selection: Years and months available to view in the dataset.
- Graphical Options
 - View Options: Options relating to how the map is viewed.
 - Additional Lines: Additional data that can be overlaid on the map.
 - Color Map: Many different color schemes for the data being displayed.

2.3 RESULTS: 4DVD SYSTEM FUNCTIONS AND OPTIONS

The menu system is where everything can be controlled from on the website. Here a user would first begin by selecting a dataset in the “Dataset Selection” section. This would then load the first date and first Level_ID of the dataset to the screen on the main window. After selecting a dataset, the “Layer Selection” section will also be populated with the available layers that the dataset may have. If there is only one layer available, then the section will be empty. The “Time Selection” section is also populated with the ranges of dates available. There is a slider that can be used to navigate the different years and a drop down menu to select individual months during the available period.

When a user is done with the menu, they can hide it by clicking the “Menu” button. With the menu shrunk, a user could then focus their full attention to the websites’ main view: the climate map or globe of the Earth. The mouse wheel can be used to zoom in and out; clicking, holding, and dragging can move to different locations around the globe.

To reopen the menu, click on the “Menu” button again. A user can then change the way the main graphical window is displayed in the “Graphical Options” section of the menu. The first version of 4DVD has 12 different graphical settings for manipulation.

The first section of “Graphical Options” gives the user access to two sub-sections that change the look of how the data is presented. These two sections are the “View Options” and the “Additional Lines” options.

2.3.1 View Options: Bump Mapping

Next come all of the “Graphical Options.” The first section, “Globe,” contains options relating to changing the geometry and imagery that are displayed on the globe. The first option, “Bump Mapping,” uses one of the amazing features that WebGL has at its disposal: normal maps. This technique is commonly used in video games to get more detail out of an object with limited polygons. Bump mapping simulates the bumps and groves on the surface of an object by perturbing how light reflects off an object’s surface [6]. While the globe being generated in the main window is a sphere with a smooth surface, bump mapping enables one to simulate mountains and valleys on its surface without modifying its shape. This is done via an OpenGL pixel shader that runs directly on the graphical hardware and re-calculates how light bounces off every pixel on the screen in a given frame. The system uses a “normal map” ([16]) texture on the sphere that tells the rays of light which direction they should bounce off depending on the angle that the light comes in at [20].

If instead this information needed to be conveyed by actually perturbing the polygons on the sphere, the sphere would contain 33,554,432 points, equal to 8192×4096 pixels in the texture, which is much higher than the amount of data in the climate datasets that have at most 18,048 points. This 18,048 is the max number of grid points used in the 20th Century Reanalysis [11] that is available in the system.

The bump mapping technique allows the dataset to stay at its original resolution but still allows the system to add an impressive visual effect that also conveys some important information. Since these calculations take place on the graphical hardware, this visual flair comes at zero performance loss.

The system currently has options to view either the height information of just the land, or both the land and the oceans (“Land & Bathymetry”). This extra information can help identify how datasets might be affected by surrounding mountain ranges, such as cold temperature on high mountains, orographic precipitation, and blocking of moisture transport.

However, since this system will mostly be used by climate researchers, this “visual flair” can be disabled if the user wants a purer view of data values. Simply changing the option to “none” disables the “visual flair” and removes the light effect so that the colors match the actual climate data value according to the color legend bar.

2.3.2 View Options: Globe

There are three different projections that can be selected for displaying the data. The first two options, “3D” and “3D-Ortho”, change the projection matrix used for transforming the points from 3D space to the screen. The plain “3D” projection is a simple matrix with a perspective while the “3D-Ortho” projection is an orthographic projection matrix.

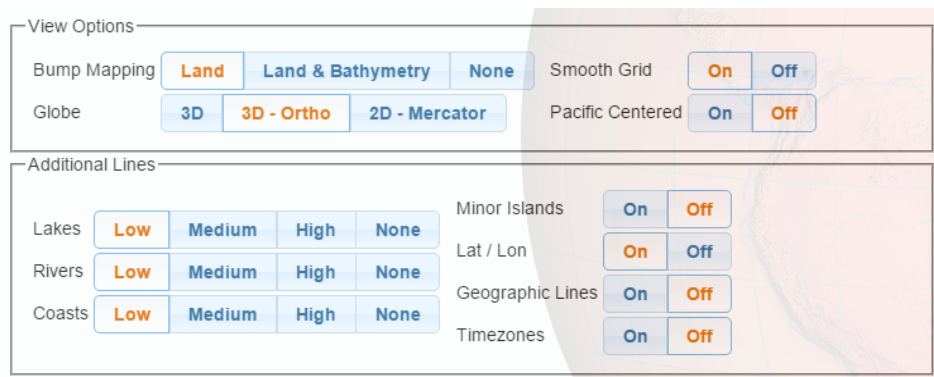


Figure 2.4. 4DVD menu of options.

When working with 3D graphics, multiple transformations need to be done on objects before the data is actually visible on the screen [61].

Of the three projection options available: the third option, "2D Mercator," is not an actual 3D projection matrix. Instead, the data from the globe must be rebuilt out as a flat rectangle and the camera is positioned above this rectangle. Since the object is completely flat relative to the camera, either of the 3D projections could be picked for this last option.

All of the math behind switching projections, re-processing the grid points onto a flat surface instead of on a sphere are taken care of within the browser in JavaScript.

2.3.3 View Options: Smooth Grid

By default, the "Smooth Grid" option is on, and the colors corresponding to different values of a dataset are a smooth transition from one grid point to the next. In the climate science community, the climate data is actually stored in a grid, with each grid box having one value, and are discontinuous as a spatial field, but the real climate parameters are spatially continuous, except for some extreme weather shocks. The system smooths the transitions from grid box to grid box (i.e., point to point) by interpolating the values between the points. This is all done on the graphics hardware by very fast OpenGL pixel shader code. No extra processing needs to take place in the browser to get these values, so it does not affect the system's performance.

Because this view is not the "real" view one would get from the data, a user may wish to turn off this "Smooth Grid" interpolation function. By disabling the function, the data will switch from the smoothed transitioned colors to the gridded data's boxy or pixelated feel. While it may not look as nice, the boxy feel has better fidelity to the real climate science data, but perhaps not to the true climate field.

2.3.4 View Options: Pacific Centered

The “Pacific Centered” option shifts the middle of the map from longitude 0 to longitude 180°E. At the same time, it also changes the longitude coordinates from -180 to 180, to all positive values from 0 to 360. Visually it only affects the “2D- Mercator” view of the map, but it will also change the longitude coordinates of the individual data downloaded from the website.

Surprisingly, the calculation for doing this is not as straightforward as one would believe. In “3D” views, not much changes, but in the “Mercator” projection, everything falls apart because of the line data that is also placed on the globe. A lot of code had to be written to adjust all the linear data (like the coasts and rivers) when this option is enabled. Otherwise, lines that cross over the longitude = 0.0 location would then cut across the image instead of stopping.

Since the solution was done programmatically, the system can easily change this processing to center the “2D Mercator” view on any longitude coordinate desired. The current version of 4DVD only gives two options: “Pacific Centered off” centering at 0, or “Pacific Centered on” centering it at 180, but an option can easily be added to center the global map on any longitude line.

2.3.5 View Options: Additional Lines

These options enables a view of coastline, rivers, and others on the globe. The website (www.naturelearthdata.com) had a great deal of open source shapefiles for not only coastline data, but shape files containing various pieces of information that could be overlayed on 4DVD. Yet, a shapefile is not in itself compatible for use on a website. The Shapefile file format is an open geospatial vector data format used for displaying points, polylines and polygons, but they store data in a binary format [14], which is not easily compatible with web applications. Thus, the data needs to be converted from the shapefile data to something that is readily available for use with a website.

To facilitate this, a simple R program was written to re-encode the latitude and longitude coordinates into a json file. The big difference between the json format and the original format is that all data is treated as line data. Polygons that are used to generate data such as lakes are usually filled with a color but the new format assume everything is a line. During the conversion, to signal a line segment ends, the file simply inserts a NULL value to signal the application to stop drawing.

Unfortunately, there is a limit to the number of elements that can be drawn on the screen with WebGL per a single draw call. This required the use of some logic to detect if this threshold would be crossed during drawing the next line segment. If so, the code stops the creation of the object, and creates a new 3D object to store the values in. So, while the coasts look like they could be one object, there are actually several draw calls that are required to draw the whole thing up.

The alternative to this would have been to draw every single line segment (line segments end with that NULL value mentioned previously), but this actually slowed the graphical processing down too much as the draw calls are instantiated by the JavaScript code. Since JavaScript is much slower at processing than the WebGL code, the fewer draw calls the better.

All of these “Additional lines” are used so the user has the ability to overlay different shapes on the globe. Coasts, lakes, rivers, islands, time-zones and geographic lines are currently available.

Lakes, rivers, and coasts have three different levels of resolution. By default the website loads the lowest resolution for these shapes, but the medium and high resolution are available by choice. Depending on the user’s internet connection, the higher resolution lakes, rivers, and coastal lines may take some time to download. These lines can also be removed entirely from the map by clicking the “None” button.

2.3.6 View Options: Color Maps

4DVD makes it easy to add any desired color scheme for the map. Many color maps have already been implemented. Color maps essentially have two types: non-centered and zero-centered. Non-centered color maps are ones that evenly pass through the gradient of colors. This color scheme is appropriate for displaying precipitation data which is always positive. The zero-centered color scheme has a clear zero transition from positive to negative and is ideal for viewing anomalies. Whichever color is in the middle of a gradient is assigned the zero value. Values higher and lower than zero will get re-scaled to fit the dataset correctly, but are not necessarily evenly spaced between the hues. Obviously, zero-centered color maps only work well with diverging color sets.

The vast majority of the color maps available come from colorbrewer [19] through the “Color Brewer 2.0” option. All the Color Brewer 2.0’s sequential (both multi-hue and single-hue) and diverging color schemes have been converted to work for the 4DVD system.

For users more accustomed to Matlab, the 4DVD system offers color schemes very similar to Matlab’s under the menu button “Matlab” in the “Color Map” choices. If none of the already available color schemes satisfies a user, the 4DVD system can be augmented so that the user can easily add his own.

2.3.7 Time series display: digital data for the climate history of a given location

While the main focus of this website is the large globe view of climate data for a user, another equally important view is available: climate history time series. Regardless of the view in use, a single click on the map or globe will select the grid point closest to the cursor. A “Grid Box Info” menu box then appears at the lower left corner of the display window. This box displays the

latitude and longitude coordinates of the location selected and will also give the value of the dataset at that location. Upon clicking on the “View Timeseries” button in that box, the time series data will be loaded over the screen.

The computing logic behind finding the closest grid point is fairly straightforward and given below:

- Calculate the click projection converting the location where the user clicked to a line in the scene that goes through the globe.
- Loop through all the triangles that make up the globe to find out which ones the line went through. There are three different cases here:
 - The line went through two triangles. Calculate the actual coordinate of where the line intersects both triangles and pick the point that is closest to the camera.
 - The line went through one triangle. Calculate the actual coordinate of where the line intersects with the triangle.
 - If the line did not pass through any triangles (user clicked off of the globe), then no coordinate is found.
- With the coordinates from above, simply loop through all the grid points and pick the point that has the smallest distance to the coordinates calculated. The closest one is the grid point that will be used for selecting the time series data.

When the time series chart loads onto the screen, there are options to load multiple layers onto the same chart if the dataset has multiple layers. So, if someone wants to compare the differences between 1000.0 millibar and 500.0 millibar, one simply needs to select both of them in the “Layers” section.

Other features available include the ability to zoom in on selected dates (scroll wheel). While zoomed in, one can move through different times (click, hold, drag, release). If one clicks on a specific dates value, the globe view will update in the background and switch to that selected date. By passing the mouse over a date, the value for each of the loaded layers will be printed to screen, so one can numerically compare the values for given dates, instead of eyeballing it from the graph.

2.3.8 Data downloads: digital data for a map or time series

The last important feature of 4DVD is that any data visible on the screen can be downloaded as a CSV file. When a dataset and date are selected, all of the data used to generate the visual reside on the local machine. Since the data is already there, exporting the data to CSV (or potentially any other format) becomes trivial. Both the global gridded data and the time series data are available for download on their respected screens.

2.3.9 Adding Example Data - NetCDF

Although any global gridded dataset can be added to this system, the whole system has been prototyped with NOAA's 20th Century Reanalysis (v2c) datasets [11]. Almost all of the monthly datasets from 20th Century have been implemented and added into this version of the 4DVD system. Since the data is originally stored in the NetCDF file format, a Java library was used to aid in converting everything out of the NetCDF file format [51].

A bash script was written that looped through and downloaded all the available 20th Century Reanalysis datasets. It then converted the data out of NetCDF, populated each dataset into its own individual database, and automatically set up the databases with stored procedures, indexes, and keys to make the database usable. Finally, it updated the "lookup" database with information that the dataset had been loaded and ready for the web applications described above.

The main reason why 20th Century ReAnalysis was chosen is that if the loading and processing of so many datasets in the NetCDF format could be completely automated, then our 4DVD system can be applied to other datasets, such as the Intergovernmental Panel on Climate Change (IPCC) data. We know that we could just as easily develop computing processes to convert the other data over to web applications.

2.4 AN APPLICATION EXAMPLE: VISUAL DELIVERY OF THE NOAAGLOBALTEMP DATASET

An example usage for 4DVD is to show the spatiotemporal variations of El Nino based on observed global temperature data since 1880. Figure 2.5 shows the menu choice to visualize the global surface air temperature anomalies (SATA) of the latest strong El Nino month, January 2016, according to the NOAA Merged Land Ocean Global Surface Temperature Analysis (NOAAGlobalTemp) ([23, 48, 54]). This is the most updated global land ocean SATA dataset with $5^\circ \times 5^\circ$ spatial resolution and monthly temporal resolution from January 1880 to present. The source data for 4DVD is from the NOAA National Centers for Environmental Information (NCEI) (<https://www.ncdc.noaa.gov/data-access/marineocean-data/noaa-global-surface-temperature-noaaglobaltemp>)

The following options have been picked for this example:

- Dataset Selection: NOAA
- Time Selection: 2016, January
- Graphic Options: Bump Mapping "Land", Smooth Grid "On", Globe "2D-Mercator" (to view the entire world with a glance), Pacific Centered "On" (to best visualize El Nino whose strongest and most robust signal is warm surface surface temperature (SST) anomalies over the eastern tropical Pacific).
- Additional Lines and Color Map: The default 4DVD options were selected.



Figure 2.5. The 4DVD menu selections to visualize the January 2016 temprature data.

The above option choices yielded the spatial map for January 2016 shown in Fig. 2.6. The black area areas indicate regions of missing data, which are mostly the polar regions above 70°N or 70°S. In this figure, the total missing data area is less than 10% of the Earth surface. The January 2016 El Nino signal was 5°C warm anomalies over the eastern tropical Pacific, which created warm anomalies over the western North America. This is expected for a robust El Nino impact signal ([42] and agrees with the NOAA Climate Prediction Center (CPC) website of “Global ENSO Temperature and Precipitation” (<http://www.cpc.ncep.noaa.gov/products/precip/CWlink/ENSO/regressions/>).

The warm anomalies over the southeastern China and Japan were also typical El Nino impacts. The unusual warm anomalies of more than 10°C over Alaska and Russia north of 6-°N were due to the persistent Arctic warming since 1998, and were probably not due to El Nino. The unusually cold anomalies over Mongolia and the western Europe might be due to the persistent

Arctic high pressure and perturbed Rossby waves and jet flows, corresponding to the negative phase of Arctic Oscillation (AO) [50]. The AO index was -1.449 for January 2016, according to NOAA CPC (http://www.cpc.ncep.noaa.gov/products/precip/CWlink/daily_ao_index/ao.shtml). This high Arctic pressure weakens the westerlies in the upper atmosphere, pushes cold air to more southern latitudes, leading to cold winters (about -3 or 4°C anomalies) over northern Europe, northern Asia, and northern United States. For January 2016, the warm SST over the tropical Pacific with a large amplitude of 5 or 6°C warm anomalies still overwhelms cold air from the AO in the western U.S. Thus, the January 2016's western U.S. still had warm anomalies, but the northeastern U.S. had cold anomalies. Brutal Arctic cold outbreaks were persistent over northeastern U.S. in the 2015-16 winter. A report from Accuweather on January 19, 2016 was entitled “Harshest arctic air of season grips midwestern, eastern U.S.”

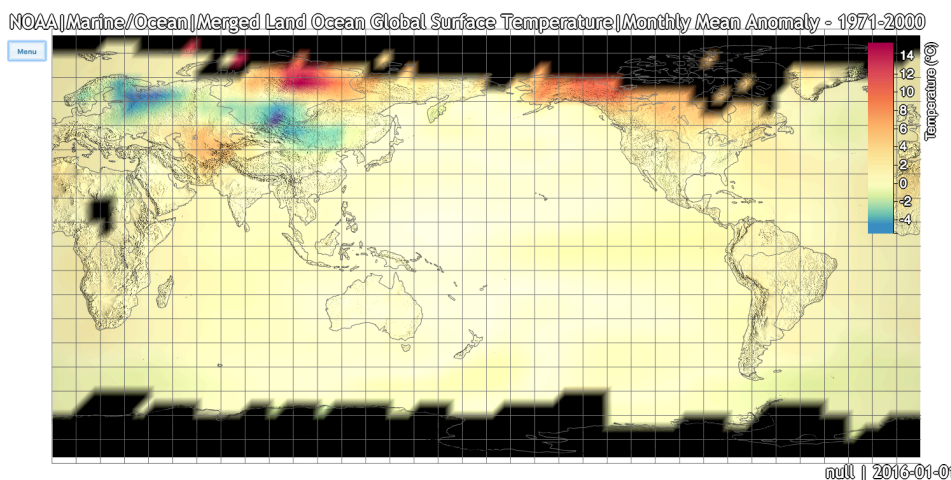


Figure 2.6. Spatial map of the January 2016's global SATA with respect to the 1971-2000 mean climatology.

To download the temperature data for this map, one can click the “Download Options” button at the bottom of the menu box, then a black “Download Gridded Data” box appears. Clicking on this box downloads the temperature data on the grid into a CSV file. The data looks like below:

Latitude	Longitude	Value
-67.5	2.5	-0.3709
-67.5	7.5	-0.3747
-67.5	12.5	null
.....		

Here, “null” indicates missing value.

4DVD can easily show the history of El Nino. A click at the eastern tropical Pacific could generate a time series, indicated by a dialogue box “View Timeseries” at the lower left corner. Clicking on this “View Timeseries” box prompts a time series to appear on and overlaps with the map window, as shown by Fig. 2.7. The major El Nino’s SST anomalies are clearly reflected on the figure, such as Jan 2016, October 1997, January 1983, November 1972, January 1941, and January 1889. These results are compared well with CPC’s Southern Oscillation Index (SOI) data from 1951 based on the standardized Tahiti and Darwin sea level pressure differences:

<http://www.cpc.ncep.noaa.gov/data/indices/soi> Figure 7 also agrees well with the center of heat index (CHI) amplitude defined as the area-weighted average SST over the tropical Pacific strip (5°S- 5°N) [15].

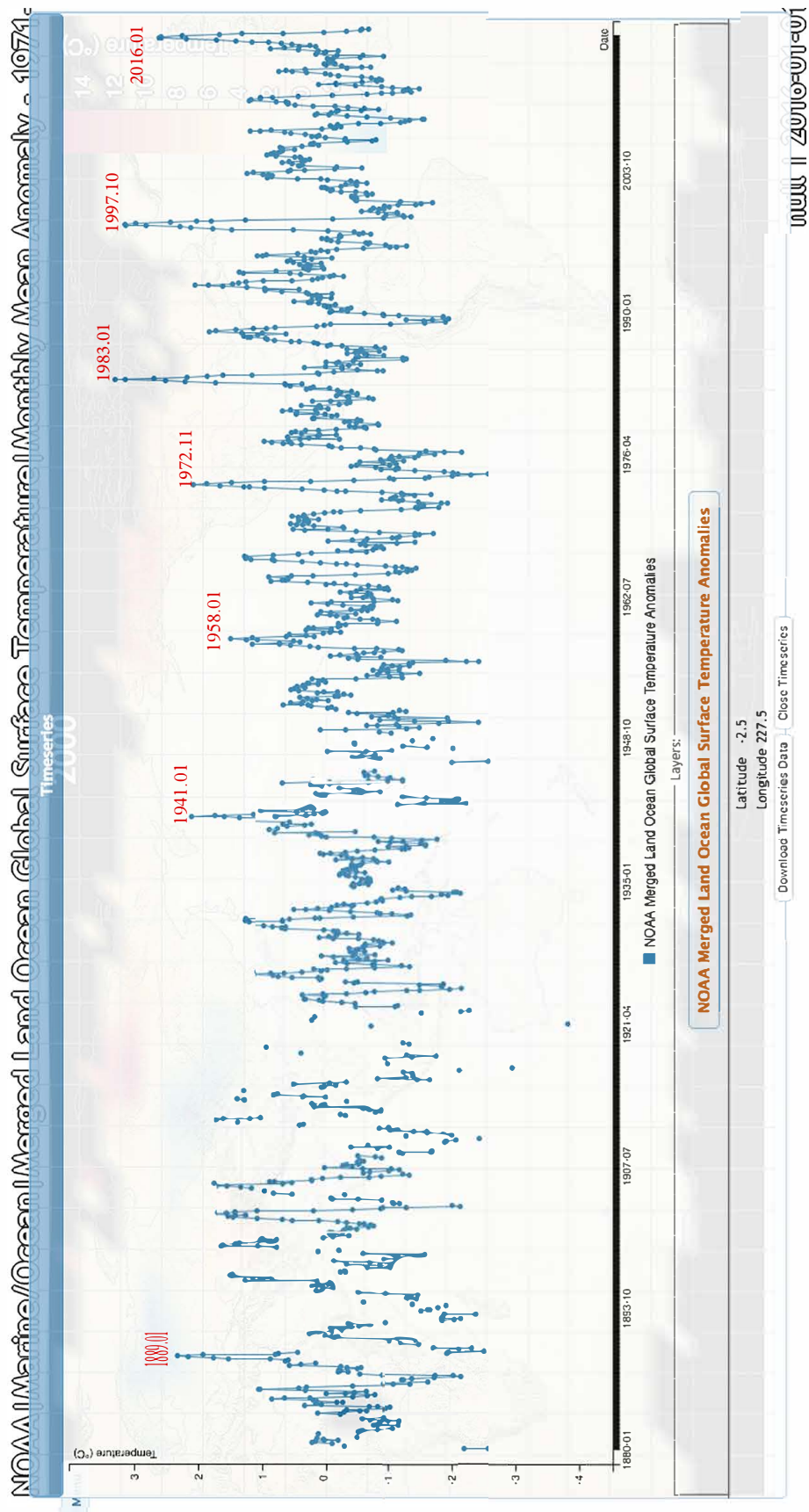


Figure 2.7. The SST time series at (2.5°S, 132.5°W), a location on the Nino 3.4 region over the eastern tropical Pacific.

To make more detailed analysis, such as studying the spectral properties and quasi-periodicity of the El Nino events, one can download the digital data of the time series by clicking the “Download Timeseries Data” button. The CSV file of the time series is downloaded to a user’s computer. The data looks like below:

Name	Date	Value
NOAA	1880-01	-2.5131
NOAA	1880-02	-2.1397
NOAA	1880-03	null
.....		

One can close the time series window by clicking the “Close Timeseries” button at the bottom of the time series window. The map window appears again.

2.5 CONCLUSIONS FROM THE ORIGINAL PAPER

We have described the basic ideas, development procedures, functions, options, and application examples of 4DVD: a very fast visual delivery system for big climate data using distributed database technology. The 4DVD system consists of three parts: database, web server, and web browser. The 4DVD database part uses MariaDB database technology and includes a main database GriddedClimateData and multiple sub-databases. Each climate dataset, such as NOAAGlobalTemp, uses a sub-database. The main database keeps a lookup of all the available datasets and hosts a single stored procedure (g_Dataset). There are many advantages of using a sub-database for each dataset. A few are listed below:

This design enables an optimal use of distributed file service technology.

It makes the visual data delivery fast.

It allows an easy expansion of 4DVD for an additional dataset by a simple addition of a sub-database.

It minimizes the maintenance work of the 4DVD system.

The 4DVD’s web server part is a standard http service based on the basic Apache web server with the addition of PHP and its database access libraries.

The 4DVD’s web browser part optimally utilizes the computing resources on a user’s computer by running the calculations on the user side, according to the user’s request of images, animation, and digital data. This design reduces the computing work from the backend server, and thus helps sustain the server’s stability and accessibility by a large number of users.

Various kinds of visualization and data download options have been designed to utilize many updated open source technologies, such as the display of Earth’s surface topography and

shaders. The system has also been designed to facilitate the future addition of more datasets, such as the massive amount of data from the Coupled Model Intercomparison Project (CMIP) Version 5 <http://cmip-pcmdi.llnl.gov/cmip5/>. Any dataset added is a simple addition of a new database specifically dedicated to the dataset. Thus, additional datasets will not slow down the 4DVD system.

The application examples have clearly shown that the 4DVD system can easily deliver climate data to classrooms, museums, libraries, and households of ordinary people. The system's spectacular image and animation are appealing to students and general users. This is an important technology that makes the climate data accessible to not only climate scientists, but also to the general public. 4DVD could facilitate the climate education in schools and will help keep the general public, business sectors, and government informed with reliable climate information so that they can make informed decisions on climate and environment. In this way, 4DVD can prepare the societies around the world to better meet the climate variation challenges of the 21st century.

2.6 ADDITIONS SINCE PUBLICATION

2.6.1 Adding Example Data - NetCDF

Since the paper's publication, work has continued on the project. The first addition was the implementation of a system to "automagically" populate NetCDF data into the system. Since the original prototype had been done using NOAA's 20th Century Reanalysis (v2c) datasets [11], we already had some experience with reading the NetCDF file format via its official Java library [51].

A bash script was written that looped through and downloaded all the available 20th Century Reanalysis data sets. It then converted the data out of NetCDF, populated each dataset into its own individual database, and automatically set up the databases with stored procedures, indexes, and keys to make the database usable. Finally, it updated the "lookup" database with information that the dataset had been loaded and was made available to the web applications described above. It can currently process all of the 312 datasets available. Unfortunately, the entirety of this data isn't available on our demonstration system (<http://climate.mrsharky.com/>) as we have limited resources available on this system.

This was done as a demonstration that the loading and processing of the NetCDF format could be completely automated. This implies that other teams can just as easily write their own processes to convert their data over to this web application, but if the data happened to be in a NetCDF format, they could just use our implementation.

2.6.2 New features since the publication

Since publication, we've also implemented many new features into the system. One issue with the original system was that resizing the window after it's already been initialized would not resize the globe. Switching between window sizes would force the user to completely reload the web application. This is both time consuming on the user's behalf, and depending on the users cache settings could force a complete reload of all the data on the website: something that needs to be avoided. To remedy this, the system was updated to check for window size changes which would then force a change on the size of the WebGL content as well.

Another feature that has been added is the ability to store the database on a different server than the one hosting the actual website. This was done because thanks to Dr. Shen's contacts at the [National Climatic Data Center \(NCDC\)](#), we may be able to store our databases on their servers. This required us to completely separate the database from the web server allowing them to run on different systems. This is a security requirement as NCDC doesn't want anyone messing around on their systems. However, since the database would usually only store static data, we could simply provide them a script to populate it with the necessary data and then we would be finished.

One major improvement built into the system was a way to "save" a current users state. For example, in section 2.4 (which was part of the original paper) an example is given directing a user to click on various options and settings. To be able to save a users state this information was embedded into the website's URL. This way, a user who happens to like a particular view with certain enabled options can now share that view with someone else, by simply cutting and pasting the complete URL. As an example, the following [link](#) will zoom in on the island of Guadeloupe where I was born, on the year and month of my birth for a particular dataset. The scene also loads some particular graphical settings to make this small island visible and adds some minor islands around it as well. Please note that it can take a while to load and parse the high definition coastal outline datasets that are being loaded, so it may take a few moments for everything to load correctly.

This one seemingly minor feature was actually very difficult to implement correctly because everything was coded in pure HTML and JavaScript. This difficulty ended up being one of the major deciding reasons to refactor the application to work under an actual web framework.

2.6.3 A new framework - Angular 2

At this point, this research was believed to be completed since a paper was submitted, accepted, published and everything was wrapping up. However, adding new features became quite difficult as minor change requests were getting harder to facilitate. While overhauling the user interface, we ran into many problems related to the lack of a framework for this project.

While the original interface is functional, it's wasn't user friendly or attractive and takes up too much of the screen's real estate. Attempting to update and overhaul it, proved to be extremely difficult. This contradicted one of our original goals: that it should be easy to redo the user interface. We hadn't attempted the rewrite it at the time and didn't believe an overhaul would be difficult.

Since everything is so tightly coupled, minor changes in one section of the code can completely break other almost unrelated sections. In the end writing the code in pure HTML and JavaScript was problematic. It did make prototyping the system from scratch fast but as the code base grew, in concept and scope, things become difficult to manage.

At first we attempted to just clean up and re-organize the existing JavaScript and HTML code. It did help for a time, but soon we were again running into difficulties. After a lot more research, we decided that a complete rewrite into Angular 2 would be the only viable solution. [Angular](#) was chosen for a few reasons:

- JavaScript can be hard to debug and work with. It is not [strongly typed](#) which has been a source of pain since the beginning of the project. Angular 2.0 instead uses [TypeScript](#) which is a strongly typed language that compiles straight to JavaScript.
- JavaScript is similar to C in how it works. There is no sense of object inheritance or other structures associated with object oriented languages. When a code base gets large, it becomes difficult to manage. Angular does support object oriented programming and so, if a project is coded correctly, the code can be [loosely coupled](#) making it easy to swap out parts. For example, it would make it easy to rewrite the user interface without affecting other parts of the code base which is one of our new goals.
- Angular is in a way also similar to coding in Java with [Maven](#). You can easily add in packages that others have written to you project. This keeps you from reinventing the wheel every time something is needed that someone else may already done and thoroughly tested.
- Angular also integrates directly with the HTML code that is written, removing the disconnect that exists between HTML and JavaScript which makes it easy to add components to a site dynamically.
- Angular will eventually allow us to scale the website to different screen sizes: smart phones for example.

Switching over to Angular has also allowed us to use the Model-View-Controller (MVC [26]) design pattern which makes for a separation of tasks among different portions of the code. This will also aid in the user interface design as it will be easy to swap different interfaces in and out with one another.

The new Angular version of the website (<http://climate2.mrsharky.com>) is now much easier to manage. Interestingly, while a large amount of time was spent overhauling the front end, the back end database and Php portions of the code remained completely intact. Once the data is

loaded, there isn't a need to change anything on the database front. During this code conversion process the database portion of this work has remained the same.

This shows that the database design decisions made early in the process were resilient and shows the use of separations of principles as both versions of the front-end communicate with the exact same back end. This proves that many different front end interfaces could be hosted by many different institutions such as national agencies, laboratories, and universities. These institutions could do whatever they want with their design, look, or even the type of visualization they want to give users the ability to see. Even with these differences, they could all access the same back end server to get whatever data they need.

2.6.3.1 AN UPDATED INTERFACE

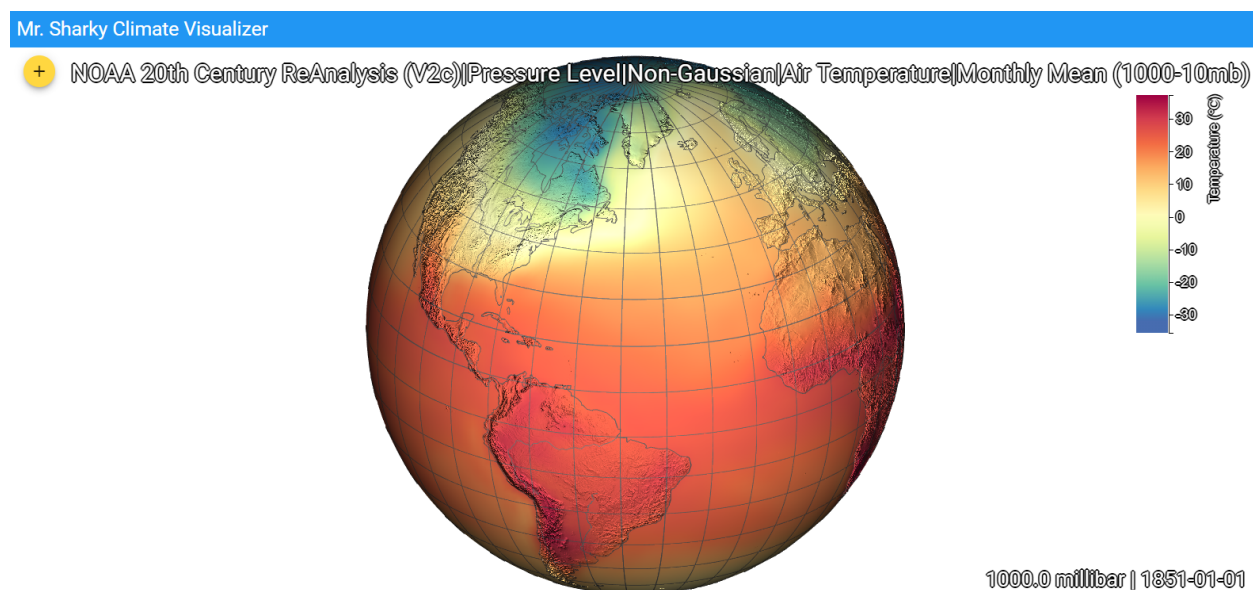


Figure 2.8. “Climate2” main view 3D

Angular 2 has made it much easier to generate a new and clean interface (Figure 2.8). A lot of clutter has been eliminated by completely removing the old menu system as it took up a significant amount of the usable space. The menu is now hidden by default, and is accessed by clicking on the plus symbol () in the top left corner of the interface. When the menu is selected, the map screen dims and is shifted to the right as the menu moves in from the left side (Figure 2.9). This allows the user to visually see the changes being made on the right side while making the menu “pop” so the user knows it’s in focus. A simple click back on the map view exits the menu and brings the map back into focus.

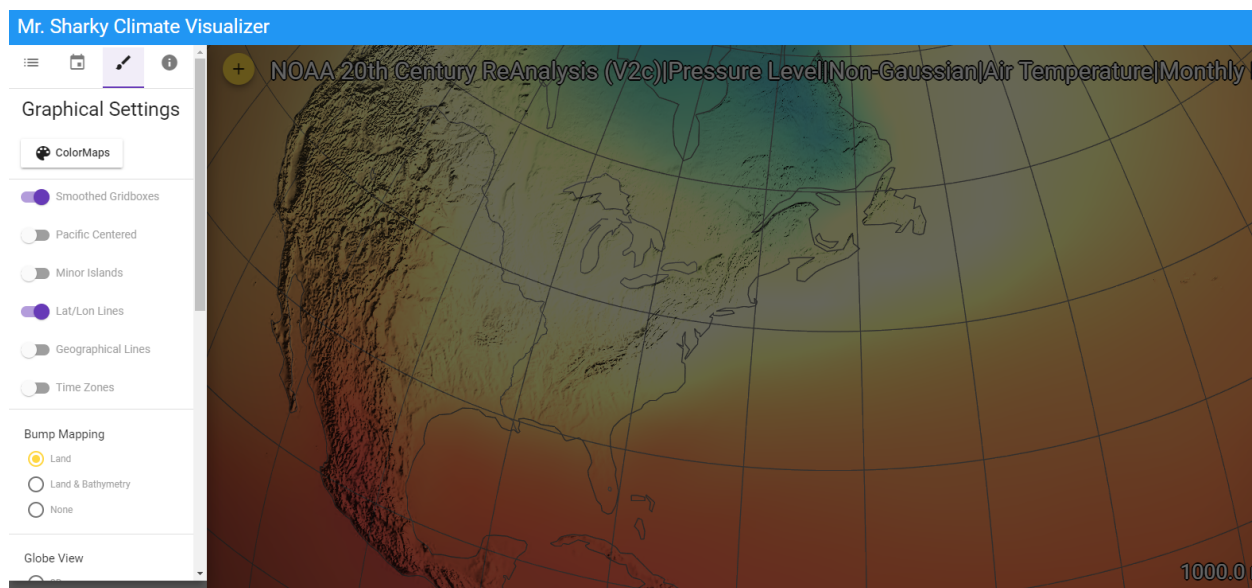


Figure 2.9. Side menu opened

The menu's options have been re-arranged to be able to fit in this new side menu. There are 4 picture tabs on top corresponding to the following menu options:

-  Dataset Settings
-  Date
-  Graphical Settings
-  Information

2.6.3.2 NEW CLEANER MENUS

The “Dataset Settings” tab () is where a user can select one of the datasets available on the server. By clicking on the “Datasets” button a new window fade in with a list of the available datasets that can be selected (Figure 2.10). The nesting logic that was used in the original interface has been removed as it was difficult to navigate through the available options. It was difficult because precision was required to properly hover from one menu to the next; an accidental slip would force the user to start over again. Nesting will eventually be re-introduced into the system, but via clicks instead of hovers. When selecting a dataset from the this menu it's available layers will be made available for selection on this same tab.

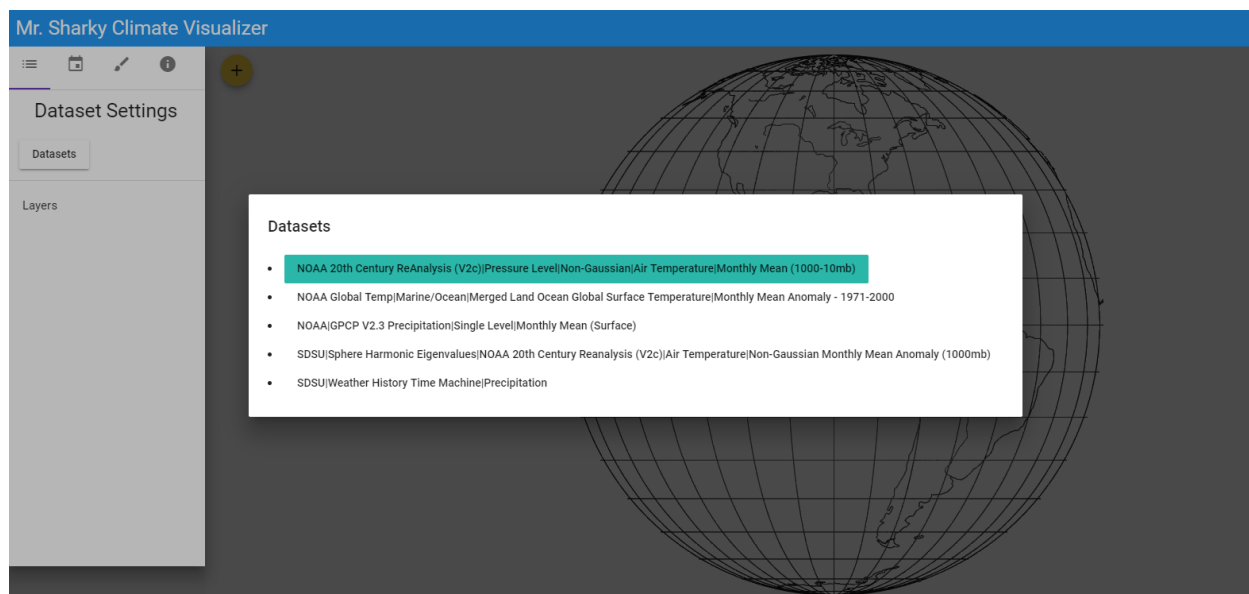


Figure 2.10. Dataset selection window

The next tab over is the “Date” tab (📅). Here a user can select whichever months and years are available to the dataset via a few options. There’s a slider that can be used to “slide” to a specific year and a box beneath it where a user can either toggle up or down a year, or simply type one up. The last option beneath that is a drop down to select from available months. These options are identical to the ones in the original version of the website, with the exception that the animate ability has been removed for the time being.

The “Graphical Settings” tab (🔧) is next and contains all of the different toggles and switches we had on the original interface, except everything takes up less screen space and is more visually appealing. At the top, you’ll find the “ColorMaps” button which will load up all of the available color maps in a new overlay window 2.11. The nesting logic from the old system has been removed here as well, so for the time being, it’s just a large list to scroll through. Unlike the previous version the color map previews are now bigger and easier to see. While the system is completely functional, there still is room for improvement. As an example, the inverted and non-inverted version of the color maps are available as different color maps, and the non-centered vs zero-centered versions of the color maps are available as well. This quadruples some of the different available color schemes as they fit under all of those categories. They’re the same colors, just either inverted or stretched so the middle color is always at zero. It would be much better to have toggles for these features if a particular color map supported it.

Color Maps Available

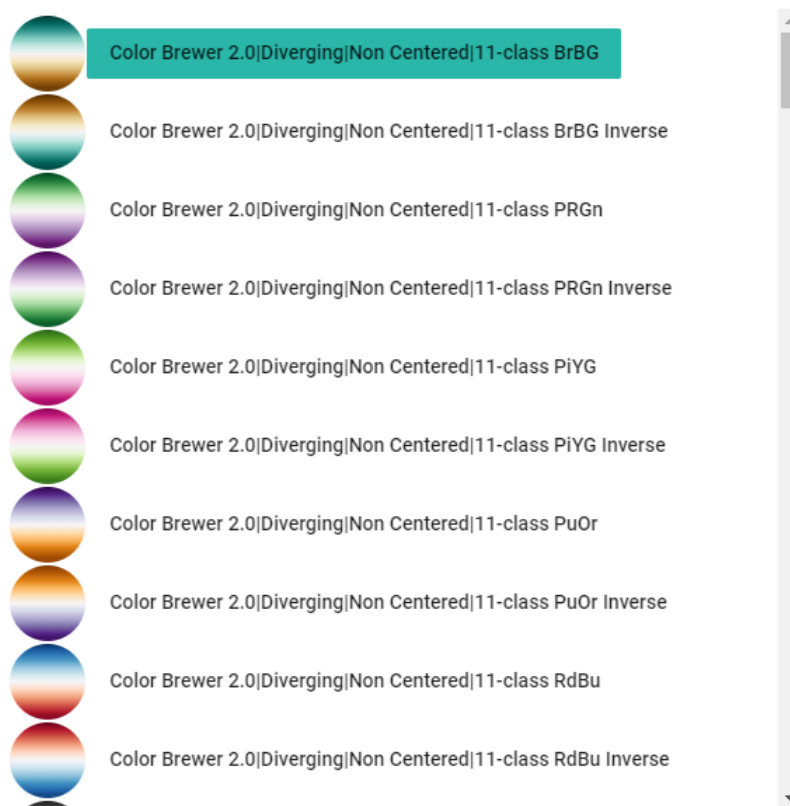


Figure 2.11. Colormaps window

Continuing with this tab we find all the same switches and radio buttons that were in the original graphical settings. None of their functionality has changed and more details on them individually can be found starting around section 2.3.1.

The last tab, “Information” (), is where the link to download the current view is available. This space will eventually be used to house information about the current dataset such as it’s original source and a link to that source. This will also be a good place for copyright info and information about the project.

Finally, the time series data is still available by single clicking on a particular grid point and selecting the “Timeseries” button that comes up in the lower left (Figure 2.12). We are also now using a native Angular 2 library for the generation of the time series. The library was very easy to implement but it is not as fast or user friendly as the previous version. There may be a need to eventually port the old time series code over as it could handle large amounts of data with relative ease compared to the new version. The work never ends...

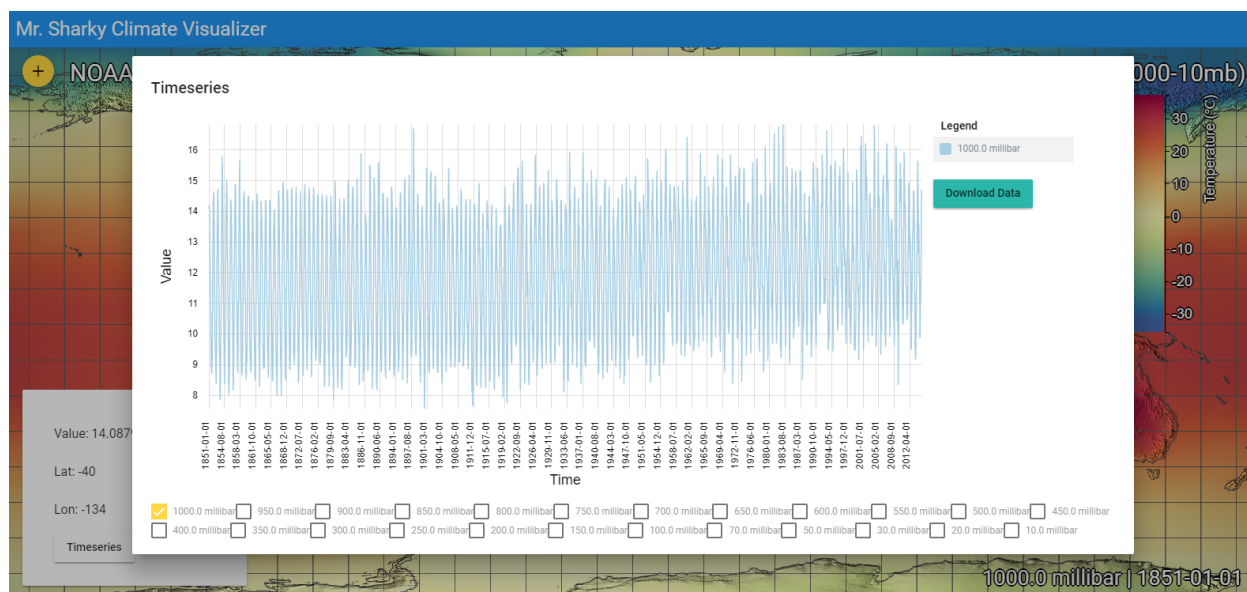


Figure 2.12. “Climate2” main view 2D

2.7 CONCLUSIONS AND FUTURE WORK

The project is now dual licensed through San Diego State University and we already have a few parties interested in using the work in it’s current state. This research also led to a collaboration between us and Dr. Albert Yang and [Dr. Chung-Kang Peng](#) of Harvard Medical School. Students under my tutelage have been working on re-implementing the work here to be used with brain imaging data instead of climate.

All in all, this side project that was born out of frustration grew into an interesting final product. I intend to continue to aid in the development of the product even as my research at San Diego State nears an official end.

As has been previously mentioned, there are still other improvements that could be implemented. The system currently provides the latitude and longitude coordinates for every grid point. Since these are “grid” points, the data footprint could be lowered by simply providing the unique latitude points and unique longitude points. JavaScript logic could then be added on the client’s side to give each grid point it’s correct coordinates. This could potentially save some bandwidth between the server and the end user and allow the system to support more users. It could also potentially save some processing time because the json file output would be smaller than it currently is.

We’d also like to get one large daily dataset implemented in the system. We believe the database can handle it, but may need some smart partitioning scheme configured. This update would also require some UI changes to support a calendar to select the dates, but it would give the user even more options.

Finally, more improvements could be done by switching out PHP with node.js as this could potentially give us a slight boost in performance [30].

CHAPTER 3

OPTIMAL INTERPOLATION VIA DISTRIBUTED PROCESSING

3.1 BACKGROUND

“Reanalysis” is the process of reprocessing archived weather observations from multiple sources using a modern forecasting system. These sources can come from maritime logs, satellites, or ground stations; they are only limited by the condition that they can be accurately modeled [12]. These data sets are consistent in both spatial and temporal resolution and can span decades in length. However, reliability can vary greatly depending on locations, time periods and variables considered because of the ever changing mix of observations that go into the models [13]. So, while these datasets provide us with something easy to use that incorporates more data than could be individually collected, it unfortunately comes at the cost of accuracy.

A major benefit of these datasets is that they can represent the overall global trends very well. If we could couple this trend information with individual high accuracy datasets then we may be able to get a very accurate picture of how global climate is behaving.

The central idea behind this research is to use the largest of these “reanalysis” datasets to generate a picture of what the global climate looks like and use that information to figure out how to properly weight individual stations observations. This would give the accuracy of individual weather stations but still use insights from these higher spatial resolution reanalysis datasets.

These high spatial resolutions do create too much data to process. We also need a way to reduce the dimensionality of these datasets along with a way to remove higher order trends that simply add noise. To do this we turn to spherical harmonics for dimensionality reduction and principal component analysis to remove the trends.

Since weather takes place on the surface of our spherical planet, using spherical harmonics as a way to reduce the dimensionality of global datasets makes sense. Just like sines and cosines are used to represent functions on a circle for a Fourier series, spherical harmonics are used to represent functions on the surface of a sphere [37]. By converting the global data to the spectral space and truncating it to a reasonable harmonic we end up reducing the size of the problem. Furthermore, by taking this spectral data and running a principal component analysis on it, we can remove higher order climate trends that have little impact on the global climate removing noise and simplifying the problem even further.

This derived data set gives us a good representation of the overall climate trends in spectral form. We then take this data and use it to learn how to optimally weigh individual weather stations to see how much each contributes to these over all trends.

3.2 SPHERICAL HARMONIC TRANSFORM

3.2.1 Background

The first step in the optimal interpolation is to convert global gridded data from spatial to spectral space via the spherical harmonic transform. This transform step became the most difficult part of the research and ended up taking a large majority of the time. While the underlying formulas aren't that complicated, computing them fast, accurately, and to high angular frequencies is no easy task. Throughout this research many incremental improvements were done to this process to aid in making the calculations more manageable. The first version of the code was written in Matlab and took approximately 45 minutes to run a single conversion from spatial to spectral for an angular frequency truncation (Q) of about 100. The final version can do that same calculation in a few seconds. There are even further improvements that can still be done to improve the speed a bit more. However, the calculations are more than fast enough for the processing of what we need to do at this point.

Before diving into the spherical harmonics transform that's been implemented, Shannon's Sampling Theorem should briefly be mentioned [44]. Shannon's Theorem simply stated that: "If a function $x(t)$ contains no frequencies higher than B hertz, it is completely determined by giving its ordinates at a series of points spaced $\frac{1}{2B}$ seconds apart." Basically, if a signal is sampled at a rate higher than twice it's frequency, there is enough information to recreate the original signal, even at locations that weren't sampled!

Now, Shannon's Theorem solves for the eigenvalues and eigenfunctions of the 1-dimensional boundary value problem with periodicity. This technique is more commonly known as the Fourier Transform, but this research will focus on the discrete case: the Discrete Fourier Transform (DFT):

Instead of having just one dimension like in the DFT, this idea needs to be extended into a 2-dimensional boundary value problem with periodicity, as there are now both latitudes and longitudes to contend with.

Unlike the DFT where it's a large summations of sines and cosines, the simplest eigenvalue and eigenfunction solution to this will use large summation of spherical harmonics.

The discrete “spherical” transform DST and inverse DST are equations (3.1) and (3.2) respectively.

$$\tilde{x}_{kl} = \sum_{m=1}^{M-1} \sum_{n=0}^{N-1} \hat{x}_{mn} Y_{kl}^*(\vec{n}_{mn}) w_{mn} \quad (3.1)$$

$$\hat{x}_{mn} = \sum_{k=0}^Q \sum_{l=-k}^k \tilde{x}_{kl} Y_{kl}(\vec{n}_{mn}) \quad (3.2)$$

Where Y_{kl} are orthonormal functions of two angles and two indices k, l over a spherical surface [37]. n are the evenly spaced spherical grid points along the longitude, and m is the Legendre-Gauss Quadrature spaced latitude grid points. $w_{mn} = \frac{2\pi}{N} W_m$ with W_m being the Legendre-Gauss Quadrature Weights. The Q truncation that’s been previously mentioned is the upper bound we will allow to solve for k in equation 3.1) and we see it directly in 3.2

$$Y_k^l(\theta, \phi) = (-1)^l \sqrt{\frac{(2k+1)(k-l)!}{4\pi(k+l)!}} P_k^l(\cos\theta) e^{il\phi} \quad (3.3)$$

Where:

$$P_k^l(x) = \frac{(-1)^l}{2^k k!} (1-x^2)^{l/2} \frac{d^{k+l}}{dx^{[k+l]}} (x^2-1)^k \quad (3.4)$$

Is the Associated Legendre polynomials equation [34].

Now, as M becomes larger, the Legendre-Gauss Quadrature spaced latitude points become more and more evenly spaced, and the weights become more uniform. So for large M , the DST and inverse DST can be replaced with the following two formulas:

$$\tilde{x}_{kl} = \frac{\sqrt{4\pi}}{N} \sum_{m=1}^{M-1} \sum_{n=0}^{N-1} \hat{x}_{mn} Y_{kl}^*(\vec{n}_{mn}) \quad (3.5)$$

$$\hat{x}_{mn} = \frac{\sqrt{4\pi}}{N} \sum_{k=0}^Q \sum_{l=-k}^k \tilde{x}_{kl} Y_{kl}(\vec{n}_{mn}) \quad (3.6)$$

It should be noted just like a Fourier transform, perfectly harmonic data can be converted from spectral to spatial and back to spectral with absolutely no loss of information. However, if the source data is not a perfect harmonic, we will not be able to recover the data without some error when we compare the original data to the rebuilt data. It will contain slight differences that amplify in size as one approaches the poles in the spatial domain. This is due to the singularity at said poles which causes instability in the recreated dataset. This issue will surface in our dataset

that we convert later on because it either originated in the spatial domain, or we simply aren't given enough grid points to recreate the original harmonic.

3.2.2 Implementation

3.2.2.1 ORIGINAL VERSION

The original version of the code took the formulas quite literally with very few enhancements. The first cut of the code was written in Matlab and would take more than 45 minutes to run a single transform (spatial to spectral only) at a Q truncation (where $k \leq Q$) of 100.

With subsequent re-factors, we were able to shrink the processing time some by vectorizing parts of the code but it was still too slow to be useful. Logic was later added to calculate many of these in parallel. While it didn't increase the speed of calculating an individual transform, we could at least perform multiple transforms at once. This version of the code was originally used as part of my Master's thesis.

3.2.2.2 PYTHON

In an effort to increase the speed, the logic was recoded into Python for a number of reasons. At this point in time Python was gaining in popularity, and I thought this would be a good exercise to learn it. Interestingly, there's a solution that I discovered that made it to [stackoverflow](#) for which this reader may know why I was working on this specific issue.

One of the goals of this project was to eventually determine if we could distribute these calculations out to a cluster for calculation instead of having it computed on an individual computer and Python is one of the supported platforms that works with Spark. What I didn't know at the time was that Python wasn't capable of doing complicated custom Spark jobs that other Spark supported languages, like Scala and Java, could do. If we would have continued down the path of using Python, this code base would not have worked out successfully.

Thankfully, there was actually a net performance loss when switching to Python and I also found the language difficult to work with. The lack of curly braces, whitespace importance, and Python's dynamically typed language ended up being a source of major headaches. While others may love it, I personally found Python to be difficult to work with. So, even though I answered a Python related question on Stackoverflow, it doesn't mean I enjoy the language.

3.2.2.3 JAVA

Since the Python porting proved to be of limited success, I decided to begin porting the code to Java as I had become more accustomed with the language through my work outside of this research. Java is also incredibly portable and lends itself to being a native language to

distributed libraries such as Hadoop and Spark. One of aforementioned goals of this research is to distribute out the calculations and work to a cluster of computers to speed up the calculations. Switching to Java ended up being a smart move as the language is very well supported in Spark and the custom logic that we would need to implement in Spark can be easily accomplished.

Thus began the work of porting the code to yet another language. Unfortunately, just like the switch to Python, there was no performance gain over the original Matlab code. However quick and dirty the Matlab programming language is, the underlying logic that is done is still very efficient as it does use native libraries for calculations at its core.

The Java implementation still had all the baggage we had with the original implementation so we still could not successfully calculate Q truncations higher than 100 effectively.

All was not lost, as having migrated the code base to Java we gained all of the flexibility that came with the switch. While the code was indeed slower now (we had done a direct and dirty port) no effort had yet been spent on trying to improve the code logic, and we had raised the ceiling for potential speed increases.

It was however, not without some issues.

3.2.2.4 ASSOCIATED LEGENDRE POLYNOMIALS

The first issue was that Java lacked any kind of implementation that could calculate the “Associated Legendre Polynomials”. Equation 3.3 requires these values P_k^l to be able to calculate the spherical harmonic. Both Matlab and Python already had implementations that could calculate these out of the box, Java did not. At first glance, the problem didn’t appear that complicated as there are a number of recurrence relations that could be used [55], but these original solutions were numerically unstable. They worked well in the beginning, for low numbered Q truncations but would cause issues when both k and l got large.

After some work we eventually settled on which of the many relationships available would work out the best. We used the following recursion relations and starting values to our advantage:

$$P_l^l(x) = (-1)^l (2l - 1)!! (1 - x^2)^{l/2} \quad (3.7)$$

$$P_{l+1}^l(x) = x(2l + 1)P_l^l(x) \quad (3.8)$$

$$P_k^l(x) = \frac{(x(2k - 1)P_{k-1}^l(x) - (k + l - 1)P_{k-2}^l(x))}{(k - l)} \quad (3.9)$$

Where $!!$ is the double factorial. Using these three formulas, we can build out any P_k^l we wanted. The major issue with this first cut of the code was the factorials that are in the $Y_k^l(\theta, \phi)$

function. As k and l increase, the numbers get larger and larger and the algorithm is still numerically unstable. It did however surpass the poor stability of our initial implementations.

The following code was developed that scaled out “fairly” well:

```
public class AssociatedLegendrePolynomials {

    private Map<Integer ,Map<Integer ,Double>>> _calculatedValues ;
    private double [][] _legendre ;

    public double [][] GetAsDouble () {
        return _legendre ;
    }

    public RealMatrix GetAsRealMatrix () {
        RealMatrix mat = MatrixUtils.createRealMatrix(_legendre.length
            , 1);
        for (int k = 0; k < mat.getRowDimension(); k++) {
            mat.setEntry(k, 0, _legendre[k][0]);
        }
        return mat;
    }

    public FieldMatrix<Complex> GetAsFieldMatrixComplex () {
        RealMatrix mat = MatrixUtils.createRealMatrix(_legendre.length
            , 1);
        for (int k = 0; k < mat.getRowDimension(); k++) {
            mat.setEntry(k, 0, _legendre[k][0]);
        }
        return ToFieldMatrixComplex(mat);
    }

    public double AssLegFunc2(int l, int m, double x) {
        boolean alreadyCalculated = this._calculatedValues.get(l).
            containsKey(m) ? true : false;
        if (alreadyCalculated) {
            return this._calculatedValues.get(l).get(m);
        } else if (l == m) {
            double right = Math.sqrt((1.0-x)*(1.0+x));
            double doubleFactorial=1.0;
```

```

        double value = 1.0;
        for (int i=1; i <= m; i++) {
            value *= -doubleFactorial*right;
            doubleFactorial += 2.0;
        }
        this._calculatedValues.get(1).put(m, value);
        return value;
    } else if (l == (m+1)) {
        double value = x*(2*m +1) * AssLegFunc2(m,m,x);
        this._calculatedValues.get(1).put(m, value);
        return value;
    } else {
        double value = (x*(2*l -1) * AssLegFunc2(l-1,m,x) - (l+m-1)
            *AssLegFunc2(l-2,m,x))/(l-m);
        this._calculatedValues.get(1).put(m, value);
        return value;
    }
}

public AssociatedLegendrePolynomials(int N, double X) {
    this._calculatedValues = new HashMap<Integer,Map<Integer,Double>>();
    for (int k = 0; k <= N; k++) {
        this._calculatedValues.put(k, new HashMap<Integer,Double>());
    }

    double[][] output = new double[N+1][1];
    for (int k = 0; k <= N; k++) {
        output[k][0] = AssLegFunc2(N,k,X);
    }

    this._legendre = output;
}
}

```

3.2.2.5 LOG SPACE

Another benefit of switching the code over to Java, was the fresh look at the code helped to find many places for improvement that could be made.

One such improvement was that some calculations could take place in the log domain. Specifically in the factorial portions of the code (Eq. 3.3). This trick allows for a simple addition on the factorials instead of multiplying them out [36].

$$n! = \prod_{i=1}^n i = e^{\log[\prod_{i=1}^n i]} = e^{[\sum_{i=1}^n \log i]} \quad (3.10)$$

Afterwards, the exponent can be taken on the results to put the values back in the regular domain. The following is an example of such a trick, but we used it in more than a few places to simplify some of the calculations as well.

$$\sqrt{\frac{(2k+1)(k-l)!}{4\pi(k+l)!}} = \exp\left(\frac{1}{2}\left(\log(2k+1) - \log 4\pi + \sum_{n=1}^{k-l} \log n - \sum_{m=1}^{k+l} \log m\right)\right) \quad (3.11)$$

This helped a great deal because it scales the numbers better. Since these calculations are done in code, factorials don't exactly play nice, the numbers can get really big really fast. Only so many bits of information are used when storing a number (64 bits in our case), so the bigger the number stored, the greater the loss of precision. By doing the calculations in log space, the factorials are kept from blowing out of proportion, and since the math is basically: (Big Number)/(Even Bigger Number) the result is a simple fraction. Preventing these numbers from growing too big increases the accuracy of the system because as the Q truncation values get bigger and bigger, the numbers in the spherical harmonic triangle actually get smaller and smaller. At some point, they get so small that they become zero because of the loss of precision to calculate them.

This trick makes it possible to get around $Q = 150$ before this loss of precision issue starts happening. Before the change, this bottleneck hit around the $Q = 100$ threshold.

While this code change helped us increase our system's overall accuracy, it added a bit more complexity to the code, and increased our upper bound on Q . Both of these had the undesirable effect of slowing the calculations down.

Another downside to this log trick is that it forced us to move a lot of the math calculations from the normal plane to the complex plane. This was because there were sections of the code where it was possible to get logs of negative numbers. In the real domain, this would cause a lot of grief, but not so in the complex one.

In the end, this increased our accuracy by allowing us to go to higher Q truncations, but at the cost of slowing the system down.

3.2.2.6 NONEQUISPACED FAST FOURIER TRANSFORM (NFFT)

While migrating the code, an open source C library was discovered that claimed it could take care of the spherical harmonics transform section that we had already ported [27], but, by using a lot of tricks, and the fact that it was coded in C, this library could potentially out pace our current implementation with ease [24].

Just because you may be using Java for most of your research, doesn't mean you should completely ignore solutions built in C/C++. No matter how hard Java tries, it will never be as fast as code written in C/C++, but the code will certainly be much easier to read and follow along.

Java Native Interface (JNI) allows Java code to call native compiled code and pass variables between the two. While Java is flexible, it's not often associated with numerical processing as it's compiled by an interpreter at runtime. This allows Java code to be very portable, but lacks the speed that native compiled libraries like C can have. Processing these calculations on compiled code could net us some huge speed gains.

A JNI wrapper was written as part of this research to see if we could use this library instead of our own implementation. If the differences between the two are minimal, then the code can be switched to use their implementation. With the admittedly very limited experience I have with C/C++, I was able to write my own wrapper to call up these native libraries. There are numerous tutorials online that show you how to do it. The one I ended up using was written by Chua Hock-Chuan from Nanyang Technological University in Singapore [22]. This tutorial is usually the first hit when you google: "JNI Tutorial."

Throughout this research, JNI was my common "go to" when I couldn't find a native Java solution to a particular task that needed doing. I can't emphasize enough that it's always better to avoid reinventing the wheel. There usually exists a solution to many common programming tasks out there. Avoid wasting your time writing a custom solution wherever possible. Even if you can't find a Java solution, exploiting JNI can be a great workaround to using an already vetted library who's author(s) are usually smarter than you.

Preliminary research showed that this library was very fast, but the errors it generated were far greater than the ones from the current implementation. Depending on the function and certain characteristics, the library would give reasonably good results. Unfortunately, in some cases the errors it generates can be quite large. For brevity, the research implementation will be known as "our" method, while the C library implementation will be known as the "NFFT" method.

Beginning with a harmonic example: $3Y_6^{-2} + 2Y_5^2$. Plots containing the original, rebuilt, and the sum of squared errors at $Q = 49$ can be found in figure 3.3. The values on the left side of table 3.1 contains the full set of errors between the two implementations. While the NFFT method is very fast, the errors it generates are larger for this particular example than the errors in our method. A plot of comparing the values in the table can also be found in figure 3.1.

It's clear that the our method performs much better with respect to generating lower errors. Unfortunately it does come at the cost of processing time. At this point a $Q = 49$ truncation could take a few minutes under our method (going from spatial to spectral and back to spatial). While the NFFT method does the same in roughly 0.03 seconds. This difference is astronomical! It is unfortunate that the errors get rather high.

The next example is a simple function of $\sin(2x) + \cos(2y)$ (where x are points in radians across the longitude, and y are points in radians along the latitude) really shows problems with the NFFT implementation. Depending on the Q truncation used, vastly different answers are calculated. Table (3.1) contains the numerical values for this plot on it's right side. So, if the wrong Q is accidentally selected, the rebuilt spatial values can be drastically different than the original. A plot comparing the original, rebuilt, and errors can be found in figure 3.4.

Our implementation works much better in this example. Errors of this magnitude seem to negate all the speed benefits that would be gained from switching to this library. For this research, the NFFT method posses too much error for the processing.

The reasoning behind the poor performance may have something to do with the libraries intended purpose. The library seems to assume that most use cases have a user starting in the spectral domain and the library is advertised as being able to calculate high Q truncations. The library also had numerous initialization options that I honestly may not have used correctly. Contacting the authors of the library did not bear much fruit either, as they were not responsive. As it currently stands, the library was of little value to our research effort.

The one bright spot from this research tangent was JNI. JNI proved an invaluable tool in being able to use non Java libraries. A tool that we ended up using a few times during the course of this research. While I didn't directly use this JNI code for the final solution in this research, I have used JNI in my professional career outside of academia since learning about it here.

3.2.2.7 REFACTORING

Since the code was functioning correctly in Java, it was time to do a complete refactor of it to try and speed things up.

We were storing a lot of useless information in memory that we didn't really need. At many of the intermediary steps, we were intentionally keeping values that were only used once and never referenced again. This code was a carry over from the original development code that

Table 3.1. Sum of squared errors for the two formulas at various Q-truncations.

Q	Our Method	NFFT Method	Q	Our Method	NFFT Method
10	1.63E-027	2.0299928587	10	0.2177263502	5.3598707565
11	9.48E-028	1.5512610391	11	0.0099378587	0.0206100802
12	1.62E-027	1.2552990188	12	0.1920140185	6.2089734387
13	1.54E-027	0.9764447143	13	0.0081897349	0.0176379677
14	2.01E-027	0.839103983	14	0.1722923179	7.0589711511
15	1.58E-027	0.6609809451	15	0.0069517229	0.0154565091
16	2.69E-027	0.5924032872	16	0.1566241758	7.9095807212
17	8.48E-027	0.4711813325	17	0.0060316131	0.0137854578
18	3.87E-027	0.4356658954	18	0.1438378658	8.7606257636
19	3.38E-027	0.3492268536	19	0.0053222661	0.0124628983
20	6.04E-027	0.3307648138	20	0.133180191	9.6119914039
21	5.95E-027	0.2668669674	21	0.0047594767	0.01138889
22	1.91E-026	0.257638038	22	0.1241434068	10.4635998376
23	2.49E-026	0.2090241277	23	0.0043025331	0.0104984443
24	4.16E-026	0.2049579736	24	0.1163718974	11.3153965657
25	2.02E-026	0.1670896163	25	0.0039244237	0.0097474825
26	2.01E-026	0.1659631934	26	0.1096086532	12.1673422999
27	1.76E-026	0.1358778534	27	0.003606549	0.009105051
28	3.81E-026	0.1364315234	28	0.1036629895	13.0194080083
29	2.16E-026	0.112125787	29	0.0033356978	0.008548771
30	3.59E-026	0.113625746	30	0.0983902431	13.8715717737
31	4.73E-026	0.0937040058	31	0.0031022369	0.0080620574
32	2.10E-026	0.0957142048	32	0.0936785435	14.7238167382
33	5.60E-026	0.0791793363	33	0.0028989814	0.0076323544
34	3.30E-026	0.081437436	34	0.0894399276	15.5761297207
35	1.14E-025	0.0675612615	35	0.0027204666	0.0072499789
36	7.36E-026	0.0699087313	36	0.085604214	16.4285002648
37	9.51E-026	0.0581492142	37	0.0025624645	0.0069073409
38	6.94E-026	0.0604907499	38	0.0821146807	17.280919967
39	3.81E-026	0.0504375502	39	0.0024216539	0.0065984052
40	1.07E-025	0.052716913	40	0.078924954	18.1333819952
41	1.25E-025	0.0440546968	41	0.0022953904	0.0063183113
42	1.26E-025	0.0462400214	42	0.0759967274	18.9858807367
43	8.68E-026	0.038723188	43	0.0021815432	0.0060630994
44	9.90E-026	0.0407978897	44	0.073298063	19.8384115368
45	1.53E-025	0.0342328231	45	0.0020783765	0.0058295102
46	3.85E-025	0.0361898882	46	0.0708021074	20.6909705019
47	1.67E-025	0.0304222715	47	0.0019844624	0.0056148361
48	1.92E-025	0.0322606319	48	0.0684861079	21.5435543499
49	1.27E-025	0.0271662372	49	0.0018986156	0.0054168082
50	2.41E-025	0.0288884483	50	0.0663306513	22.3961602934

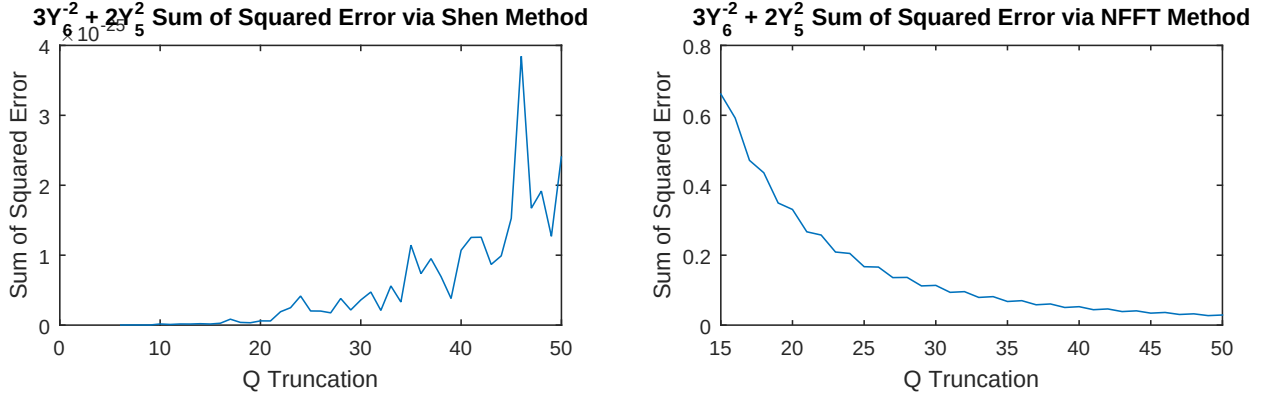


Figure 3.1. Error for harmonic example

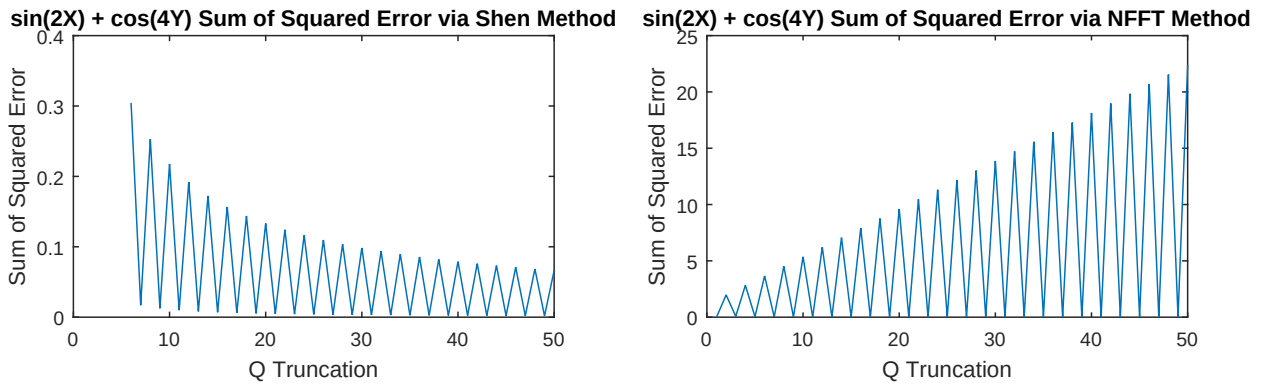


Figure 3.2. Error for function example

was meant to aid in finding bugs. However, since the code was functioning correctly at that point, there was no need to keep storing all of these objects in memory. In fact, it would be beneficial to reuse these elements and simply append them with the new calculated values as we went around.

For example, in some of the summation calculations we were running through a loop to populate an array of values, only to then loop through said array at the end to sum up its values. Why not sum the values as you're calculating the original loop? Minor fixes like this ended up giving us major speed gains as we were removing excess loops from the logic. The basic premise was we tried to remove excess loops wherever possible.

Another place where we were storing a lot of useless information was in the harmonics themselves. Table 3.2 shows a representation of how the data was stored. Everywhere there is an x is where a harmonic value could potentially be. Everywhere there was a zero was wasted space as we never used that field.

The memory footprint can be further shrunk by not storing values where $m < 0$. Instead, we can use the following equation 3.12 to pull that data out if needed [57] [37]. While this does add some calculations to the mix, when we're actually processing the spherical harmonic

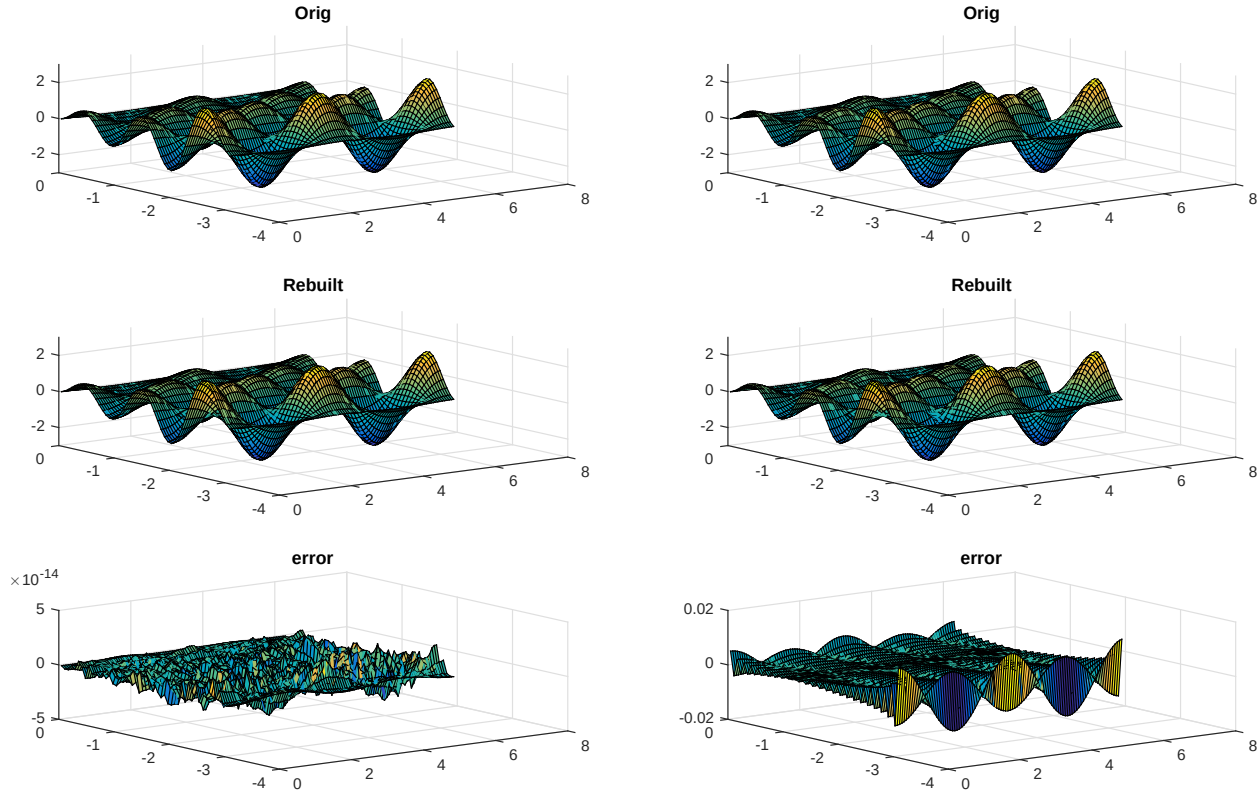


Figure 3.3. Harmonic Plot Q=49

transform, it's more efficient to use this logic instead of calculating the negative m values from scratch.

$$Y_l^{m*}(\theta, \phi) = (-1)^m Y_l^{-m}(\theta, \phi) \quad (3.12)$$

The minimum data is stored in memory with a pattern similar to table 3.3. With the following two equations used to set aside the necessary elements we will be storing in memory, and how to retrieve a specific harmonic back out.

$$N = \frac{(Q+1)^2 + Q + 1}{2} \quad (3.13)$$

$$i(l, m) = \frac{l^2 + l + 2}{2} + m \quad (3.14)$$

As Q gets larger this saves us a lot of space in memory compared to the old method and while there is a slight trade off in computation, we make up for it in using the same relationship to speed up the spherical harmonic transform calculations, which results in an overall net gain.

Another change that gained us some very healthy speed gains was moving all of the calculations out of matrix libraries. In the beginning of the port, Java matrix libraries were used to

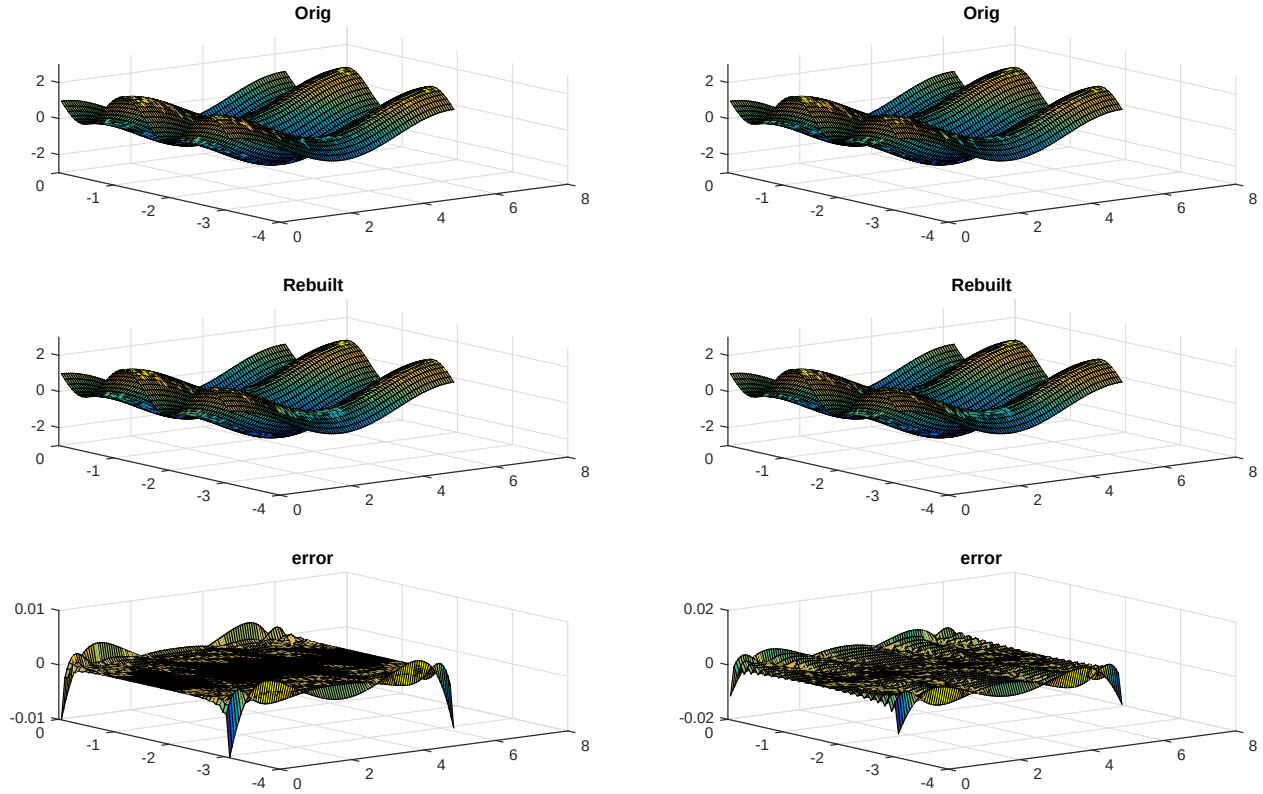


Figure 3.4. Function Plot $Q=49$

simplify the necessary coding. This made it very easy to perform matrix manipulations on the numbers, but added overhead, as libraries were being used that added a lot of unnecessary boilerplate code. The library was useful and double checked that you sized the matrices correctly before multiplying, but creating and destroying so many of these objects was overkill. To solve this, I ended up writing functions to perform matrix manipulations by hand on standard one and two-dimensional Java arrays. This had to be done for both “double” variables and “Complex” variables, and had to also allow mixing of the two types. This alone added well over a thousand lines of code to the code base. While these functions may have lacked a lot of the features of the more proper Java matrix implementations, they performed with considerably less overhead and thus ran faster. This rewriting was a slow process but like all other efforts, netted us even more speed gains in the end.

3.2.2.8 NEW FORMULAS

By now, speed was no longer an issue, but processing the factorials in log space was nearing the end of its usefulness. While we could now go to high Q truncation values (higher than what we needed for this research), doing calculations this way brought about its own issues; we did lose some precision to the estimate.

Table 3.2. Example of spherical harmonics spectral form.

l \ m	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6
0	0	0	0	0	0	0	x	0	0	0	0	0	0
1	0	0	0	0	0	x	x	x	0	0	0	0	0
2	0	0	0	0	x	x	x	x	x	0	0	0	0
3	0	0	0	x	x	x	x	x	x	x	0	0	0
4	0	0	x	x	x	x	x	x	x	x	x	0	0
5	0	x	x	x	x	x	x	x	x	x	x	x	0
6	x	x	x	x	x	x	x	x	x	x	x	x	x

Table 3.3. Example of how data is indexed and stored.

	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6
							1						
1							2	3					
2							4	5	6				
3							7	8	9	10			
4							11	12	13	14	15		
5							16	17	18	19	20	21	
6							22	23	24	25	26	27	28

By chance, I noticed new formulas I had not seen while going through reference [40]. The main idea, was to move some of the calculations out of the transform equations and into the Associated Legendre polynomial equations. This allows us to cancel more parts of the equation, remove the factorials completely. The log trick was no longer needed! Another benefit is that we removed more for loops from the overall program. Moving the logic here greatly reduced the necessity for loops since we were already processing the loops during this phase. This would net us even more of a speed increase.

We basically do a new version of the Associated Legendre polynomials that we call $\tilde{P}_l^m$.

$$\tilde{P}_l^m = \sqrt{\frac{(2l+1)(l-m)!}{4\pi(l+m)!}} P_l^m \quad (3.15)$$

The recurrence relation then changes to:

$$\tilde{P}_l^m = \sqrt{\frac{4l^2-1}{l^2-m^2}} \left[x \tilde{P}_{l-1}^m - \sqrt{\frac{(l-1)^2-m^2}{4(l-1)^2-1}} \tilde{P}_{l-2}^m \right] \quad (3.16)$$

We can now start the recurrence relation with two starting values. The first of which is:

$$\tilde{P}_m^m = (-1)^m \sqrt{\frac{2m+1}{4\pi(2m)!}} (2m-1)!! (1-x^2)^{m/2} \quad (3.17)$$

Thanks to the identity $(2m)! = (2m-1)!! 2^m m!$ [56], we then simplify to:

$$\tilde{P}_m^m = (-1)^m \sqrt{\frac{2m+1}{4\pi}} \prod_{i=1}^m \frac{(2i-1)(1-x^2)}{2i} \quad (3.18)$$

The beauty of this simplification is that while we're still technically doing a factorial calculation inside of the product; since we're balancing out the bottom and top double factorials in the summation loop, the numbers aren't exploding. This, along with the removal of all the factorials in other parts of the code, makes these new formulas extremely numerically stable.

The second starting value is:

$$\tilde{P}_{m+1}^{ms} = x \sqrt{2m+3} \tilde{P}_m^m \quad (3.19)$$

With all three of the above formulas, we are left in the same position as earlier but with much better logic. We then had to modify the transform code to remove the variables we pulled into the new $\tilde{P}_l^m$ and we saw another healthy speed gain; we thus again increased Q 's upper limit. With the removal of all the factorials from the equations there likely is no longer an upper limit to these equations.

One last benefit from this trick is that we simplified the code significantly. You may recall that when we switch to doing calculations in log space, we had to change a lot of our math to the complex domain. While we still have results that are in that domain, we were able to pull back a lot of those calculations and put them back in the real domain. This simplified the code as we no longer had to worry about doing complex arithmetic and further reduced the amount of code that was being called. With these changes, we netted yet another speed gain.

This whole process of refactoring and reformulating everything changed the code dramatically. For example, the original version of the conversion from spatial to spectral was around 200 lines of code long; the final version brought it down to about 120. This doesn't mean that we ended up with less code (we actually ended up with a lot more code), but the portion of the code that handles the conversion specifically shrunk drastically. These removed lines contained loops and values that were no longer needed to calculate, which both simplified the code and made it run faster.

3.2.2.9 HIGH PERFORMANCE SPHERICAL HARMONIC TRANSFORM FOR NUMERICAL SIMULATIONS (SHTNS)

During the course of these numerous refactors, another C library was discovered that showed promise [43]. We decided to again use the JNI wrapper method to see if this library could prove as good replacement for our current implementation. This was done prior to the completion of all of the refactors, so we had not yet netted all of the possible speed gains.

When I began writing the wrapper, I immediately reached out to it's author, Nathanael Schaeffer, with questions and some minor requests for sample usage code. Nathanael was very helpful, and provided me with examples of how to run his code correctly. He was readily available to answer questions as well.

This proved an excellent opportunity to determine if I had coded my implementation correctly. Everything appeared to work fine with the tests I had done, but this could validate my code worked. In the previous JNI wrapper attempt, our results were so different from NFFT that if these results also didn't match, maybe the problems were an issue with our implementation.

To test my process, I decided to pick the first date from the NCEP re-analysis dataset and process it, from spatial to spectral, and back again. Then, take the sum of squared errors between the original and rebuilt values to see what the differences were. I did this with both our implementation and SHTns to see if everything lined up correctly. As can be seen in table (3.4) the values are nearly identical between the two sets of SSEs.

To thank Nathanael for his help, we provided him with the JNI wrapper, C, and Java code that had been written and it was included in the next release of his library.

Unfortunately, while the library performed well, we did required the flexibility of our own implementation to do some of the required calculations later on. It wasn't all for nothing; through this collaboration, SHTns was given a wrapper library so others can use it with Java in the future. More importantly, we confirmed that our implementation was working correctly.

For calculations dealing with a single transform back and forth, our library actually performed faster than SHTns. SHTns does have the ability to cache many values so if subsequent runs use the same coordinate points and truncations, those values will simply be pre-cached and used over in the subsequent calculations. This is not something we currently exploit in our code.

While we again didn't use the JNI code generated, we will exploit JNI later on.

3.2.2.10 MULTI-THREADING

Up to this point, the code was running more than fast enough for our calculations, and we hadn't even considered multi-threading it. There were two routes we could explore for

Table 3.4. SHTns vs Our implementation

Q	SSE (SHTns)	SSE (ours)
2	395724.7840524517	395724.7840524517
3	379583.70431904437	379583.7043190444
4	346818.7212545067	346818.7212545066
5	309507.2877373056	309507.2877373056
6	278294.23825750075	278294.23825750075
7	242932.66822644198	242932.66822644195
8	220411.93132931652	220411.93132931652
9	194886.80271118574	194886.80271118565
10	172118.92746376994	172118.92746376994
...	...	...
89	2836.1906154754997	2836.190615475484
90	2693.5510360854278	2693.5510360854096
91	2537.358195660882	2537.3581956609037
92	2401.6592313783285	2401.659231378339
93	2266.1945811668875	2266.194581166875

multi-threading: try to split of individual harmonics to individual threads, or simply calculate simultaneous runs of the code at once.

We decided to experiment with both options and wrote code to test out both methods. We found that attempting to multi-thread an individual transform, had trouble saturating the available processors if the Q truncation was too low. However, if the truncation was high; it made these seemingly impossible calculations doable. The code now leaves multi-threading an option to be selected by the user.

Next, the code was tested to make sure it could run multiple instances of itself at once, but the results between the threaded and non-threaded runs showed inconsistencies. We discovered parts of the code were not thread safe and values were being modified by one thread and accessed by another thread, completely destroying the consistency. The problem resided in how Java handled classes as they were always passed by reference and never explicitly copied when used. This caused issues when multiple threads were trying to access the same original class at once. One thread would make changes to the class, assuming it only affected it's particular thread while other threads were accessing it as well. The results became a random mess dependent on which had accessed the data first. Since the divvying of work out to separate threads happens somewhat randomly, results were different depending the execution. The errors where eventually discovered and the code updated so these processes became thread safe, which allowed for the code base to allow simultaneous runs in parallel.

Table 3.5. Original $Y_2^1 = 1.0$

	-5	-4	-3	-2	-1	0	1	2	3	4	5
0						$0 + 0i$					
1					$-0 + 0i$	$0 + 0i$	$0 + 0i$				
2				$0 - 0i$	$-1 + 0i$	$0 + 0i$	$1 + 0i$	$0 + 0i$			
3			$-0 + 0i$	$0 - 0i$	$-0 + 0i$	$0 + 0i$	$0 + 0i$	$0 + 0i$	$0 + 0i$		
4		$0 - 0i$	$-0 + 0i$	$0 - 0i$	$-0 + 0i$	$0 + 0i$	$0 + 0i$	$0 + 0i$	$0 + 0i$	$0 + 0i$	
5	$-0 + 0i$	$0 - 0i$	$-0 + 0i$	$0 - 0i$	$-0 + 0i$	$0 + 0i$	$0 + 0i$	$0 + 0i$	$0 + 0i$	$0 + 0i$	$0 + 0i$

3.2.3 Verification

Verification of the code had indirectly been accomplished when we tested the SHTns library. Due to many more code cleanups and simplifications that occurred, on several occasions the code would break and fail giving invalid results. To catch these kind of issues we turned to Java’s unit testing ability; we just needed something to test that would always give consistent results.

While obvious now, when one comes from the Matlab world of programming, unit testing one’s code is not something that’s intuitive. However, unit tests are always recommended. Just because a set of code performs well when it was originally tested, as changes slowly take place in other parts of the code eventually something will break. Many times I ran the code under the assumption that my results were correct only to discover there was a small one-line bug in my code that invalidated the results. An error that would have been caught with a very simple unit test. Writing unit tests in Java is easy and worth the extra effort; the tests are run every time code is compiled. The specific unit tests mentioned here can be found at `com.mrsharky.discreteSphericalTransform.DiscreteSphericalTransformTest` in the code base.

Recall that spherical harmonics won’t be able to give identical results when converting back and forth if the dataset isn’t an exact harmonic. To write a functional unit test of our process we then need to create a harmonic in the spectral space, convert it with our inverse DST (3.2) to transfer the data to the spatial domain, and then convert it back to spectral space using the DST (3.1) function. If our implementation works correctly we should have a “nearly” identical copy of the original dataset.

A test case was written than would check if a $Y_2^1 = 1.0$ harmonic (seen in table 3.5) could be converted back and forth and get identical results. Converting this data to spatial and back to spectral generated the harmonic seen in table 3.6 (note that the data in the rebuilt table was truncated to fit to the page). As evidenced by the results, our implementation processed the harmonic correctly.

A more difficult test case was then written, one that would test how effective the code could run at high Q truncations. Since unit tests are limited by the amount of memory they have

Table 3.6. Rebuilt $Y_2^1 = 1.0$

	-2	-1	0	1	2
0			$7.3e^{-18} - 0i$		
1		$-2.7e^{-17} - 3.7e^{-18}i$	$6.6e^{-17} - 0i$	$2.7e^{-17} - 3.7e^{-18}i$	
2	$1.7e^{-18} - 4.5e^{-18}i$	$-1.0000000000000009 + 3.1e^{-17}i$	$1.9e^{-18} - 0i$	$1.0000000000000009 + 3.1e^{-17}i$	$1.7e^{-18} + 4.5e^{-18}i$
3	$7.4e^{-17} + 7.2e^{-17}i$	$-9.4e^{-17} - 6.8e^{-19}i$	$-1.6e^{-17} - 0i$	$9.4e^{-17} - 6.8e^{-19}i$	$7.4e^{-17} - 7.2e^{-17}i$
4	$7.1e^{-18} + 1.9e^{-18}i$	$-7.0e^{-17} - 7.5e^{-19}i$	$-3.1e^{-18} - 0i$	$7.0e^{-17} - 7.5e^{-19}i$	$7.1e^{-18} - 1.9e^{-18}i$
5	$1.4e^{-17} + 1.0e^{-17}i$	$-3.5e^{-18} + 2.4e^{-18}i$	$-5.1e^{-18} - 0i$	$3.5e^{-18} + 2.4e^{-18}i$	$1.4e^{-17} - 1.0e^{-17}i$

access to, we wrote a regular Java program

(`com.mrsharky.discreteSphericalTransform.tests.HighRandomHarmonicTest`) that would randomly populate a harmonic at all k and l values for a given Q , convert it to spatial then back to spectral space and then compare the differences.

Memory and time appear to be the only issues here; we've been able to go as high as a $Q = 400$ truncation and been able to convert it back and forth within an error of at least $10e^{-12}$ for both the real and imaginary parts of the harmonic. This specific error requirement was arbitrarily selected and the real error may be smaller. The one draw back to this calculation is the processing time. On a six core virtual machine the process takes roughly 45 minutes to complete. This was approximately how long it took to go one direction in the original Matlab implementation at $Q = 100$.

Including the negative harmonics, there are a total of $(Q + 1)^2$ harmonics that need to be processed. The original Matlab implementation didn't take advantage of the symmetry so to calculate $Q = 100$ it would be a total of 10201 harmonics. This new version of the code can convert in both directions at $Q = 400$ which would be a total of $160801 * 2 = 321602$ harmonics. Not only that, but the higher the Q the larger the calculations, because you need to generate a larger spatial component to calculate the spectral back from it. For a given spectral that contains a maximum harmonic of $k = Q$ and $l = P \leq Q$ then you need at least $Q + 1$ latitude points and $P * 2$ longitude points in the spatial sample to reconstruct the harmonic to spectral again. So, the problem is not only bigger because of the higher Q but the other data requirements as well. This also means more Associated Legendre Polynomials are needed long with other tasks. At $Q = 100$ the entire back and forth process takes roughly 9 seconds with the new code base.

3.2.4 Data and Processing

One immediate benefit of switching to Java is we were able to use the same code we had already written for loading NetCDF files directly as part of the process written for the web application (see 2.6.1).

NetCDF, or network common data form, is a space saving data abstraction for storing and retrieving multidimensional data [41] and is very common way of spreading gridded climate data because of it's high compression capabilities.

Table 3.7. Finding the optimal Q - SSE

Q	SSE	Q	SSE	Q	SSE
2	395724.784052452	89	2836.1906154755	98	1921.7545087764
3	379583.704319044	90	2693.5510360854	99	1887.0222353859
4	346818.721254507	91	2537.3581956609	100	1864.9227882811
5	309507.287737306	92	2401.6592313783	101	1856.087879133
6	278294.238257501	93	2266.1945811669	102	1845.7149256317
7	242932.668226442	94	2166.4908398055	103	1851.8108387576
8	220411.931329317	95	2081.6960501079	104	1853.5228540447
9	194886.802711186	96	2021.166176393	105	1870.9281080684
10	172118.92746377	97	1963.8082937756		

One of the best features of Java is something called Maven. When you create a Java project there is an option to create it as a Maven one. Maven makes package management incredibly easy. When an outside library needs to be used as part of your research, you simply need to modify your “pom.xml” file to declare the dependency. To process netcdf files in our code, we simply add the following to our “pom”:

```
<dependency>
  <groupId>edu . ucar</ groupId>
  <artifactId>netcdf4</ artifactId>
  <version>4.5.5</ version>
</dependency>
```

For those who have struggled with making your own makefiles and incorporating other peoples’ libraries in said files, Maven makes the process seamless and in effect encourages users to use other open source libraries because it’s so easy to use.

3.2.4.1 OPTIMAL Q

While implementing Nathanael Schaeffer’s SHTns library, we noticed that his library wouldn’t allow us to run with Q values higher than 93. Inquiring why this was the case, he mentioned that since our source data had 94 latitudinal points, there was no point allowing Q to become higher than that because we would be over sampling from the available frequencies. This was analogous to how Shannon Sampling worked.

Simple tests converting from spatial to spectral and back to spatial showed this wasn’t actually the case. By now we had proven that both SHTns and our own Java implementation gave identical results. When inspecting the SSE between the original and rebuilt the “optimal” value appeared to be $Q = 102$, not 93 (table 3.7). The results in both this table and the table 3.8. one were generated by summing the values of multiple dates together as we didn’t want to rely on identifying the minimum from a single dataset.

Table 3.8. Finding the optimal Q - Integral over the Sphere

Q	SSE with $\sin \theta$	L=0,M=0	Q	SSE with $\sin \theta$	L=0,M=0
89	1366.7351625939	0.4194026464	100	867.4705015907	0.2661928495
90	1291.8823831911	0.3964327432	101	864.1162657051	0.2651635519
91	1213.3022534931	0.3723190034	102	861.5724786841	0.2643826849
92	1141.2352688153	0.3502038903	103	870.5641430846	0.2671422301
93	1068.981433098	0.3280310813	104	878.6075073444	0.2696113311
94	1013.3753045662	0.3109670469	105	888.562019789	0.2726661358
95	970.4235261326	0.2977862981	106	901.6308698944	0.2766767556
96	938.0050965276	0.2878378242	107	924.3367667765	0.2836436042
97	910.7394448863	0.2794707026	108	944.2639019491	0.2897581157
98	892.0126625498	0.2737239842	109	966.9474699386	0.2967185512
99	878.1633913634	0.2694741996	110	993.6890881404	0.3049248256

We initially thought this was the case simply because each of the grid points were equally weighted. Since the errors were greater at the poles, but take up less surface area, we thought that weighing the points properly for the SSE error calculation would prove this was not the case. Discussing this with Nathanael, he also had another idea to measure the error:

- Compute the squared differences at each point: $\text{diff}(\theta, \phi)$
- Compute the SH transform of this array
- The $l = 0$ and $m = 0$ are proportional to the actual integral over the sphere, use this value as error.

The results from both these of calculations are in table 3.8, and they showed that the minimum was not at 93, but instead was at 102.

Since our goal was to minimize the error involved with using spherical harmonics and SHTns wouldn't work for picking a Q truncation that it believed was too high, we decided that it was best to stick with our own implementation of the code, and continued with $Q = 102$ as the truncation for the research. This result may be caused by the source of the original dataset not coming from a pure harmonic and the values were interpolated to a Gaussian gridded scheme. Another reason for this disconnect is that perhaps this dataset was originally from a spherical harmonic, but the frequencies were truncated to the lower dimension dataset that has been made available.

3.2.5 Conclusion

There is still room for improvement in the transform method if desired. One such improvement would be to add the ability to pre-cache some of the values if the same spatial latitude and longitude points are processed over and over again. Elements such as the Associated

Legendre Polynomials, the Legendre Quadrature weights, and others could be saved in memory and read from once when processing the same grid points.

Improvements could also be made to reduce the memory footprint from processing high Q truncations as the system currently pre-processes all of Associated Legendre Polynomials before it begins the processing. This is done because it's a recursive algorithm that can start from the bottom and work it's way up. Saving all this data also takes up a large footprint in memory, because we're basically building out a triangle similar to the look of the spherical harmonics for each of the latitude points. If we instead worked our way from the low values up and removed no longer needed polynomials from the equation, then we could keep the memory footprint from expanding too fast. We would just need to keep the results from the current "row" and preceding row we're on to recreate the future row while deleting no longer needed ones. This would save a considerable amount of memory as this currently takes up the largest footprint in the whole system. Doing this would allow the system to go to even higher Q truncations with less memory.

However, for the current work to be done here, there isn't a need to go to higher and higher Q truncations as our optimal for this dataset does appear to be at $Q = 102$.

3.3 PRINCIPAL COMPONENT ANALYSIS OF RE-ANALYSIS DATA

3.3.1 Background

The reason for the previous Spherical Harmonics Transform section was to convert the reanalysis data from the spatial domain to the spectral domain to reduce the dimensionality of the problem. Now that we've completed that, we need to process a principal component analysis on the data.

Before any calculations can be done, the spectral form for a processed date of harmonics needs to be restructured. Figure 3.9 shows how we structured the spherical harmonic data in memory for processing a PCA.

Table 3.9. Example of spherical harmonics spectral form. $Q = 7(Q + 1)^2 = 49$

	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6
0							1						
1						2	3	4					
2					5	6	7	8	9				
3				10	11	12	13	14	15	16			
4			17	18	19	20	21	22	23	24	25		
5		26	27	28	29	30	31	32	33	34	35	36	
6	37	38	39	40	41	42	43	44	45	46	47	48	49

Depending on the Q picked, there will be $(Q + 1)^2$ data points in this triangle. By restructuring each date's spectral form into a column vector, the data between all the dates can be column bounded together into one large matrix. The covariance matrix is then calculated between these dates and harmonics by equation 3.20, where $\vec{S}_{S_q}$ is our vector of values for each date.

$$\hat{R}_s = \frac{1}{Q} \sum_{q=1}^Q \vec{S}_{S_q} \vec{S}_{S_q}' \quad (3.20)$$

From this covariance matrix, a Principal Component Analysis (PCA) can be calculated using less data than would have been otherwise possible if the data wasn't converted to the spectral domain. Next, the correct number of Principal Components (aka: Empirical Orthogonal Functions (EOFs) or Eigenvalues / Eigenvectors) can be picked to reach a desired level of variance explained.

3.3.2 Implementation

3.3.2.1 PRINCIPAL COMPONENT ANALYSIS OF A COMPLEX COVARIANCE MATRIX

Like before, this section of the code had originally been written in Matlab, and we now had to find a Java library that could perform a principal component analysis on a covariance matrix. Matlab (a primary numerics language) can perform this operation with a call to `pcacov`. Java had many libraries available that could perform a principal component analysis, but none of the major ones worked for our use case because we've converted our data from the spatial domain to the spectral domain we are now in complex space. Out of the major Java numeric libraries available, not a single one could perform a principal component analysis on a complex matrix.

Since the covariance matrix was available, we could instead calculate a PCA via an eigen-decomposition of the matrix. Unfortunately, we discovered the exact same issue: no library could be found that could do this calculation on a complex matrix. Since this eigen-decomposition is occurring in the spectral domain, there are complex numbers everywhere.

Thankfully, an obscure Java library was discovered that could actually do this successfully `ojAlgo` [53]. The library is difficult to use, but it does work correctly. Unfortunately it doesn't order the eigenvalues or make sure the eigenvalues returned were positive, so some work had to be done to re-order the eigenvalues and eigenvectors and fix their signs. To be able to use this `ojAlgo` library, I simply needed to add the following:

```
<dependency>
```

```
    <groupId>org . ojalgo</ groupId>
```

```

    <artifactId>ojalgo</ artifactId>
    <version>43.0</ version>
</ dependency>

```

While the library was far from perfect, it did solve the immediate problem of processing a PCA in complex space. When we began to scale the size of the matrix up, we ran into major problems with this library taking a very long time when performing the eigen-decomposition. When $Q \geq 70$, Java seems to be thrashing at memory because of the garbage collector processes. We could at least continue moving forward, but high Q truncations were out of the question.

3.3.2.2 DR ANGUS' RECOMMENDATION

After discussing this Complex PCA issue, Dr Angus suggested a very helpful resource that had a potential solution to this issue[8]. You could augment a real matrix with complex information and calculate the PCA on this augmented matrix that had effectively double the rows and columns. This technique would have worked perfectly except that when I processed this matrix through Java PCA implementations faster than ojAlgo, they would automatically restructure the eigenvalues and vectors from largest to smallest mixing up which columns were the complex and real parts of the matrix. We also can't ignore the fact that the matrix was now four times as large as the original, negating a lot of our dimension reduction gains from converting from spatial to spectral.

Since the ojalgo library didn't re-order the matrix it could have potentially solved this issue, if it could have processed. Another option could have been to rewrite one of these Java libraries to remove the auto ranking it performed on the eigenvalues and eigenvectors and re-rank it myself after converting it back to the complex domain. This was not a path I wanted to go down because that would have complicated the code base a great deal, and the risk of error on the rewrite of someone else's library could have been a huge mess.

In testing on smaller matrices, this technique was working correctly, but unfortunately it was not usable for this particular use case. Another issue is that even the other Java libraries had trouble calculating the PCA on this now even larger matrix. Even if this method were successful, the quadrupling of the matrix size increased what was already an issue.

3.3.2.3 JBLAS

With no Java native library in sight that could effectively process a PCA on a large covariance matrix, I began looking for one outside of Java that I could instead call via JNI. I found a JNI wrapper already existed that's tied directly to BLAS, LAPACK, and ATLAS.

The Basic Linear Algebra Subprograms, or BLAS, itself is more like a specification with many different implementations of it. The reference implementation is written in Fortran and involves simple algebraic operations such as addition, subtraction, and dot product, for a total of 38 operations. One of the main goals behind this was to provide the building blocks for higher level operations [29]. Higher operations like those implemented in Linear Algebra PACKage or LAPACK. LAPACK picks up where BLAS left off and solves systems of simultaneous linear equations, least-square solutions of overdetermined systems of equations, and eigenvalue problems. It uses the reference BLAS implementation for the lower level operations [2].

With the advent of modern computing, many vendors provided optimized implementations of BLAS for their specific architectures for added performance gains. These gains were wildly inconsistent across different architectures and operating systems. To solve this Automatically Tuned Linear Algebra Software (ATLAS) tried automatically generate highly efficient basic linear algebra routines [58].

The JNI wrapper library that I discovered was called jblas which uses ATLAS routines when available and falls back to the LAPACK implementation when not. The library provides an easy to use high level API to call these native library implementations [7]. A great benefit of the library is that it comes bundled with pre-compiled binaries and selects the correct one to use and load. There is actually no need to compile the native libraries themselves as they're already part of the package. Jblas can easily be added to a Maven project via the following addition to your `pom.xml` file:

```
<dependency>
    <groupId>org.jblas</groupId>
    <artifactId>jblas</artifactId>
    <version>1.2.4</version>
</dependency>
```

If you want to exploit multi-threading and other potential speed gains you must compile your own version of these native libraries. For this research, I chose to exploit the multiple threads available on my six core system. During the library initialization phase, jblas first checks to see if a native implementation is available and if none are found, it will use it's pre-compiled versions. Even though my system uses the manually compiled version, the code is portable enough to work on any computer because it will fall back and use it's bundled pre-compiled versions. In the end, the speed gains from compiling my own version didn't amount to a huge increase. While I did notice that jblas would occasionally take advantage of the multiple cores, it was only during certain portions of the run. The particular calculations happening behind the scenes may not have proven that advantageous to multiple threads.

After some thorough testing of the library, I decided that it was the right way to go because it performed faster and it could handle complex numbers without difficulty. BLAS, LAPACK, and ATLAS are the defacto standards in linear algebra operations: they're what systems like Matlab use under the hood. Jblas allows us to use these libraries in a language I becoming more accustomed to.

As there always is, there was a slight problem: there was no working implementation of a principal component analysis, or to be more precise a PCA for a covariance matrix. Thankfully, Singular Value Decomposition (SVD) was available and it is possible to calculate a PCA from a covariance matrix through the use of SVD. In the end, we just needed the eigenvalues and eigenvectors.

Although I'm not going to display the code for all of the small helper functions, the overall logic to do this ended up being fairly simple:

```
private void ProcessPcaCov(ComplexDoubleMatrix X_cov) {
    // Get eigenvectors and eigenvalues from SVD
    ComplexDoubleMatrix[] SVD = fullSVD(X_cov);
    ComplexDoubleMatrix V = SVD[0];
    ComplexDoubleMatrix S = SVD[1];
    ComplexDoubleMatrix U = SVD[2];

    // Get the Maximum Absolute value per column of V
    ComplexDoubleMatrix V_abs = Abs(V);
    Triplet<ComplexDoubleMatrix, int[], int[]> V_absMax = ColumnMax(
        V_abs);
    ComplexDoubleMatrix test = V_absMax.getValue0();
    int[] V_absMaxRowLocation = V_absMax.getValue1();
    int[] V_absMaxColLocation = V_absMax.getValue2();
    ComplexDoubleMatrix maxPerColumn = GetRowColValues(V,
        V_absMaxRowLocation, V_absMaxColLocation);

    // Get the new Column sign
    ComplexDoubleMatrix absMaxPerColumn = Abs(maxPerColumn);
    ComplexDoubleMatrix colSign = maxPerColumn.divi(absMaxPerColumn);
    ComplexDoubleMatrix coeff = RowMultiplyExpansion(V, colSign);

    // Calculate variance Explained
    double eigenValueTotal = S.sum().real();
    double[] varExplained = new double[S.length];
```

```

for (int row = 0; row < S.length; row++) {
    varExplained[row] = S.get(row,0).real() / eigenValueTotal;
}

// Now save all the variables
this._eigenValues = S;
this._eigenVectors = coeff;
this._varianceExplained = varExplained;
}

```

By this point we were now calculating the PCA at least as fast, if not faster, than the original Matlab implementation.

3.3.2.4 SPEED INCREASE ATTEMPT

Back in section (3.2.2.7) we discussed some refactoring and exploited the symmetry that exists thanks to equation (3.12). When we further inspected the results from running a PCA on the “whole” harmonic, we noticed that the symmetry was still present on the eigenvectors after we expanded them out of their columns and back into their individual harmonic forms. We could use that same trick during the PCA step to shrink the size of the covariance matrix.

This simple solution could have alleviated some of the library issues we dealt with in the previous section; by the time we realized we could exploit symmetry it was too late. Jblas was now fully implemented and ran faster than ojalgo anyways. There was no reason to go back as jblas was the better solution regardless.

The speed increase here was significant as well. What normally would have taken a 3 to 4 days to calculate the complete PCA for each month of the year would now take less than a day.

However, running it on the full harmonic (including both positive and negative values) vs the half one (include only the positive) didn’t give the same results. We decided to keep this optional because the end results would be slightly different between the two routes. All of the end results in this research currently use the full harmonic option.

3.3.3 Data and Processing

Processing of the data was done using NOAA’s 20th Century Reanalysis V2c dataset [11]. The V2c dataset contains derived surface temperature values spaced out on grid boxes across the globe. They are distributed evenly across the longitude at 1.875° and using Gaussian Quadrature (GaussLegendre quadrature) for the latitude. This spacing will actually works to the formulas advantage, as the method uses Gaussian Quadrature.

The dataset contains monthly values from 1851-01-01 to 2014-12-01, with 94 latitude points and 192 longitude points for a total of 18,048 grid points.

First, all of the data was grouped by their respective months and each date was converted from spatial to spectral. Each date was placed in it's own column and the data eventually converted into the form seen in equation (3.20). From here, the data was passed through the principal component analysis code and results inspected.

Upon converting this data to spectral form, something very interesting was discovered when the principal components from the data set were inspected. There was an issue because the first EOF contained too much information (too high a variance explained), and subsequent EOFs, not enough (Fig. 3.5). After a bit of research (Figures 3.5 - 3.7) it was determined that it was best to use a normalized version of the dataset ($Z = \frac{X-\mu}{\sigma}$), where each grid point had been normalized with data from whatever baseline period was selected. The summary data from this is found in table 3.10. The last entry (the one with a 20 eigenvalue count) was selected. It not only had the lowest eigenvalue count, but when inspecting the EOFs, it made the most sense. Do note that these variance explained cutoffs were generated before we we increase the Q value to 102. These were based on a smaller number.

While using normalized EOFs does seem like the correct decision, there are some problems with it. We eventually have to back out of the normalization when we do the processing in the next section. This backing out led to other issues that will be discussed there. For now, assume that both are options.

Table 3.10. 95% Variance Explained for each method

Dates	Type	Number of Eigenvalues to 95% variance explained
1871-2008	Not Normalized	44
1871-2008	Normalized	33
1961-1990	Not Normalized	23
1961-1990	Normalized	20

To verify the results of the EOFs and make sure they're within reason, we used Eq. (38) from [45] to check the values of the eigenvalues.

$$\frac{1}{\Gamma R^2} \sum_{\gamma=1}^{\Gamma} \sum_{i=1}^I T^2(\hat{r}_i, \gamma) \text{Area}_i \approx \sum_{j=1}^J \lambda_j \quad (3.21)$$

Where Equation (3.21) has R as the radius of the earth, i is an individual grid box, T is the temperature for grid box r at time γ and where λ is the j th eigenvalue. Since the time period of 1961-1990 is being used from the normalized data, there are a total of 30 (Γ) years being summed over, with $J = 20$ eigenvalues.

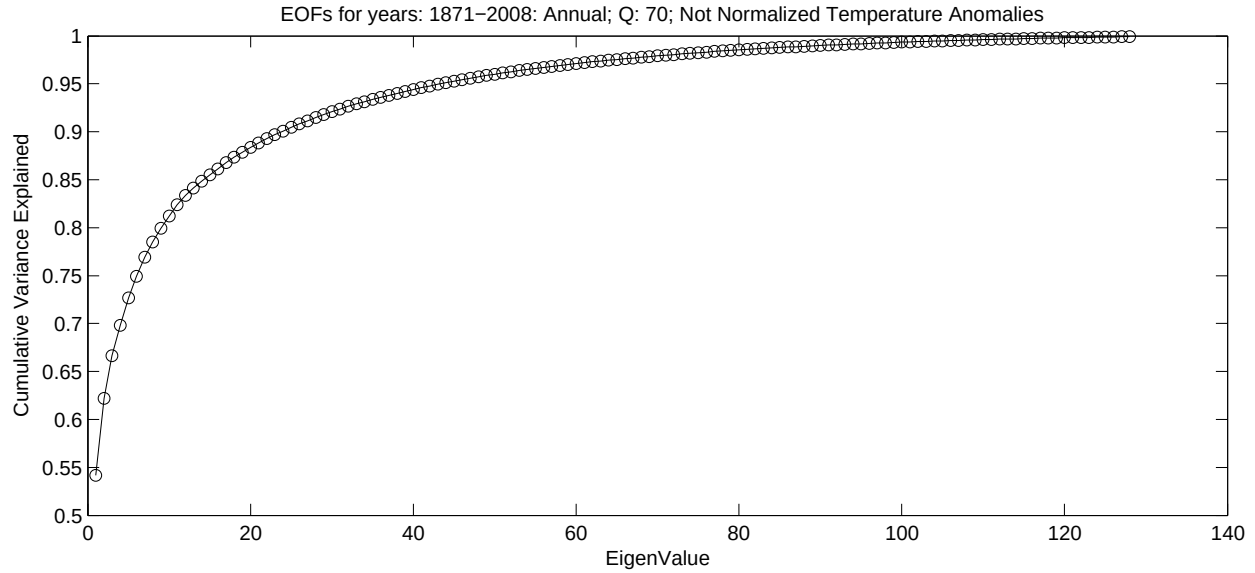


Figure 3.5. Cumulative Variance Explained for Non-Normalized Data.

The left hand side of the equation equals 0.8940 and the right hand side is 0.8353, proving that the eigenvalues are legitimate. There are a few errors on the upper bound of this calculation. In [45], this formula worked as the entire area of the globe was used for the calculation of the EOFs. In this research, a thin strip of area around the poles is omitted due to their the poles singularities; this makes using the radius of the Earth a mistake. However, as a quick check to see if the values are within reason, it more than accomplishes the task.

The end result of what's been calculated with all of these EOFs is the global climate field that will be required in the next section.

3.4 OPTIMAL WEIGHTS

3.4.1 Background

This section builds on the work of the previous two. At this point we have eigenvalues and vectors in spectral form that we will use to help optimally weigh weather stations; not all of the stations available will be used for this analysis. Instead, a selective algorithm was written to select stations for analysis and inclusion into the final calculation.

First, a baseline period is selected (ie: 1961-1990) and then for each of the 12 months we only allow stations that have a certain minimum number of observations available to be eligible for selection (ie: all 21 observations). Calculation are processed on a per month basis and the results are later combined at the end. It's important to do the calculations on a monthly basis as processing it annually made us lose too much information.

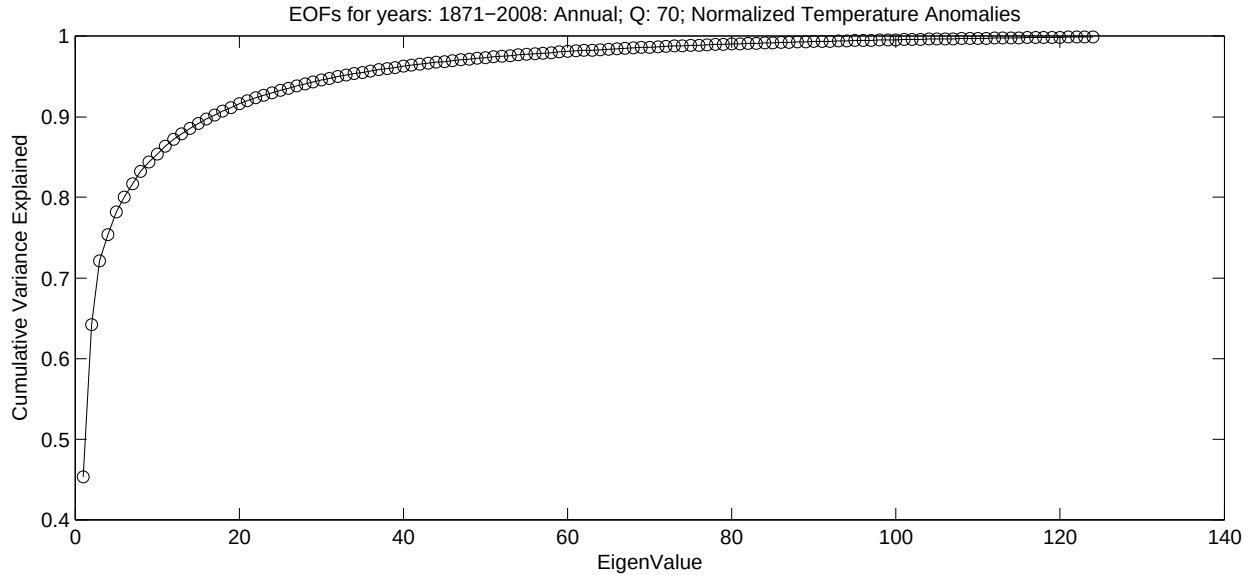


Figure 3.6. Cumulative Variance Explained for Normalized Data.

For each of these stations, we calculate a baseline mean and variance based off of the selected baseline, and then calculate a difference between this mean and a stations observed value during each of it's dates available. The variance was intentionally excluded from the calculation so we could later process the data normalized or not. While we did notice in the previous section it would eventually be better to normalize the data, this wasn't yet known to be true yet.

Next, a technique that became known as the “hole punch” selection criteria was used. For each year of the GHCN dataset, all stations that had passed the first criteria were rank ordered by how much total history they had. Now with values rank ordered, we remove the first station from the list and put it in a “picked” bin. The algorithm next goes through all of the other stations left in the ranking and removes any station within 500 km of the station just binned. The process then goes back to ranking and picks the next highest ranked station and repeats the process again until no stations are left on the ranked list. This is done each year, so potentially different stations are picked for every year of the dataset.

This is done to ensure only high quality stations with a consistent and proven track record are included in the analysis. It also helps because trying to run this process on too many stations slows the computation times down quite a bit. The pseudo code for the algorithm can be found in figure (3.8).

Next, if desired, we could apply the same normalization procedure ($Z = \frac{X - \mu}{\sigma}$) that was used on the NCEP data (normalizing the NCEP data is also optional) is used so the station data becomes compatible with the global climate field. It was later discovered that this “grooming”

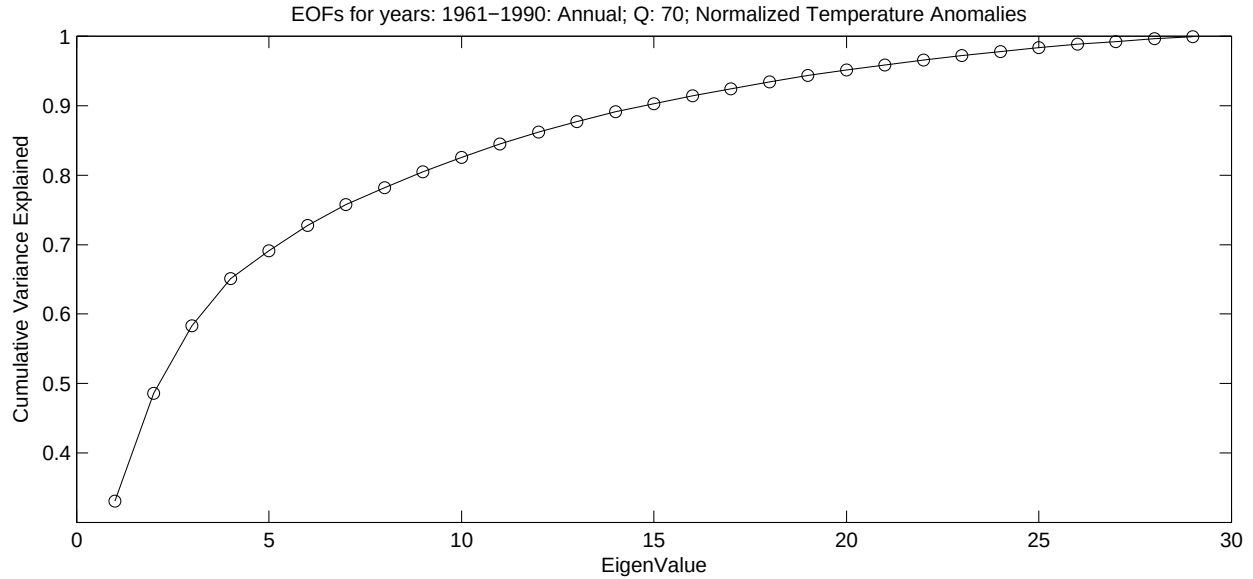


Figure 3.7. Variance Explained for Normalized Data.

procedure would cause issues; we did notice that values generated without this normalization step would be much better.

Setting these problems aside, the global climate field calculated in the previous section (3.2) can now be combined with the GHCN stations that have been picked. To accomplish this, mathematics from Shen 1997 [66] was used to calculate a one number temperature value for each available date. Mainly equation:

$$\hat{S}_{kl}(t) = \sum_{j=1}^{N_t} S(\vec{n}_j, t) Y_{kl}^*(\vec{n}_j) w_j^{(kl)} \quad (3.22)$$

Figure 3.8. “Hole Punch” algorithm

Data: GHCN available stations

Result: “Best” Stations

Remove Stations that don’t have at least 21 years of data in the baseline period;

for Each available date **do**

 Rank order all of the stations from most data to least data;

while We still have available stations **do**

 “Pick” the first station in the list and remove it from the list;

 Remove stations from the list that are within 500 km of the “picked” station;

end

 Save all “Picked” stations as the stations we will use for this particular date;

end

Where j is a particular station, N_t is the total number of stations at time t . $w_j^{(kl)}$ is the area of a region on the surface of a unit sphere around the point $\vec{n}_j$. These areas, or weights, are constrained by:

$$\sum_{j=1}^N w_j^{(kl)} = 4\pi \quad (3.23)$$

The goal is to select the optimal weights so that the estimated $\hat{S}_{kl}(t)$ will be as close as possible to the actual $S_{kl}(t)$. We want to find the weights that will minimize the mean squared error of:

$$\varepsilon_{(kl)}^2 = \langle |S_{kl}(t) - \hat{S}_{kl}(t)|^2 \rangle \quad (3.24)$$

The Shen 1997 paper also contains a formula for calculating these optimal weights ($w_j^{(kl)}$) that did not give us much success in our research attempts. Instead, we turn to an earlier paper that contains the following solution to calculating the weights [25] in equations 3.22, 3.23:

$$\sum_n \lambda_n \Re \left[\Psi_n(r_j) Y_l^{m*}(r_j) \sum_k w_k \Psi_n(r_k) Y_l^m(r_k) \right] - \Lambda = \sum_n \lambda_n \Re [\Psi_m^{*(n)} \Psi_n(r_j) Y_l^{m*}(r_j)] \quad (3.25)$$

for $j = 1, \dots, N_{\text{net}}$. To make the equation 3.25 a bit more palatable, we restructure everything into the solution of an inverse problem: $A_{(N+1) \times (N+1)} x = b_{(N+1) \times 1}$ where:

$$A = \begin{pmatrix} & -1 \\ B_{N \times N} & \vdots \\ & -1 \\ 1 & \dots & 1 & 0 \end{pmatrix} \quad (3.26) \quad x = \begin{pmatrix} w_1^{(lm)} \\ \vdots \\ w_N^{(lm)} \\ \Lambda \end{pmatrix} \quad (3.27)$$

$$b = \begin{pmatrix} C_{N \times 1} \\ 4\pi \end{pmatrix} \quad (3.28)$$

Where:

$$B_{N \times N} = \sum_n^\infty \lambda_n \Re [\Psi_n(\vec{r}_j) \Psi_n(\vec{r}_k) Y_{lm}^*(\vec{r}_j) Y_{lm}(\vec{r}_k)] \text{ For } j, k = 1, \dots, N \quad (3.29)$$

$$C_{N \times 1} = \sum_n^{\infty} \lambda_n \Re \left[\Psi_{lm}^{(n)*} \Psi_n(\vec{r}_j) Y_{lm}^*(\vec{r}_j) \right] \text{ For } j = 1, \dots, N \quad (3.30)$$

With the above equations, it was time to solve for x , the weights. Depending on what stations were being selected the solution was sometimes unsolvable because of the matrix being rank deficient. Rank deficiency is the reason we opted to use this solution over the previous one. To solve this we added a small amount of “noise” to all but the last entry of the diagonal elements in matrix A (equation 3.26). This “noise” was simply a very small number: $1e^{-10}$.

By adding a small number, the rank deficiency of the matrix would no longer be an issue, and the number would be so small that it shouldn’t effect what the actual weight “would” have been by a negligible amount. The problem was: how small should the number be, and how will this number effect the weights.

To experiment, we calculated how weights would be effected by changes in this noise parameter: the hope was that as the number gets smaller, the values should “converge” to what the actual weights should be. We processed what the weights would be if we had selected stations from the Angell-Korshover network [3]. These 63 station locations have shown to be well-distributed and have been used in other parts of this research. Table 3.11 shows a sample of these results with varying “noise” values. $1e^{-10}$ ended up being the minimum because values less than this could still give us singular matrix issues when solving for the weights. The second table 3.12, assumes that $1e^{-10}$ is the minimum and shows the sum of squared errors at varying noise levels.

While it would have been possible to use a value larger than the minimum (as the changes in the weights are minimal across many of these numbers) the choice had a negligible effect on the computations duration; the lowest number possible was selected. In one rare instance, the value of $1e^{-10}$ was too low and caused a singular matrix issue. Thus, this process was wrapped into a “while loop” that would increase the number for each failed attempt until it reaches $1e^{-4}$. If the matrix is still singular up until this, the code will then error out and fail. Since this new logic has been implement there have been no instances of this issue.

With the weights now in hand, we could now move forward with the rest of the calculations.

Table 3.11. Differing “noise” value effects on weights

Weight	1.00E-10	1.00E-09	1.00E-08	1.00E-07	1.00E-06	1.00E-05	1.00E-04	1.00E-03	1.00E-02	1.00E-01	1	1.00E+01	1.00E+02
1	0.8326727101	0.8326725643	0.8326715116	0.8326609703	0.8325555954	0.8315054432	0.8213480596	0.7438507374	0.4987208926	0.2755062904	0.2093485667	0.2004894186	0.1995688919
2	0.6582038951	0.6582038523	0.6582026564	0.658201723	0.6581823858	0.6579885921	0.6560113237	0.6342149423	0.4738235857	0.2589466262	0.2063415807	0.2001654044	0.1995562406
3	0.3964461748	0.3964461397	0.3964458645	0.3964431326	0.3964158342	0.3961448026	0.3936187918	0.3797986471	0.3697562043	0.2733021138	0.2096945828	0.2005293356	0.1995729341
4	0.011290443	0.0112904309	0.0112907012	0.0112934447	0.0113208723	0.0115941613	0.0142321225	0.0338069202	0.0887724186	0.1651192991	0.1951661118	0.1990257356	0.1994220475
5	0.6567484359	0.6567483573	0.6567477254	0.6567414322	0.6566785163	0.6560507888	0.6499127151	0.5994246211	0.40629608	0.235378021	0.2031687246	0.1998358185	0.1995031524
6	0.5254307934	0.5254307896	0.5254305843	0.5254285126	0.525407797	0.5252009819	0.5231661631	0.5054595554	0.4128556251	0.2611024611	0.2074562336	0.2002918206	0.199549406
7	0.38627226	0.3862722486	0.3862723932	0.3862738552	0.3862884674	0.3864335839	0.3877884452	0.3946992403	0.359308852	0.2460371919	0.2053233136	0.2000681052	0.1995265587
8	0.1149779731	0.1149779859	0.114978308	0.1149815263	0.1150136972	0.1153340518	0.1184068246	0.1395847925	0.1666315786	0.1836227031	0.1973574222	0.1992492838	0.199444448
9	0.1043428305	0.1043428866	0.1043434675	0.1043492593	0.1044071483	0.1049831937	0.1104728073	0.1472439828	0.2043719199	0.2029711168	0.1998261093	0.1995016251	0.1994697356
10	-0.1734022503	-0.1734021887	-0.1734017128	-0.173396954	-0.1733493718	-0.1728741543	-0.1681828102	-0.1270603652	0.0480459458	0.1856603727	0.1989633908	0.1994320507	0.1994629591
54	-0.7416921838	-0.7416920852	-0.7416911118	-0.7416813906	-0.7415841983	-0.7406141628	-0.7310963338	-0.6498265754	-0.2813919357	0.1067090082	0.1894834288	0.1984633018	0.1993658698
55	0.348603752	0.3486036877	0.3486031392	0.3485976633	0.3485429282	0.3479978859	0.3427667579	0.3050617129	0.2134680166	0.1931835462	0.1985322891	0.1993692828	0.199456473
56	0.2100646457	0.2100645977	0.2100644615	0.2100631006	0.2100495034	0.2099145471	0.2086608619	0.2020571504	0.2045633775	0.2050847196	0.2003621336	0.199560983	0.1994757334
57	0.1511771524	0.1511771406	0.1511771192	0.1511769135	0.1511748681	0.1511555466	0.1510659675	0.1551362396	0.1901572618	0.2063753614	0.2008033332	0.1996105629	0.1994807499
58	0.4081451513	0.4081452027	0.4081455012	0.4081484626	0.4081780534	0.4084718612	0.4112098208	0.4252649193	0.3791201715	0.246074361	0.2050673812	0.2000379748	0.199523497
59	0.3262377299	0.3262377516	0.3262378306	0.3262386345	0.3262466672	0.3263262984	0.3270573001	0.330439196	0.3166632828	0.2440848464	0.2056872887	0.2001147075	0.1995313288
60	0.155271161	0.1552711837	0.1552714972	0.1552746305	0.1552930524	0.1556169695	0.1585284096	0.1750394998	0.1749412309	0.1814897747	0.1969109655	0.1992008135	0.1994395602
61	0.2462939607	0.2462939721	0.2462939557	0.2462938025	0.2462922703	0.2462770741	0.246135372	0.2446817797	0.2216552838	0.2022215914	0.2000228594	0.1995285242	0.1994752065
62	0.527940373	0.5279403008	0.5279394136	0.5279350524	0.527841657	0.5269578658	0.5185791508	0.4637143482	0.3476953473	0.2416371261	0.2050175573	0.2000412364	0.1995529133
63	0.6084933218	0.6084933466	0.6084931881	0.6084915916	0.6084756142	0.6083149767	0.6066293396	0.5860654188	0.4469464701	0.2759583896	0.2109170911	0.2006775061	0.1995860421
64	2.78E-11	2.78E-10	2.78E-09	2.78E-08	2.78E-07	2.78E-06	2.77E-05	0.0002692678	0.0023117089	0.0197758063	0.1986642934	1.9937489842	19.945950787

Table 3.12. Sum of squared errors

Weight	1.00E-10	1.00E-09	1.00E-08	1.00E-07	1.00E-06	1.00E-05	0.0001	0.001	0.1	1	10	100
1	0	2.13E-14	1.44E-12	1.38E-10	1.37E-08	1.36E-06	0.0001282477	0.0078893428	0.115238164	0.310344192	0.3885329877	0.399655714
2	0	1.83E-15	5.70E-14	4.72E-12	4.63E-10	4.64E-08	4.81E-06	0.0005754699	0.0339960985	0.1594063668	0.2041795512	0.209799259
3	0	1.23E-15	9.63E-14	9.25E-12	9.21E-10	9.08E-08	7.99E-06	0.0002771402	0.0007123545	0.0151644598	0.0348761571	0.0383834079
4	0	1.47E-16	6.67E-14	9.01E-12	9.26E-10	9.22E-08	8.65E-06	0.0005069917	0.0060034565	0.023663317	0.0338102616	0.0352445401
5	0	6.18E-15	5.05E-13	4.91E-11	4.89E-09	4.87E-07	4.67E-05	0.0032860197	0.0627263825	0.1775530256	0.2057345545	0.2087691399
6	0	1.44E-17	4.37E-14	5.20E-12	5.29E-10	5.28E-08	5.13E-06	0.0003988503	0.0126731685	0.0698694672	0.1011073749	0.1057153516
7	0	1.30E-16	1.78E-14	2.54E-12	2.63E-10	0.000000026	2.30E-06	0.000071014	0.000693868	0.0196658743	0.0327425212	0.0346719872
8	0	1.63E-16	1.12E-13	1.26E-11	1.28E-09	1.27E-07	0.000011757	0.0006054956	0.002668095	0.004712099	0.0067863736	0.0071016538
9	0	3.15E-15	4.06E-13	4.13E-11	4.14E-09	4.10E-07	3.76E-05	0.0018405089	0.0100058187	0.0097168899	0.0091170565	0.0090491281
10	0	3.80E-15	2.89E-13	2.81E-11	2.80E-09	2.79E-07	2.72E-05	0.0021475703	0.0490393036	0.1289259673	0.1386561707	0.139005416
54	0	9.73E-15	1.15E-12	1.16E-10	1.17E-08	1.16E-06	0.000112272	0.00843929	0.2118763184	0.7197845826	0.8670880214	0.8883892371
55	0	4.14E-15	3.76E-13	3.71E-11	3.70E-09	3.67E-07	3.41E-05	0.0018261667	0.018261667	0.0241554404	0.022521444	0.0222709268
56	0	2.30E-15	3.39E-14	2.39E-12	2.29E-10	2.25E-08	1.97E-06	0.000006412	0.000030264	2.48E-05	0.0001103269	0.0001121251
57	0	1.40E-16	1.10E-15	5.71E-14	5.22E-12	4.67E-10	1.24E-08	1.57E-05	0.0015194489	0.0030468423	0.0024627578	0.0023332375
58	0	2.63E-15	1.22E-13	1.10E-11	1.08E-09	1.07E-07	9.39E-06	0.0008424495	0.008424495	0.0412405807	0.0433085969	0.0435229947
59	0	4.72E-16	1.01E-14	8.18E-13	7.99E-11	7.84E-09	6.72E-07	1.77E-05	0.00009167	0.0067490963	0.0145524089	0.0160545121
60	0	5.16E-16	1.13E-13	1.20E-11	1.21E-09	1.20E-07	1.06E-05	0.0003869116	0.0006874157	0.0006874157	0.0019298144	0.0019508475
61	0	1.31E-16	2.47E-17	2.50E-14	2.86E-12	2.85E-10	2.52E-08	2.60E-06	0.006070644	0.0019423737	0.0021410148	0.0021870061
62	0	5.21E-15	9.20E-13	9.70E-11	9.74E-09	9.65E-07	8.76E-05	0.0041249823	0.0324882693	0.0819695492	0.1042791449	0.1075178438
63	0	6.11E-16	1.79E-14	2.99E-12	3.14E-10	3.18E-08	3.47E-06	0.0005031008	0.0260973853	0.1105794812	0.1580668593	0.1663137396
64	0	6.26E-20	7.58E-18	7.72E-16	7.73E-14	7.73E-12	7.67E-10	7.25E-08	0.000005344	0.0003910825	0.0394675015	3.9750350119
Error	0	1.30E-13	1.02E-11	0.0000000001	1.04E-07	1.03E-05	0.0009605065	0.0564052504	0.9120067009	2.9319326832	3.681345611	7.7106415509

3.4.2 Implementation

3.4.2.1 DISTRIBUTED COMPUTING VIA SPARK

Up until now we haven't had to exploit distributed processing as the problems haven't been very large.

In the previous chapter we had to limit the problem to solving only the first few harmonics of 0, as solving larger harmonics would take too long. For generating a one number overall climate value across the globe that wasn't an issue, but if we want to generate all of the harmonics and thus a map of all of the temperatures, it would take $\frac{(Q+1)^2+Q+1}{2}$ (3.13) more computations than it originally took. For $Q = 102$ the calculation would take, at a minimum, up 5,356 times longer to process. This calculation would obviously take too long to calculate on a single computer. Which is where cloud computing comes in.

As has been stated previously, we didn't accidentally select Java over other languages: we chooses it because it is compatible with Apache Spark which is available on cloud computing providers like Amazon Web Services.

Spark, an open-source cluster-computing framework, mainly deals with the processing of large amounts of data over a cluster[65]. The challenge this research faced is slightly different, and focuses more on computational issues. This doesn't mean that Spark isn't a good fit for the processing; as it's generalized enough to be a good fit for many different types of workloads [63].

The concept of using cloud computing resources for distributed processing research isn't an entirely new idea [17], it's just underutilized in academic related research. To use Spark with your Java code, you simply need to add the following to your "pom" file:

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql_2.11</artifactId>
    <version>2.2.0</version>
    <type>jar</type>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-minicluster</artifactId>
    <version>2.7.3</version>
<scope>test</scope>
</dependency>
<dependency>
```

```

        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-common</artifactId>
        <version>2.7.3</version>
        <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-hdfs</artifactId>
    <version>2.7.3</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-mapreduce-client-core</artifactId>
    <version>2.7.3</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-mapreduce-client-common</artifactId>
    <version>2.7.3</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-aws</artifactId>
    <version>2.7.3</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-mllib_2.11</artifactId>
    <version>2.2.0</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-core_2.11</artifactId>

```

```

    <version>2.2.0</version>
    <scope>provided</scope>
</dependency>

```

There is a new argument here: “scope” and its “provided” value. Since Spark code will not be running on your computer, but on a host with access to a Spark cluster, it’s assumed that all the Spark libraries have been installed on that cluster and that the code should be using those versions, not the versions in your code base.

3.4.2.2 LOADING THE GHCN STATION DATA

To load the GHCN station data, a Spark process was written that would load both the inventory and monthly data. Writing a complete Spark job for this may have been overkill, as the data size is very small, but this was done in case we ever need to load a larger dataset in the future. For example, since we were only looking at monthly GHCN readings, if at some point we wanted to upgrade to hourly readings, then we would already have a code base to handle it. The code has been partially future-proofed if we decide to increase the temporal resolution. The aggregated results from this procedure are then serialized via Java so they can later be reloaded directly into their original class structure in Java.

3.4.2.3 LOADING JAVA SERIALIZED OBJECTS IN SPARK

Throughout each step, the intermediary results have been saved by simply serializing the Java classes that were written to store the objects. While this simplifies reloading the work into memory for subsequent programs to load, it does not work with loading data in Spark.

Spark is intended more for data manipulation in a columnar format, hence it’s setup to load data from file formats such as CSV. While possible, it would be highly inefficient to try and restructure the data we have at this point into this kind of file format. Spark does offer a way to distribute data that is already in memory out to the cluster; exploiting that is how we submit data to the cluster.

Everything is first loaded into memory and the data divided into two categories:

- Global variables
- Variables specific to a particular run

There are a number of ways to divide the data. We opted for keeping the complete serialized objects available to each node and submitted them as global variables and simply submitted the Year, Month, k , and l values for the run details. That isn’t the only way to do it; we

could run it at the Year and Month level, processing all the k and l values during a single iteration instead of spreading it out.

Spark works by distributing “rows” of information out to each node on the cluster for processing. Each of these “rows” is usually a self contained problem that a solution can be calculated on. The idea is to pass all of the necessary input variables into an individual “row” to process it correctly. Since our dataset is small, we can pass universal variables that are assumed to be needed by all runs by passing them to a constructor instead, making them available to every node that will be processing the data. We can simply lookup the relevant variables from this universal data set if and when they’re needed.

Our goal is to distribute out the processing of the weights and gather the accompanying harmonic associated with that input. So, by assigning out the Year, Month, k and l values as part of this “row” we will end up doing the processing at that specific level, returning an individual harmonic for each of these distributed calculations.

3.4.2.4 SPARK JAVARDD

Coding in modern Spark is traditionally done in through the DataFrame API. In Java this format is more commonly known as having data in a `Dataset<Row>` structure. This format has been optimized specifically for running efficiently through Spark [62], but at the cost of flexibility. Only SQL-like tasks can run through Spark when using this format which makes it useful for simple data manipulation, but it won’t work for running more complicated processes [5].

To distribute out custom code logic to a Spark cluster, we instead need to write the process through the RDD API; in Java this has the `JavaRDD` structure. RDD stands for “Resilient Distributed Datasets” and was the original format that Spark was built off of [64]. It allows a user to do anything they want when code is run on each node of the cluster. The logic just needs to be spelled out in a user defined inherited `FlatMapFunction` class.

The Spark process was written so that it could easily pick up where it left off, so jobs are submitted to the cluster in chunks by an dates. An individual date is processed along with all the k and l harmonics. When a date finishes the final calculated harmonic is saved to disk. This insures that if some catastrophic event were to take place on the server (power outage, server crash, etc...) the work calculated to that point would not be lost. The process would load the results calculated thus far and pick up where it left off. Depending on the size of the cluster, Q and truncation selected, this process could still take a very long time to run from beginning to end. Having this feature available also makes it easier to stop and start the job if other more important tasks comes up.

3.4.2.5 CODE OPTIMIZATIONS

Similarly to what was done in section 3.2.2.7, a good way to increase the speed of these calculations was to avoid using Java matrix libraries. All of the work done in the previous section could be used in this process as well.

If we only wished to calculate an overall global anomaly, there was no need to calculate the entirety of all the k and l harmonics. Since the first harmonic of $k = 0, l = 0$ is proportional to the integral of the entire spherical surface [4]. Thus, converting this single harmonic to spatial, weighing by the individual grid boxes, and then calculating an overall global value should give the same value whether the first, or any number of harmonics is calculated. This implies that we only need to calculate the first harmonic to get the actual value of all the harmonics, if we only care about generating a one number summary of the climate. Depending on the Q truncation used, this is a monumental savings in the number of calculations. Calculation of the full harmonic is important if we need a full map of the global temperature. The problem is that this issue becomes too large for a single computer to calculate.

Now, even though the calculation speed has increased a lot by simply calculating the first harmonic, there are many dates and different station arrangements that need to be calculated. To further increase the computations, two things can be done:

- Store pre-processing weights
- Multi-thread the operation.

Since weights are purely based off of the monthly PCA data and the location of the stations, if we happen to get the exact same set of station locations for the same month, we can use the same weights from that previous run. This requires the system to save the weight information for a particular set of stations; the code just needs to check if the weights have already been populated prior to running the logic. While this event does rarely occur, it will save us a lot of time when it does

Multi-threading requires us to divide the operations into something that can be multi-threaded. Since we're now only running the calculation on the first harmonic of $k = 0, l = 0$, it would be pointless to split the processing there. Instead, the processing can occur at the date level which allows the system to calculate multiple dates at once. While it is possible that two dates may end up having the exact same stations selected, with smart coding, it can be ensured that these similar dates are not processed at the same time. If they were, it would affectively be calculating the same weights and tying up multiple threads to do that operation.

3.4.2.6 SPHERICAL HARMONIC MULTIPLICATION

Table 3.13. Verification of first harmonic calculation only

Complete Q	$k = 0, l = 0$ Only	Complete Q	$k = 0, l = 0$ Only
2.5446572245	2.5367602596	-0.0447097455	-0.053311701
2.7188421053	2.7043893185	0.6534455003	0.6391982283
-2.0984522616	-2.1124587098	-1.3584152565	-1.3644977599
1.6454496017	1.6392822054	-2.2542688837	-2.26268798
0.8028336446	0.7939785403	2.2094227086	2.1991721043
0.4747035052	0.4678516089	2.0336564094	2.0130544868
0.0418023078	0.0244862194	1.1036916059	1.0931856064
2.3357084453	2.3139266403	2.1998658737	2.1914803945
1.7960698439	1.7856846357	2.8147569802	2.8052924641
2.4046844458	2.3935136522	2.3733468515	2.3506389357

For the $Q = 0$ truncation, we wanted to numerically double-check this was true by:

- Generating some random data in the spectral domain and converting it to spatial. By starting in the spectral domain, we wouldn't create a harmonic that we can not recreate.
- From the spatial data, calculate the average value weighted by grid box for this dataset. This will be our baseline.
- From the original spectral data keep only the first harmonic $k = 0, l = 0$ and set the other harmonics to 0.
- Convert this truncated spectral to spatial.
- From this new spatial data, calculate the average value weighted by grid box for this dataset. This is our test.

Results of this numerical check confirmed this to be partially true. As can be seen in table 3.13, the values are close, but not completely identical. We discovered that these differences wouldn't be significant when the data wasn't normalized, but if it was, the differences between running the entire harmonic vs just the first one were far greater.

Table 3.14 shows the difference between running the whole harmonic vs just the first between normalized and non-normalized data. As previously mentioned, normalizing the data has many benefits, but using the first harmonic vs complete harmonic shows discrepancies between the results. The $Q = 20$ normalized results should not have changed signs. All of the results should have stayed negative and not changed signs.

While there are benefits to normalizing the data, it also has costs. We believed performing the "de-normalization" procedure in the spectral domain would help correct these issues. Normally, this processes is done in the spatial domain with the standard deviations gathered for each grid box during the principal component analysis procedure. However, by performing this procedure spectral domain instead we hoped these issues would be mitigated.

Table 3.14. First harmonic issues with normalized data

$Q = 0$ Not Normalized	$Q = 20$ Not Normalized	$Q = 0$ Normalized	$Q = 20$ Normalized
-0.7243337522	-0.7243751215	-0.1823753261	0.1723992728
-0.7047094343	-0.7047258066	-0.2019531272	0.096360793
-0.4127889293	-0.4128109635	-0.1005434379	0.1888661854
-0.84348063	-0.8435400164	-0.2556104894	-0.1815810165
-0.2582994231	-0.2583103473	-0.0755738703	-0.0485545522
-0.5697567499	-0.5698874176	-0.239977996	-0.0070130227
-0.4574287512	-0.4575739412	-0.1821243133	-0.0331458192
-0.8921690675	-0.8923426566	-0.1533232146	-0.3237588779
-1.0996391816	-1.0997346262	-0.1901059506	0.0805876248
-0.63784052	-0.6378372442	-0.3861952767	-0.1717153437
-1.2332411798	-1.2333866011	-0.4274376507	-0.1787212123

Using our inverse spherical harmonic transform equation from earlier (3.2) we can show how to multiply two spherical harmonics together in the spectral space. We start with the two spectral domain values we'd like to multiply together f_{k_1, l_1} and g_{k_2, l_2} to get h_{k_3, l_3} .

$$f(\theta, \phi) = \sum_{k_1=0}^{Q_1} \sum_{l_1=-k_1}^{k_1} f_{k_1, l_1} Y_{k_1}^{l_1}(\theta, \phi) \quad (3.31)$$

$$g(\theta, \phi) = \sum_{k_2=0}^{Q_2} \sum_{l_2=-k_2}^{k_2} g_{k_2, l_2} Y_{k_2}^{l_2}(\theta, \phi) \quad (3.32)$$

$$h(\theta, \phi) = \sum_{k_3=0}^{Q_3} \sum_{l_3=-k_3}^{k_3} h_{k_3, l_3} Y_{k_3}^{l_3}(\theta, \phi) \quad (3.33)$$

We then use Clebsch-Gordan expansion of the product of two spherical harmonics [1].

$$Y_{k_1}^{l_1}(\theta, \phi) Y_{k_2}^{l_2}(\theta, \phi) = \sum_K \sum_L \sqrt{\frac{(2k_1+1)(2k_2+1)}{4\pi}} \langle k_1, 0, k_2, 0, | K, 0 \rangle \langle k_1, l_1, k_2, l_2, | K, L \rangle \quad (3.34)$$

Where $\langle \rangle$ are the Clebsch-Gordan coefficients [35]. Applying this expansion to our harmonic data we get:

$$h_{k_3, l_3} = \sum_{k_1=0}^{Q_1} \sum_{l_1=-k_1}^{k_1} \sum_{k_2=0}^{Q_2} \sum_{l_2=-k_2}^{k_2} f_{k_1, l_1} g_{k_2, l_2} \sqrt{\frac{(2k_1+1)(2k_2+1)}{4\pi}} \langle k_1, 0, k_2, 0, | k_3, 0 \rangle \langle k_1, l_1, k_2, l_2, | k_3, l_3 \rangle \quad (3.35)$$

Instead of trying to solve the Clebsch-Gordon coefficients the alternative Wigner 3-j symbols can be used for the purpose of adding angular momenta [33]. These symbols relate via:

$$\langle j_1, m_1, j_2, m_2, | j_3, m_3 \rangle = (-1)^{-j_1+j_2-m_3} \sqrt{2j_3+1} \begin{pmatrix} j_1 & j_2 & j_3 \\ m_1 & m_2 & -m_3 \end{pmatrix} \quad (3.36)$$

Where $()$ is the a Wigner 3-j. Equation 3.35 is only solving for the specific harmonics $k = k_3, l = l_3$ of our new harmonic h . To solve the new complete harmonic we actually must solve for $\sum_{k=0}^{Q=k_1+k_2} \sum_{l=-k}^k$ giving the new harmonic a larger size and increasing the number of calculations. For a large Q this operation then becomes computationally expensive, but there are a few properties that Wigner 3j has that can be used to subtly remove a single “for loop” and skip some of the calculations. Even with these updates, the process is still computationally expensive and shouldn’t be used for large Q values. With more research we may find more ways to speed this process.

3.4.3 Validation

3.4.3.1 TIMESERIES NUMERICAL VALIDATIONS

As a test, numerical verifications were generated with faked data. A random $Q = 7$ harmonic is generated with $t = 50$ time points. We then convert the data to the spatial domain and sample it via Gaussian quadrature along the latitude axis ($Q + 1$ points) and equally spaced along the longitude ($Q * 2$ points). Note that these are enough points to exactly recreate the harmonic via spherical harmonics, so it is a lot of points. Next, eigenvalues and vectors are generated from the complete harmonic.

Using the sampled points generated earlier, we then recreate the dataset based off of the eigenvalues and vectors generated. This is done by both optimally weighing just the first harmonic, and then running the process on the full harmonic. The results of this verification are in figure 3.9. The results show two things: calculating the first harmonic only is good enough for getting the overall time series of the globe, and while the process isn’t perfect, it does a decent job of representing the original dataset. The SSE for between the original and rebuilt is 0.2097.

Digging deeper into the results we look at one of the harmonics in the time series and see how well the process was able to reconstruct the original data, both in the spatial and spectral space. Specifying $t = 1$ the original harmonic (3.15) and rebuilt harmonic (3.16) are fairly similar. Looking at the spatial original (3.17) vs the reconstruction (3.18) we can see that the pattern persists. This technique does a fairly good job of recreating the original dataset with a SSE of 3.8695.

Table 3.15. Original random harmonic at $t = 1$

k \ l	0	1	2	3	4	5	6	7
0	(0.83, 0.0)							
1	(0.44, 0.0)	(0.94, -0.06)						
2	(-0.33, 0.0)	(-0.09, 0.55)	(0.19, -0.25)					
3	(0.20, 0.0)	(0.15, 0.06)	(0.37, 0.49)	(0.089, -0.18)				
4	(0.47, 0.0)	(-0.00, 0.60)	(0.64, 0.12)	(0.42, -0.18)	(0.67, 0.66)			
5	(0.38, 0.0)	(0.58, 0.52)	(0.80, -0.56)	(0.59, -0.93)	(0.39, 0.36)	(0.48, 0.02)		
6	(-0.33, 0.0)	(0.11, 0.54)	(0.47, 0.95)	(0.32, 0.91)	(0.66, 0.56)	(0.86, 0.79)	(0.45, 0.35)	
7	(0.69, 0.0)	(0.50, 0.12)	(-0.02, 0.80)	(0.16, -0.56)	(-0.99, 0.03)	(0.11, 0.84)	(0.59, 0.12)	(0.27, 0.58)

Table 3.16. Rebuilt random harmonic at $t = 1$

k \ l	0	1	2	3	4	5	6	7
0	(0.68, 0.0)							
1	(0.40, 0.0)	(1.01, -0.01)						
2	(-0.17, 0.0)	(-0.07, 0.62)	(0.25, -0.14)					
3	(0.26, 0.0)	(0.25, 0.12)	(0.48, 0.43)	(0.08, -0.17)				
4	(0.32, 0.0)	(-0.00, 0.66)	(0.67, 0.15)	(0.48, -0.12)	(0.59, 0.68)			
5	(0.30, 0.0)	(0.63, 0.52)	(0.77, -0.60)	(0.69, -0.96)	(0.38, 0.37)	(0.48, 0.051)		
6	(-0.2, 0.0)	(0.11, 0.51)	(0.59, 1.059)	(0.30, 0.80)	(0.57, 0.54)	(0.80, 0.69)	(0.45, 0.35)	
7	(0.70, 0.0)	(0.50, 0.11)	(0.061, 0.79)	(0.11, -0.54)	(-0.96, 0.07)	(-0.021, 0.766)	(0.62, 0.14)	(0.29, 3.91E-16)

Table 3.17. Original random harmonic at $t = 1$ (spatial domain)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	-2.539	-2.258	-1.087	-0.386	-0.240	0.343	1.250	1.519	0.931	0.103	-0.401	-0.554	-0.847	-1.700
2	-3.170	-3.030	2.082	2.129	-1.527	-0.701	1.787	1.776	0.370	-0.163	0.180	1.125	2.484	0.956
3	-0.829	-0.967	1.237	-0.715	-2.045	-0.087	0.320	0.834	0.870	0.741	0.033	-2.160	-0.708	-0.014
4	1.133	2.416	-0.637	-1.484	1.636	0.864	0.956	1.607	2.092	1.246	0.807	-1.977	-0.190	-0.304
5	-2.275	-1.370	0.655	3.157	0.932	0.250	3.553	-2.169	0.244	0.877	0.147	-0.172	0.974	0.227
6	-0.248	-0.190	1.574	0.670	-3.842	0.382	4.024	-1.644	2.351	1.247	-1.385	0.929	0.126	-0.041
7	0.107	0.240	-0.025	-1.006	-0.766	2.745	3.333	0.055	-1.218	-3.391	-2.802	1.378	1.719	0.214
8	-0.185	0.384	1.489	2.881	4.177	4.635	3.516	1.155	-1.199	-2.424	-2.005	-0.743	-0.072	-0.168

Table 3.18. Rebuilt random harmonic at $t = 1$ (spatial domain)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	-2.694	-2.429	-1.193	-0.333	-0.025	0.592	1.409	1.572	0.924	0.103	-0.358	-0.511	-0.866	-1.797
2	-3.330	-3.254	1.772	1.990	-1.192	-0.352	1.852	1.811	0.235	-0.378	0.247	1.269	2.443	0.821
3	-0.872	-0.905	1.436	-0.657	-1.857	-0.016	0.163	1.005	1.011	0.626	0.133	-1.900	-0.676	-0.142
4	1.037	2.044	-0.679	-1.292	1.641	0.822	0.800	1.278	2.111	1.207	0.525	-1.939	-0.142	-0.480
5	-2.312	-1.608	0.102	2.739	0.949	0.345	3.557	-2.428	0.018	0.605	-0.073	-0.316	0.900	0.152
6	-0.439	0.196	1.693	0.345	-3.636	0.489	3.967	-1.421	2.248	0.964	-1.243	0.843	0.023	-0.130
7	-0.050	0.283	0.040	-1.029	-0.498	3.061	3.576	0.314	-1.345	-3.835	-3.119	1.196	1.773	0.266
8	-0.244	0.257	1.366	2.857	4.314	4.893	3.780	1.299	-1.262	-2.656	-2.258	-0.885	-0.084	-0.153

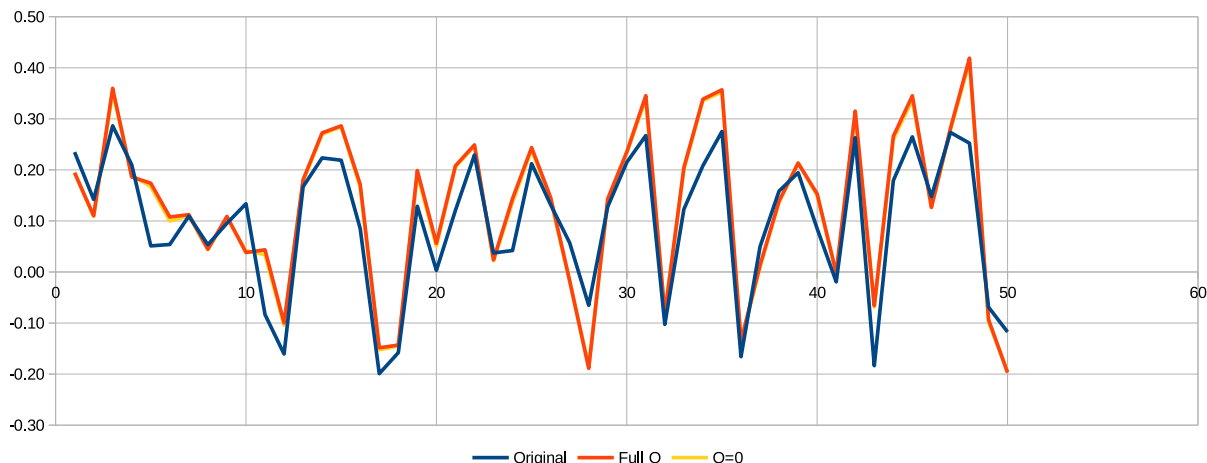


Figure 3.9. Numerical Validation - Random time series

In the real world, station locations aren't distributed out in a grid like this example, so another run was done using randomly picked station locations and see how well they could recreate the original dataset. For this analysis we will focus our efforts on the time series alone as we will “cheat” the calculations by only calculating the first harmonic, as it goes much faster.

Random location counts were picked at 112 (to match the original count done with the equally spaced analysis), 500 and 1000. The results for these runs can be seen in figure 3.10. The SSE of these three samples are 2.2639, 0.4153, and 0.3703. Note that none of these are as good as the original 0.2097 SSE we got for 112 sampled points along a grid box. This is most likely due to the randomly sampled points not giving good separation amongst themselves like how it's done via the grid box technique, and the hole punch technique we will be using.

3.4.3.2 TIMESERIES NCEP DATA VALIDATIONS

To confirm that the logic can run from beginning to end and produce reasonably good results, the GHCN data is first omitted from the analysis. Instead, a scheme to select points from the original `air.sfc.mon.mean.nc` NCEP 20th Century ReAnalysis v2c monthly surface air temperature dataset in order to simulate a station-like dataset. Those points were processed through the exact same logic as the GHCN stations would have been and then submitted through to the system.

Since we have the gridded dataset that these simulated station values came from completed, we can calculate the actual temperature from it; then we compare it to the temperature value from the simulated dataset that's processed through the complete code base all the way to the annual global temperature that it calculates. The final values between these two should be fairly similar.

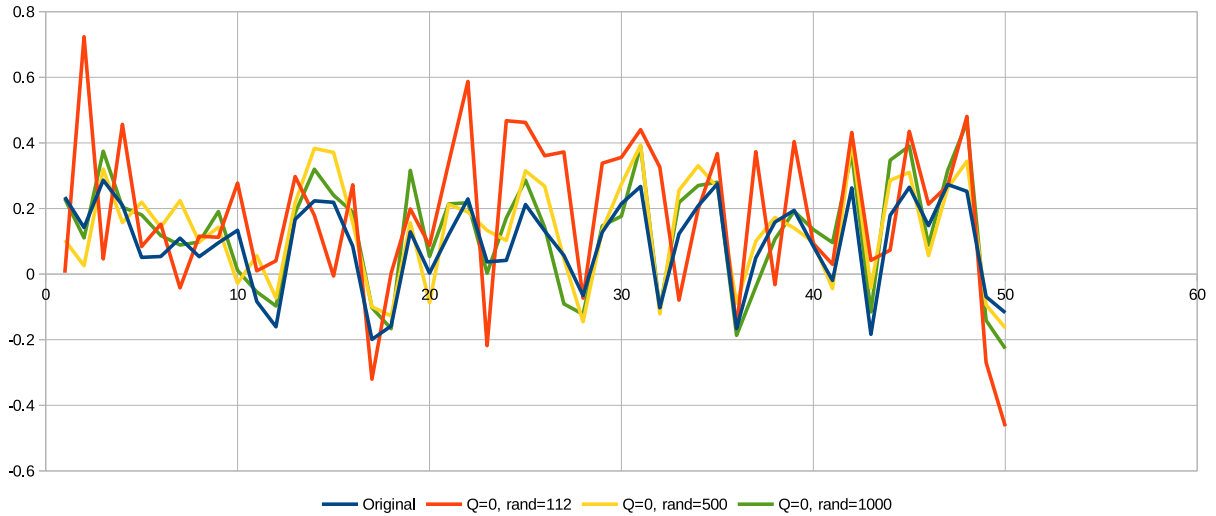


Figure 3.10. Numerical Validation - Random time series, random points

To “fake” station data we need to generate simulated values from this NCEP dataset, so we convert the spatial data to spectral to the optimal $Q = 102$ discovered previously (section 3.2.4.1). From this spectral dataset we can generate data anywhere on the globe. There is one important caveat: there is some error in the spectral to spatial conversion as the dataset doesn’t give us enough grid points to get the full harmonic; we know this from the Optimal Q issue discovered earlier (section 3.2.4.1). This implies that the simulated dataset generated will have some errors because the values for these station locations will effectively be interpolated, and since we don’t have enough grid points to generate a zero errors spherical harmonic, there will be errors in the rebuild. The numerical verifications in the previous section did not have this problem as the original dataset originated in the spectral domain, so the conversion process between spatial and spectral is effectively lossless.

The first “fake” station data generated is one based off of the 63 optimal station location coordinates based off of the Angell-Korshover network [3] and interpolated values are generated for each of these locations. Subsequent “fake” station locations are selected via a simple grid box technique at various resolutions.

Next, a PCA dataset was then generated from the same `air.sfc.mon.mean.nc` NCEP dataset at various harmonic levels ($Q = 0, 10, 20, 30, 40, 50, 60$) with the traditional 1961-1990 baseline selected for differencing. The “fake” station dataset was then submitted through the optimal weights and global temperature code using the various PCA datasets to compare their performance against one another. All runs were calculated at a truncated final harmonic of $Q = 0$ because the first harmonic contains the average spherical value. These requests were all submitted at a variance explained of 80% for the number of eigenvalues to select for the calculation.

While the final result only uses a $Q = 0$ truncation, the PCA generated eigenvalues and eigenvectors are not truncated to the other values mentioned earlier. The PCA $Q = 0$ truncation actually returns an eigenvector of length 1 (since we're only processing on a single harmonic) with a value of 1. This effectively solves the optimal weights as being uniformly weighted which can be viewed as baseline as if we didn't apply the optimal weighted logic to the system.

To compare these results, a baseline dataset was generated by mean differencing for the same baseline period for each grid box from the original NCEP dataset. Since this data was already in a spatial form, equation (3.37) could be used to calculate an overall global temperature value. This average was calculated for each date of the dataset and then each of the monthly values for a given year were averaged together to give an annual global temperature anomaly value.

$$C_t = \frac{1}{A} \sum_{i=1}^I A_i \hat{x}_i \quad (3.37)$$

This technique was tested with both normalized $((X - \mu)/\sigma)$ and regular data on both the PCA conversion and the “faked” datasets. Normalization tended to help the PCA $Q = 0$ truncated results, but caused issues with the non-truncated results we discussed.

The results from this validation can be found in figure 3.11. Notice the $Q = 0$ is the worst performer, and is an example of using this technique by equally weighing all of the different stations. As Q increases the overall error declines and the values became closer to the actual values. Considering we only sampled 63 stations from an original 18,048 (section 3.3.3) points in the dataset the results are pretty impressive. Depending on how the errors are viewed, $Q = 60$ isn't always the “best” representation of the original dataset. Table 3.19 contains the SSE by dividing the data in different ways.

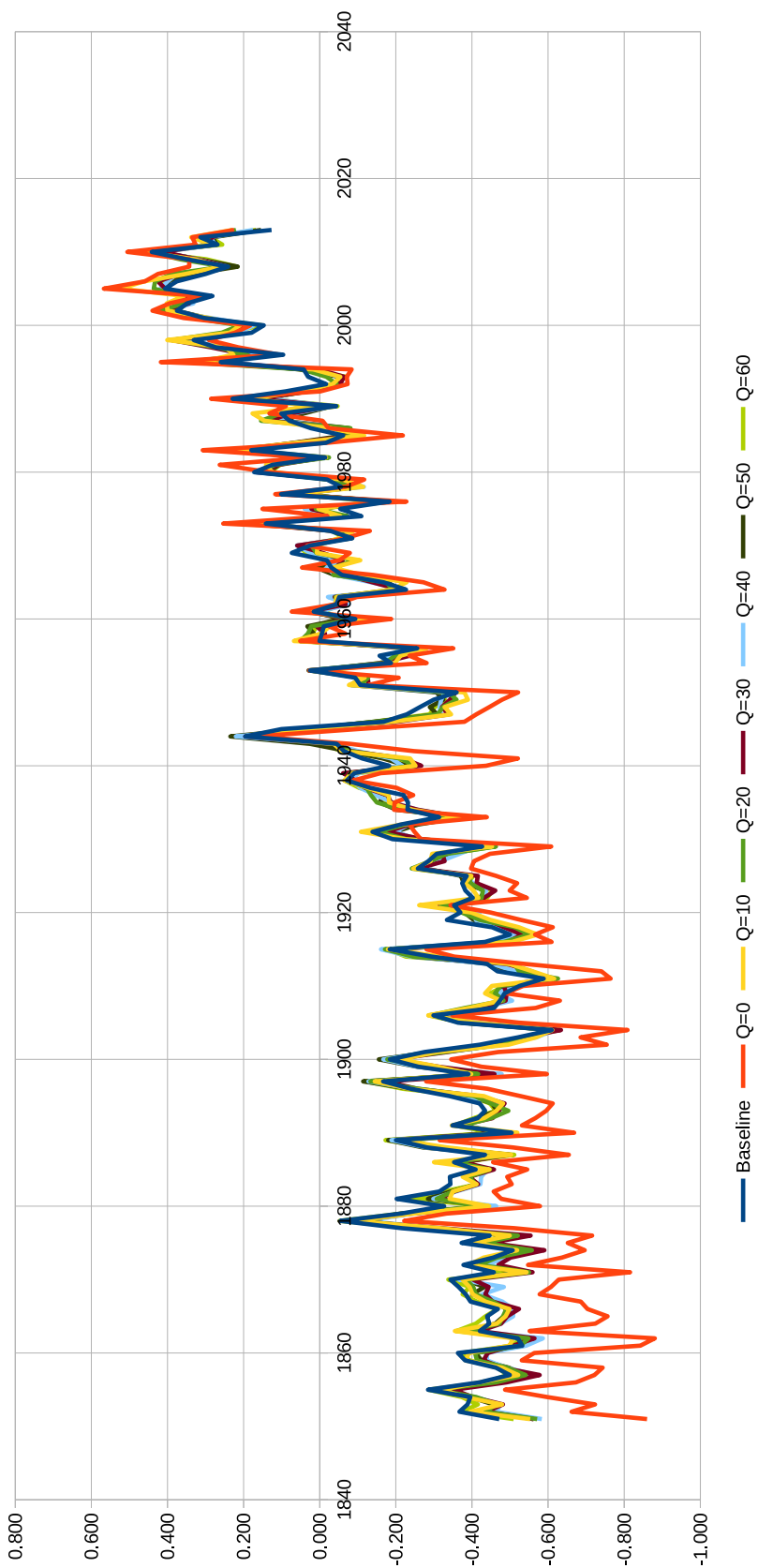


Figure 3.11. 63 stations various Q_s baseline 1960-1990 timeseries

Table 3.19. 63 station values SSE

Month	Q=0	Q=10	Q=20	Q=30	Q=40	Q=50	Q=60
Annual Averaged	4.25	0.39	0.39	0.51	0.47	0.33	0.38
January	15.73	2.81	1.89	1.96	2.75	1.90	2.32
February	12.27	3.79	1.50	3.40	2.21	1.86	1.70
March	11.27	2.19	2.25	1.71	2.55	2.22	3.30
April	10.75	1.96	1.69	1.67	1.58	1.60	1.26
May	12.19	3.36	1.38	1.39	1.67	2.18	1.31
June	7.57	3.53	1.34	1.18	1.62	1.40	1.41
July	3.09	1.80	2.13	1.84	1.35	1.60	1.45
August	4.60	2.14	1.37	2.41	2.01	2.27	2.00
September	3.72	2.17	1.90	1.71	1.78	1.78	1.50
October	4.42	2.30	1.40	2.18	1.74	1.68	1.54
November	7.78	1.75	2.10	2.60	2.00	2.03	2.32
December	13.53	2.23	2.30	3.58	3.75	3.02	3.49
Averaged Monthly	4.25	0.45	0.34	0.43	0.36	0.29	0.24
Overall Monthly Error	106.91	30.03	21.24	25.64	25.01	23.56	23.60

The first row “Annual Averaged” is produced by taking a PCA on the averages of all the months during a year, unlike “Averaged Monthly” which averages the final results of each month individually. Since $Q = 0$ is each station equally weighted, we end up with the same value for both rows. When $Q \neq 0$ the optimal weights logic is running because of the now differing eigenvalues and vectors. The overall best run when looking at the “Averaged Monthly,” which is what was plotted in figure 3.11, the higher the Q the better, but depending on the characteristic we want to solve for, this that is not always the case. This is because of the sampling size of 63 locations. If we instead sample 200 equally spaced stations (10 on the latitude and 20 on the longitude) we get the results in figure 3.12 and table 3.20.

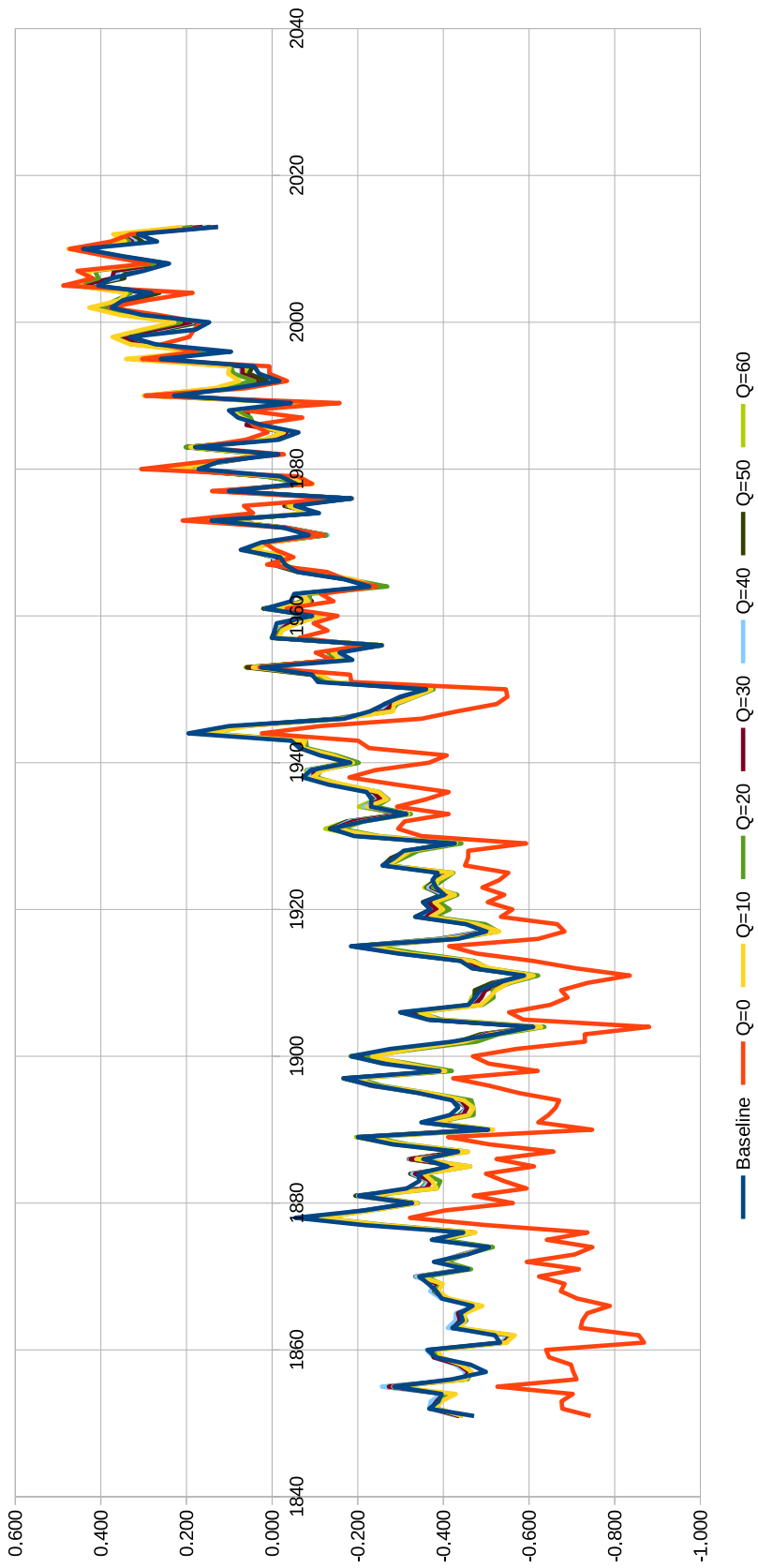


Figure 3.12. 200 stations various Q_s baseline 1960-1990 timeseries

Table 3.20. 200 station values SSE

Month	Q=0	Q=10	Q=20	Q=30	Q=40	Q=50	Q=60
Annual Averaged	5.63	0.29	0.29	0.14	0.12	0.10	0.09
January	7.61	0.92	0.72	0.62	0.56	0.53	0.46
February	5.11	1.31	0.82	0.65	0.71	0.68	0.69
March	7.32	0.93	0.63	0.50	0.46	0.48	0.43
April	10.95	0.50	0.33	0.33	0.31	0.32	0.34
May	15.32	1.04	0.67	0.41	0.43	0.35	0.29
June	10.91	0.79	0.75	0.49	0.41	0.41	0.38
July	6.94	0.70	0.72	0.50	0.41	0.35	0.37
August	7.34	0.76	0.98	0.50	0.47	0.29	0.33
September	6.09	0.61	0.67	0.30	0.23	0.20	0.20
October	2.79	0.34	0.46	0.29	0.31	0.35	0.25
November	4.20	0.59	0.44	0.40	0.37	0.37	0.38
December	7.71	0.63	0.62	0.60	0.41	0.45	0.39
Averaged Monthly	5.63	0.23	0.21	0.11	0.09	0.07	0.07
Overall Monthly Error	92.31	9.13	7.83	5.58	5.08	4.78	4.49

This pattern of improvement continues when more points are selected 3.21.

3.4.3.3 THE ISSUE WITH NORMALIZING THE DATA

We attempted to normalize the data since the previous work (section 3.3.3) showed that normalizing the PCA data yielded better overall eigenvalues and vectors. However, while the first $Q = 0$ truncation showed substantial improvement, higher truncations performed worse and worse. Normalizing the data was not a good idea because we had to undo the normalization by multiplying by the standard deviation for each grid box when we have the data back in spatial, before the final area weighted average is calculated.

Through experimentation, we noticed normalized data truncating the Q vs not truncated the Q yielded completely different results when calculating the overall global average, which contradicts earlier results (figure 3.9). This is where section (3.4.2.6) comes in, because we believed that the problem was the multiplication of the standard deviation happening in the spatial domain instead of the spectral one. Thus, the calculation was performed in the spectral domain instead to see if this would change the results. It did not, and the values were still miss matched. An example of this miss-match can be seen in table 3.22.

Normally, the results between these truncated and non truncated runs would be identical. “Spectral” vs non-spectral runs represents doing the standard deviation correction in the spectral space or not. This method is extremely inefficient and cannot go to very high Q truncation values.

Table 3.21. 450 station values SSE

Month	Q=0	Q=10	Q=20	Q=30	Q=40	Q=50	Q=60
Annual Averaged	7.04	0.10	0.08	0.09	0.09	0.09	0.05
January	15.54	0.53	0.36	0.25	0.17	0.18	0.15
February	12.76	0.60	0.45	0.16	0.23	0.23	0.16
March	11.97	0.49	0.26	0.15	0.15	0.16	0.17
April	12.13	0.28	0.25	0.23	0.15	0.12	0.15
May	15.89	0.20	0.23	0.23	0.12	0.10	0.11
June	10.58	0.22	0.19	0.14	0.14	0.14	0.14
July	9.04	0.19	0.24	0.13	0.24	0.21	0.16
August	8.43	0.30	0.17	0.13	0.13	0.11	0.12
September	5.21	0.36	0.21	0.11	0.16	0.14	0.14
October	4.00	0.12	0.18	0.16	0.14	0.14	0.15
November	4.46	0.27	0.21	0.15	0.14	0.16	0.15
December	10.13	0.28	0.46	0.61	0.47	0.27	0.30
Averaged Monthly	7.04	0.06	0.04	0.03	0.03	0.02	0.02
Overall Monthly Error	120.14	3.83	3.20	2.43	2.23	1.95	1.89

Table 3.22. SSE of $Q = 10$ Processed

	Truncated Q	Truncated Q Spectral Multiplication	Full Q	Full Q Spectral Multiplication
Annual	3.5237	3.5224	3.5939	3.5946
January	3.7507	3.7506	8.0724	8.0721
February	4.3626	4.3624	8.9313	8.9315
March	3.4689	3.4684	4.4153	4.4149
April	4.3064	4.3055	3.0202	3.02
May	6.5442	6.5425	4.988	4.9877
June	5.3371	5.336	3.2039	3.2037
July	6.6376	6.6369	3.7752	3.7748
August	6.9538	6.9524	6.1017	6.1014
September	3.5319	3.5308	1.9285	1.9286
October	2.5073	2.5068	2.3882	2.3883
November	4.7483	4.7478	8.6481	8.6481
December	6.0302	6.0296	8.0554	8.0548
Averaged Annual	1.4504	1.45	0.5719	0.6627

Table 3.23. Getting the same value for different Q truncations when forcing the same harmonic of $Q = 0$ on the standard deviation

Annual	$Q = 0, Q = 0$ SD	$Q = 10, Q=0$ SD
1851	-0.3830487561	-0.3829599422
1852	-0.3051958488	-0.3051544609
1853	-0.3394220011	-0.3393520829
1854	-0.4082707274	-0.4081948857
1855	-0.2682972515	-0.2682033981
1856	-0.3345769226	-0.3345084669
1857	-0.3381586922	-0.3381924447
1858	-0.381705537	-0.3817218759
1859	-0.3995502407	-0.3995428882
1860	-0.3954855655	-0.3955094746

The next comparison is the “Full” vs “Truncated” Q . The difference here is simply calculating the complete harmonic vs just the first harmonic.

The reason for this inconsistency was not the different Q values, but the changing harmonic representation of the standard deviation. When calculated at $Q = 0$, only the first harmonic of the standard deviation was used. When calculated at higher Q values, higher order portions of the standard deviation were used. To numerically verify this was the case, we took a $Q = 10$ harmonic and specifically only added a $Q = 0$ standard deviation and compared it to taking a $Q = 0$ harmonic for both. These results can be seen in table 3.23 and are the reason for the discrepancies between calculating the values at different Q truncations.

Performing the normalization technique does help if we are weighing the points equally instead of using the optimal logic, but if perform the actual complete optimal weighted scheme, they cause more trouble than good.

3.4.4 Data and Processing

The station data was originally going to come from “Global Historical Climatology Network” monthly V2 (GHCN) [38]. This dataset contains monthly mean/min/max temperatures from a collection of stations all over the world. The copy used has monthly readings from 1701-01-01 to 2010-06-01 with a total of 7280 temperature stations. After our research had began, a newer version of this dataset became available [28], and we switched to the newer dataset midway. The newer version had the same number of stations, but the data had been cleaned up from the previous version. The final date of the dataset increased to 2016-11-01 which allowed us to show more recent temperature changes.

The new version eliminated many issues that in the older one. The duplicate inconsistent readings for the same station had been corrected and an entirely new dataset was created that attempted to remove erroneous values. That extra “c” was a big improvement.

Unfortunately, this dataset still had some of the same baggage; the number of available long term stations in the 21st century quickly takes a nose dive as can be seen in Figure (3.13).

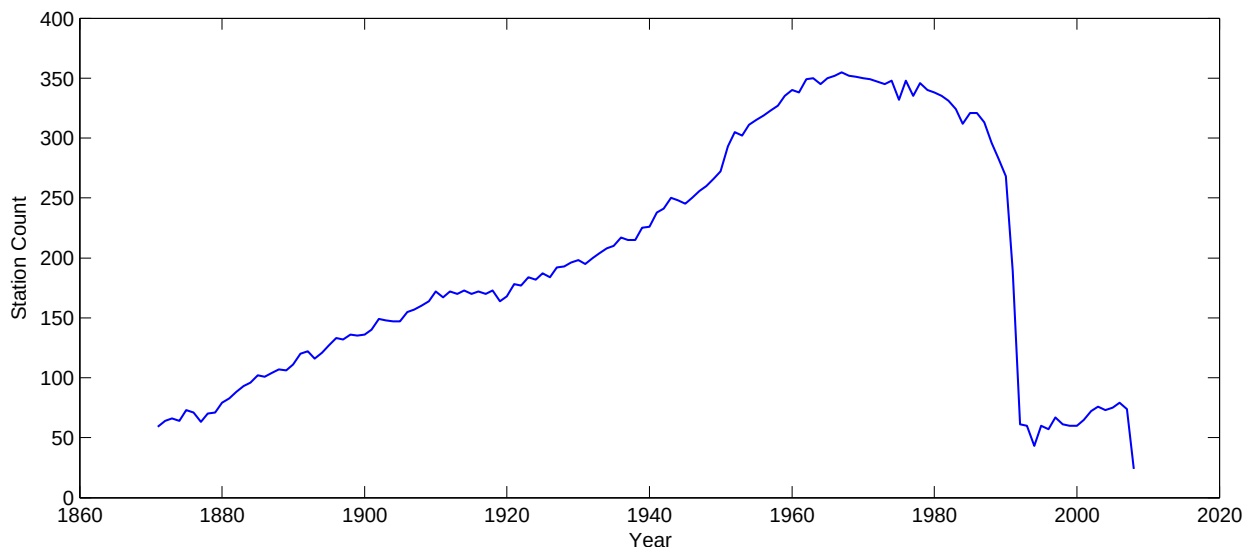


Figure 3.13. Station Counts

Coverage was still an issue with there being a better representation of stations in the northern hemisphere than the southern. This problem increases the further back in time that the dataset is inspected. There could be issues with calculations towards both the beginning and ending years of this dataset. Coverage over the oceans is also problematic.

Figures (3.14)-(3.17) shows where all the long term stations are located at various points in time assuming a 500 km “hole punch.”

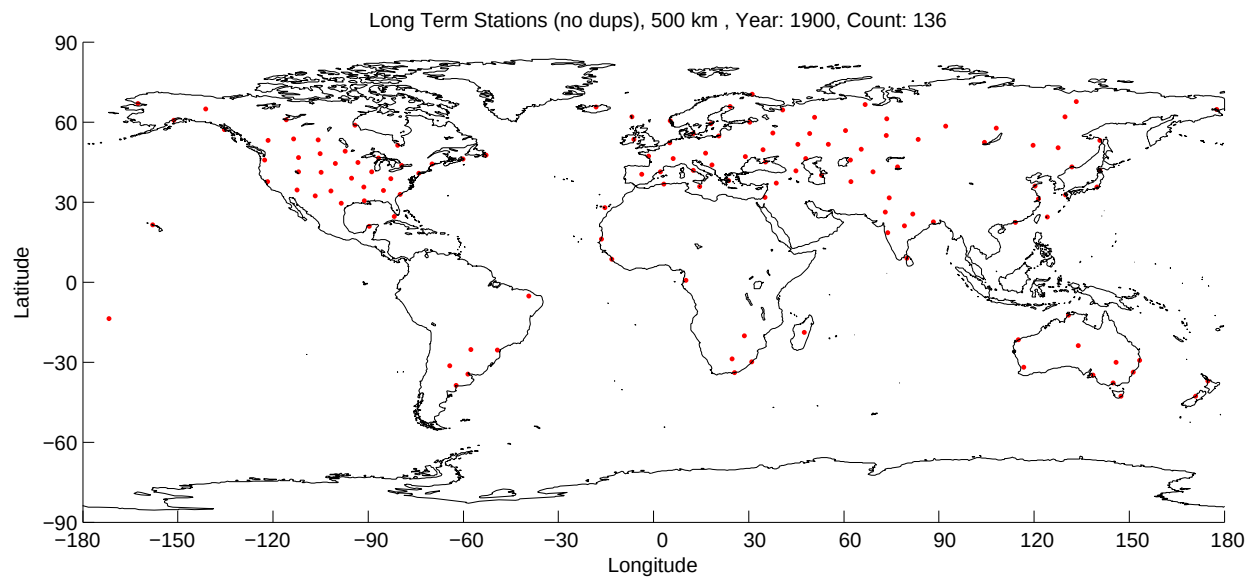


Figure 3.14. Long term station locations

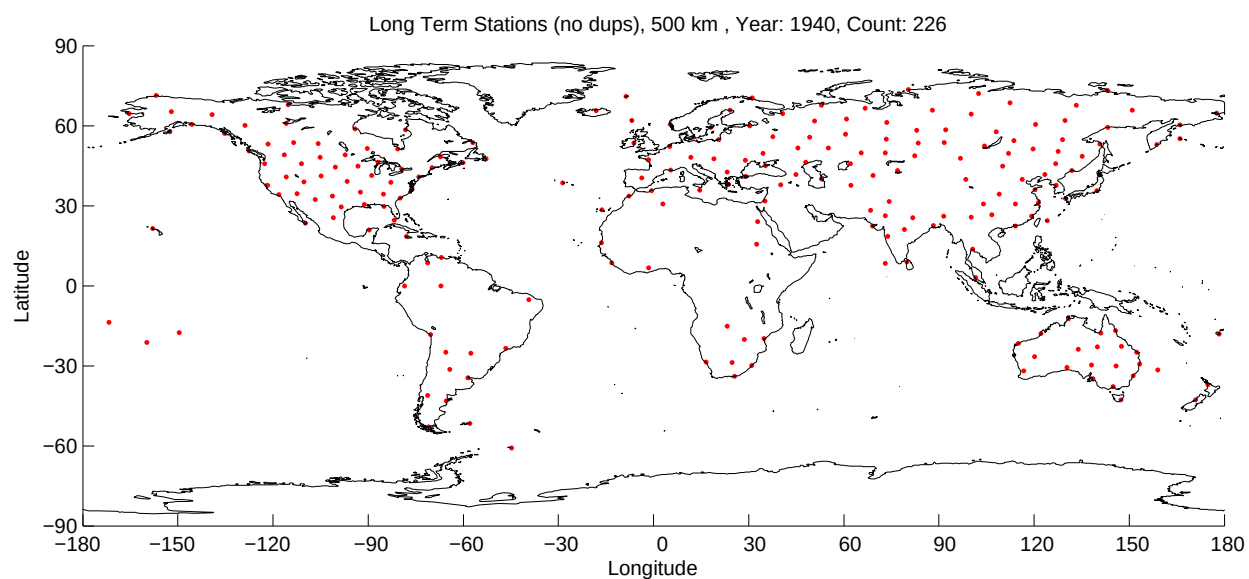


Figure 3.15. Long term station locations

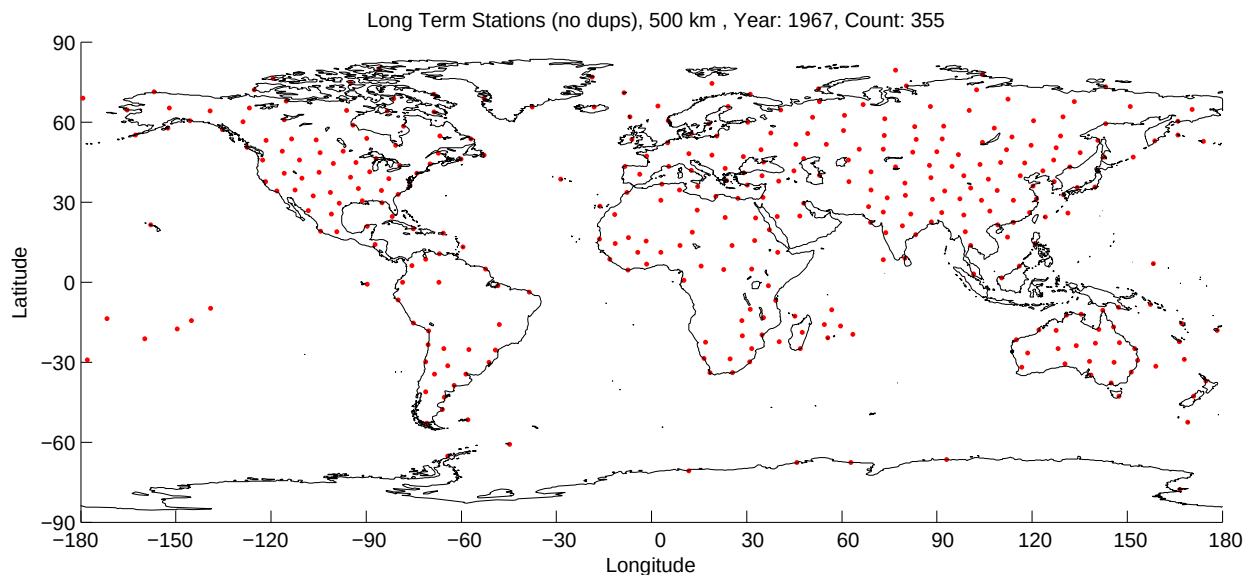


Figure 3.16. Long term station locations

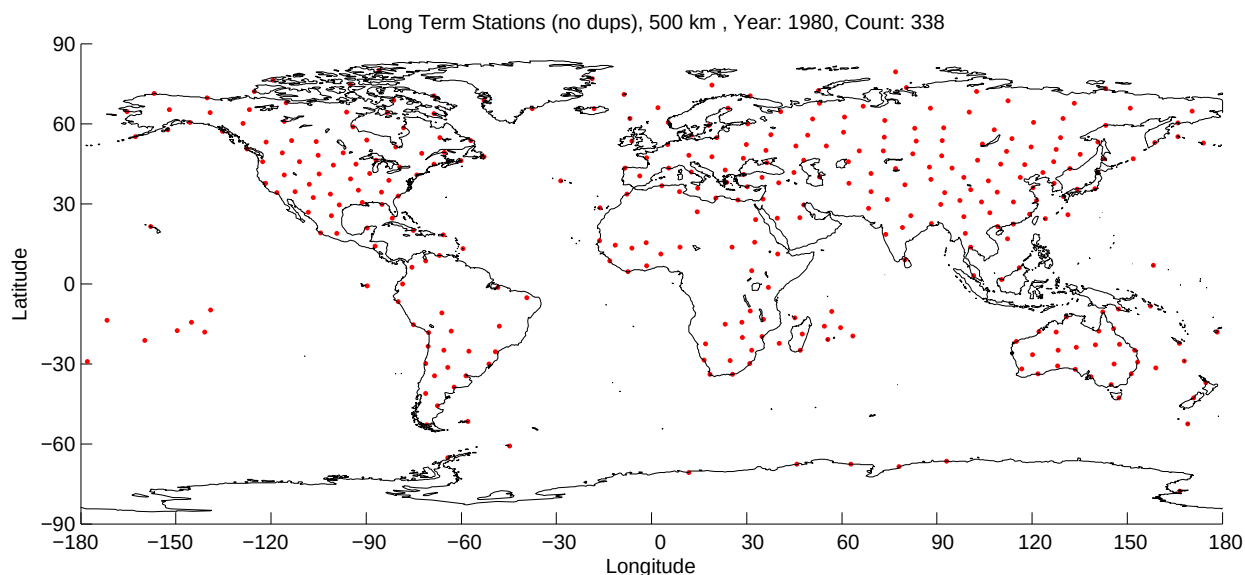


Figure 3.17. Long term station locations

Using the same 500 km selection criteria, the station count peaks at 355 (3.16) stations in the world and they are distributed fairly well. There may be a few gaps in some places of the world (Africa, South America, etc...) but overall this technique does a good job of ensuring that only the “best” stations are selected. It also makes sure that stations are spread out evenly across the globe.

3.5 RESULTS AND CONCLUSIONS

With all the pieces of the puzzle ready, we now use our “hole punch” method to select stations from GHCN and find the optimal weighting scheme for them to generate an annual temperature.

The hole punch method was first applied with a 500 km radius and required a complete set of all 30 years of data during the baseline period of 1961-1990. Results at various Q truncations can be seen in figure 3.18. There is no “truth” marker, so instead we’re using the NCEP data as a baseline. This is just used for comparison and should not be treated as the actual truth.

You’ll notice that in the end we didn’t actually calculate up to the optimal $Q = 102$ because high Q truncations would only work in hand with higher station counts. Our “hole punch” method peaked at around 350 stations, so higher Q ’s wouldn’t help. Also, calculating the full harmonic would have taken too long, even with our Spark cluster optimizations.

There’s clearly not enough stations available for analysis in the beginning of the dataset, as seen in figure 3.13. This explains the erratic behavior of the time series during the beginning of the period.

However, if we instead compare our results with GISS Surface Temperature Analysis [18] [49], we find that our technique shows results very close to theirs 3.19. Though, our results always appear to be a little lower than the GISS results because their analysis was done with an earlier 1951-1980 baseline period compared to our 1961-1990 baseline period.

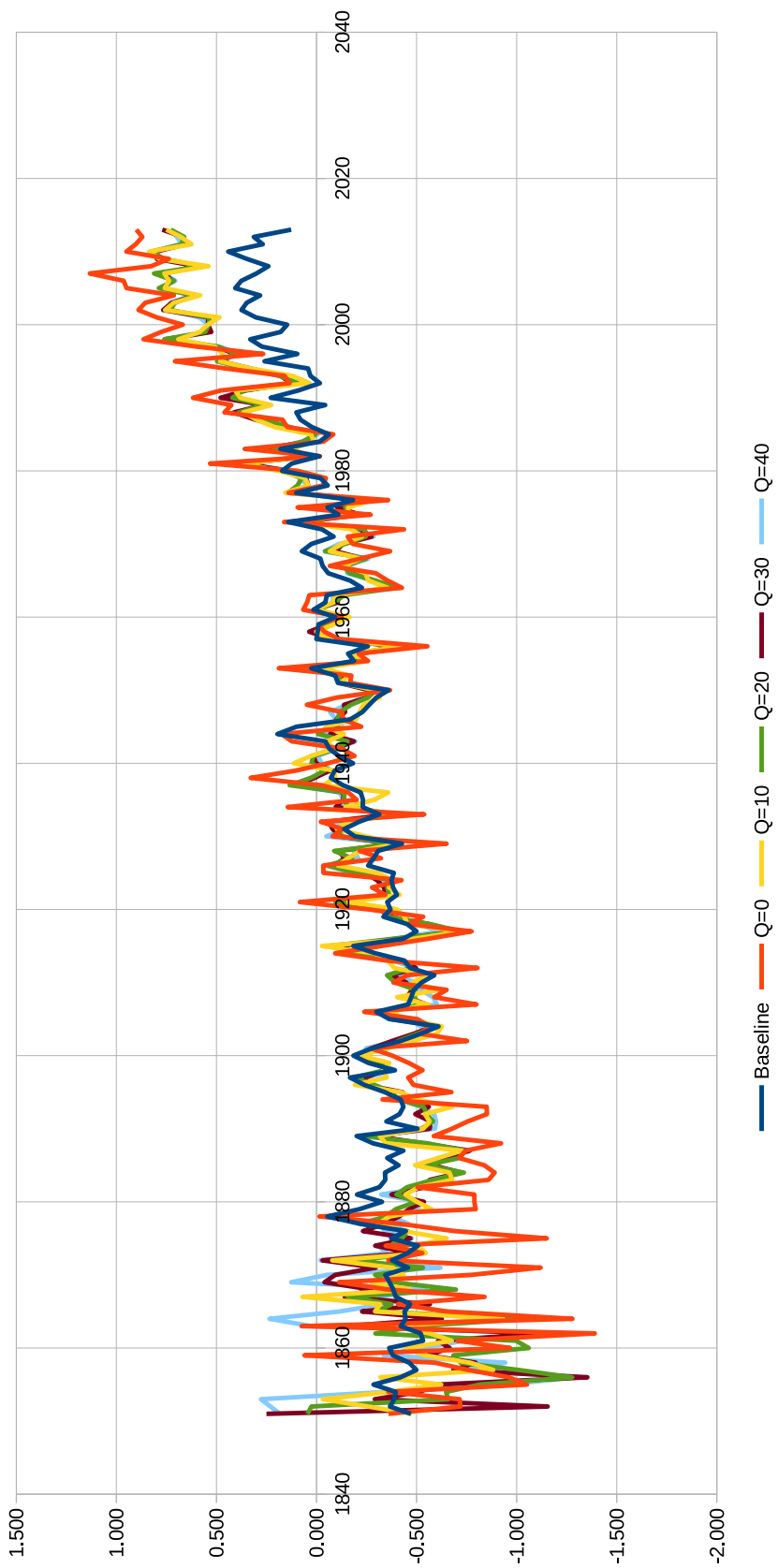


Figure 3.18. Optimal Weighting compared to NCEP

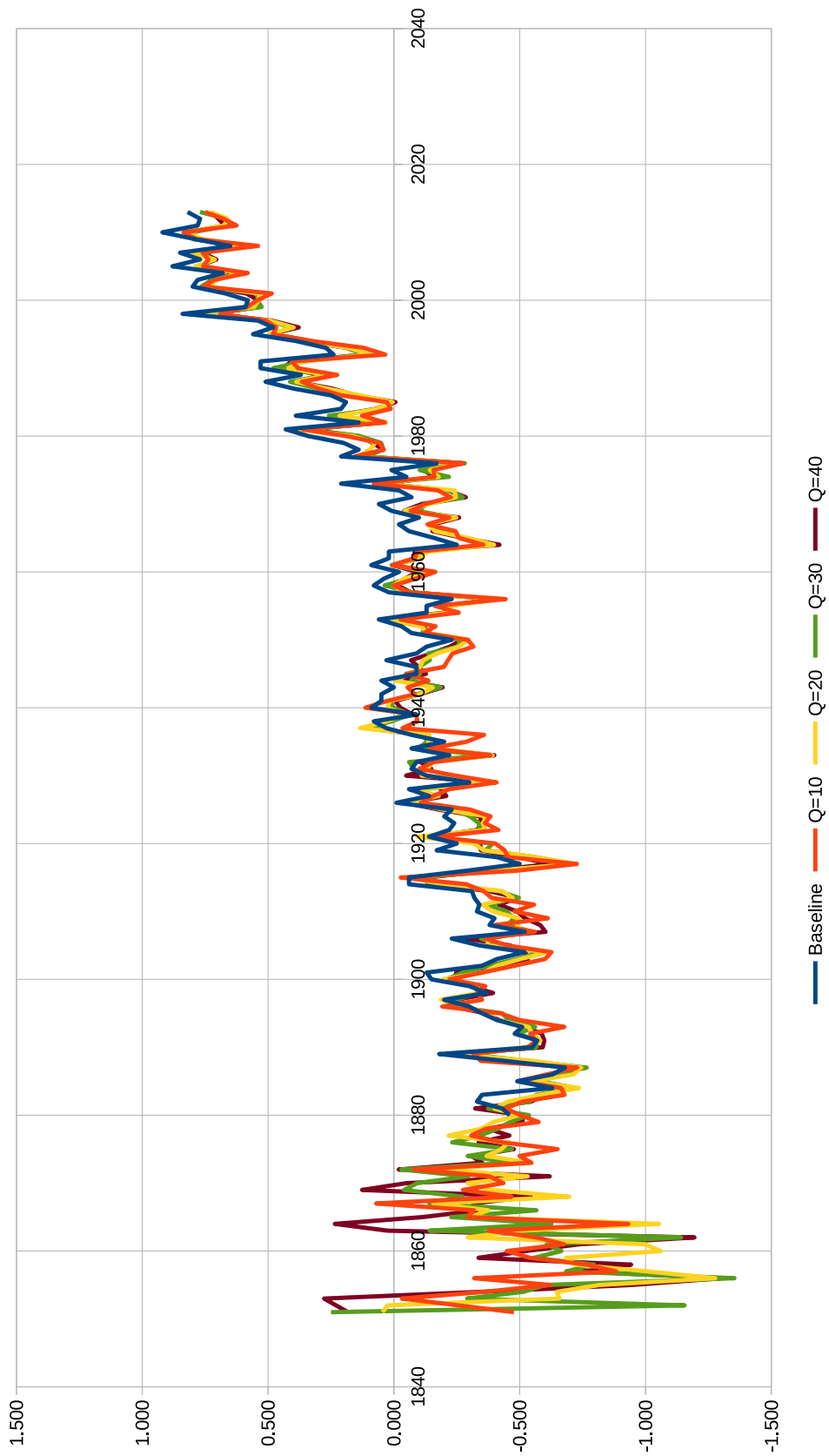


Figure 3.19. Optimal Weighting compared to GISS Surface Temperature Analysis

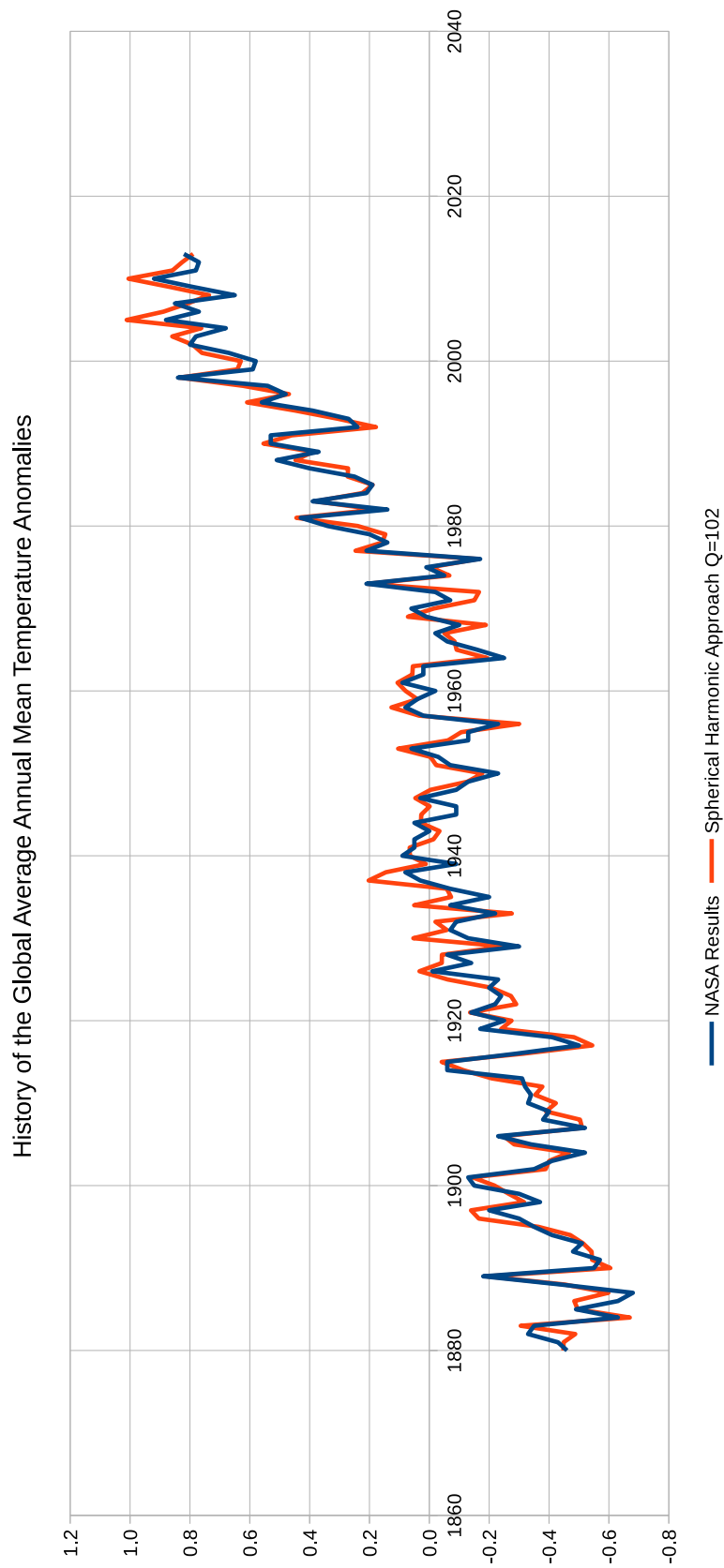


Figure 3.20. Optimal Weighting compared to GISS Surface Temperature Analysis - Corrected baseline

We then re-ran the whole process (on a truncation of $Q = 102$) to match then GISS baseline dates of 1951-1980. This re-run required re-processing everything since the baseline changes the intermediate results. Results from this re-run are shown in figure 3.20.

Note the GISS Surface Temperature Analysis datasets come in two different flavors. The first flavor tries to calculate the temperature over both land and ocean regions, and the second one only over the land regions. We compare our dataset to the land based version as we have very little spatial representation over the oceans as illustrated in many of the figures in section 3.4.4. Even though our method is designed to calculate the values globally, since we lack station representation in many of these regions the technique has no stations to weight.

Another benefit from using this procedure is that we can calculate not only the first harmonic (which gives us the overall temperature average), but the harmonic for the entire globe. We can then take the complete harmonic, convert it to spatial and theoretically give the temperature value for any location on the sphere. However, doing this will significantly increases the number matrix inverses that need to be calculated, and this is where our choice of Java and refactoring our code to work on Apache Spark comes in. We can now distribute the workload out to a cluster to calculate, which should increase our speed roughly linearly: double the compute power double the speed.

Calculating all the dates and $Q = 102$ harmonics would still take too long with our resources, but a few were calculated for analysis and some interesting results were gathered. Results of a complete reconstruction of $Q = 102$ for 1980 is shown in figure 3.21 and 3.22.

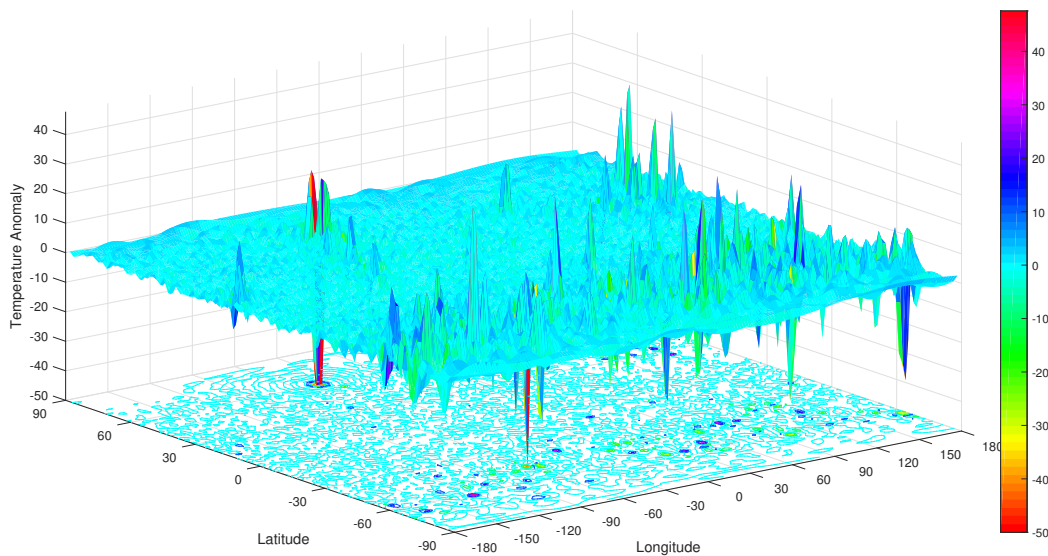


Figure 3.21. Spatial Reconstruction 1980 $Q = 102$ side view

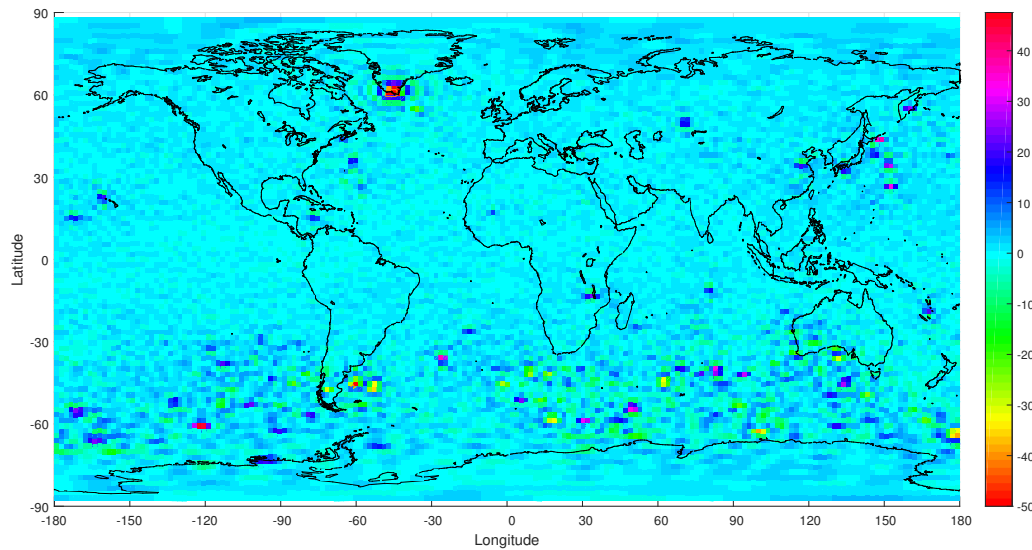


Figure 3.22. Spatial Reconstruction 1980 $Q = 102$ top view

Notice the poor reconstruction of some of the regions here. In particular, oceans in the southern hemisphere's have rather poor reconstructions. The reason for this is we have no GHCN station representations in these areas. We expect the majority of these results would be at or around zero and shouldn't get large values in either the negative or positive side; that is not the case.

Lowering the Q truncation to $Q = 30$ decreases the extreme values and illustrates the issue better (figures 3.23 and 3.24). These new figures show the extreme variations better as the colors span larger areas. Interestingly, it also shows a large irregularity at the tip of Greenland where we have no station.

While this technique works well for generating an overall climate trend, without better spatial station representation there are issues in reconstructing the temperature for certain regions. There are processes we could use to aid in mitigating these issues:

- We could relax our requirement for stations having a complete representation during the 30 year baseline period.
- Figure out an optimal baseline period based on spatial representations. There are stations in GHCN that are actually buoys in the middle of the ocean, some of these could possibly be used.
- While $Q = 102$ was the optimal truncation to reconstruct a dataset that had been converted to spectral, it may not be the optimal truncation to recreate a dataset based off of station locations.

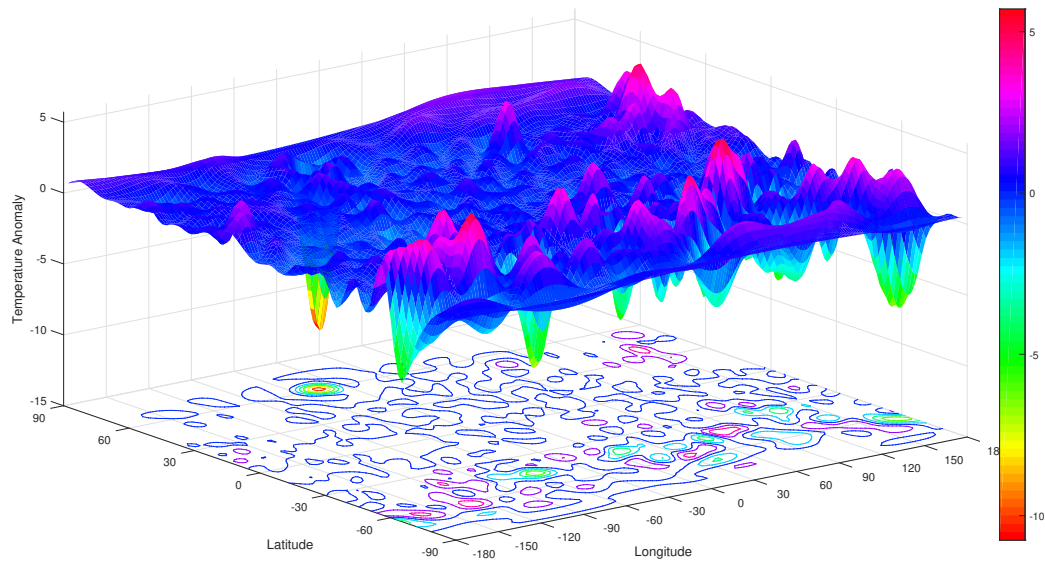


Figure 3.23. Spatial Reconstruction 1980 $Q = 30$ side view

- More experimentation could be done deriving faked data from the original NetCdf gridded dataset to try and recreate it. Working with the data this way we would have a “truth” marker to try to optimize on.

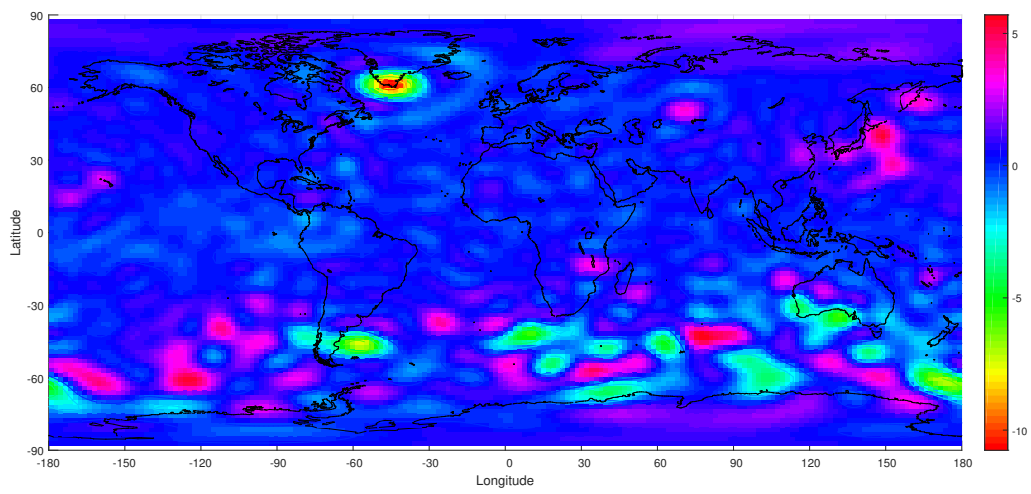


Figure 3.24. Spatial Reconstruction 1980 $Q = 30$ top view

CHAPTER 4

CONCLUSION AND FUTURE WORK

All of the research presented here at first may seem disconnected but with closer examination it is completely interrelated. The goal that this research is hoping to share is that cloud computing has gotten to the point where every aspect of research in general can be performed in the cloud.

Basically, cloud computing makes it possible to run complicated data processes on both large and computationally expensive datasets with methods that would normally have been reserved for super computers. Researchers with limited means and meager resources can still run complicated ideas with smart planning and they can do all of their research with cloud based resources.

On one side, we showcase the distribution and delivery of climate datasets via the 4DVD web application that exists in the cloud, and on the other we show that some complicated numerical processing can take place on a Spark cluster which also can exist in the cloud.

The work in relation to 4DVD has led to the open sourcing of our project (<https://github.com/dafrenchyman/climate> and <https://github.com/dafrenchyman/climate2>) and interest has come from many different sources. While there are climate uses for this software, other uses have also come up as this research has led to a collaboration between SDSU and Harvard Medical School in relation to brain imaging, a use case that we never even considered.

While 4DVD is fully functional, there is always room for improvement. There are already plans in place to continue this work. There are many directions we can go. One of the more interesting possibilities is the Webassembly (WASM) route. Webassembly is a new web based virtual machine that can be compiled from C/C++ code. The resulting code can run extremely fast and is faster than JavaScript. Many of the mathematical calculations being done could instead be done more efficiently with compiled code via WASM. This would speed up a lot of the processing that takes place behind the scenes on the front end. There are noticeable delays when certain processes are being run that could be improved. These changes would also allow for even larger spatial datasets to be run. Other benefits could include reading in binary data instead of JSONs which would result in faster downloads. This could also lead future work into re-thinking how to organize the data in the database. The entirety of a particular date and layer could be stored in a binary “blob” object instead of individual entries which would result in even faster lookups from the database, as only one row would be returned instead of one for each grid box.

The optimal interpolation portion of our work proves that computationally heavy work can be run through Apache Spark and thus can be processed on numerous cloud computing service providers such as AWS. A new thread-safe Java-based spherical-harmonics-transform code has been implemented that can calculate high Q truncations has been open sourced and made available on-line (<https://github.com/dafrenchyman/mrsharky>). This code base also includes an easy to refactor Apache Spark JavaRDD process that could be modified to work for other computationally expensive problems. This makes it easy for others to get their code to work in a distributed computing fashion. Java native interface was also exploited as part of the research here and with a little effort, others could take their native computationally expensive C/C++ code and expose it to the Apache Spark ecosystem to get their projects up and running on a cluster. While this is not something that was specifically done in this research, there are examples of both JNI and Apache Spark in the code base so the two processes could easily be combined together.

In this research we've shared a new method to calculate the global temperature average via this optimal interpolation technique that gives results similar to that of the NASA Goddard Institute of Space Studies (GISS) Surface Temperature Analysis dataset. Results that are based off of James Hansen's famous research.

We've also analyzed some of the complete spatial reconstructions that this new technique allows us to do. The work relating to these complete spherical reconstructions is incomplete. More could be done to find the correct Q truncation to use based off of the number and location of available stations for a given date. This optimal Q will also be dependent on the resolution of the PCA dataset. Finding this optimal Q could be aided by calculating the errors for the weights. The formulas for them are available in the original referenced paper and could be added to this process.

In the end, we showed two very different ways in which cloud computing can be put to use for researchers needs and we hope others will exploit them in the future.

APPENDIX
NOTES TO THE ASPIRING JAVA NUMERICS
PROGRAMMER

NOTES TO THE ASPIRING JAVA NUMERICS PROGRAMMER

I thought it might be useful to spend a little time in this very last section to discuss helpful practices that I learned along the way writing numerics in Java.

First, make sure your Java project is a Maven one. Maven makes package management incredibly easy. You want to use someone else’s library in your research, it’s as simple as putting in something like the following in your “pom.xml” file.

```
<dependency>
    <groupId>org.jblas</groupId>
    <artifactId>jblas</artifactId>
    <version>1.2.4</version>
</dependency>
```

I would also spend some time to learn how JNI works. If you refer to section 3.2.2.6, I discuss a good resource for learning JNI. While Java makes programming and development easier than in traditional C/C++, there aren’t nearly as many numerical libraries available and it will never be as fast as native compiled code, but it sure is much easier to read and follow! I learned JNI as part of the research done here, but have since applied it in my career outside of academia a few times. It is a valuable skill to have.

A.0.1 Helpful Libraries

A.0.1.1 JAVATUPLES

Most programming languages aren’t great at having functions that return multiple values. In C/C++ pointers are passed when calling the function that can be populated, in R a list can be returned that has all of the elements, and in Java the norm is usually to return a specifically written class that gives you access to all the variables either via getters or directly. While going this route is the correct solution when this class has certain built-in functions for different types of processing, if the point of the class is just to pass a bunch of different objects back, then it’s not the correct way to go.

To fix this, there’s the very useful javatuples library:

```
<dependency>
    <groupId>org.javatuples</groupId>
    <artifactId>javatuples</artifactId>
    <version>1.2</version>
```

```
</dependency>
```

This library makes dealing with simple multiple-value returns extremely easy. It's especially useful when paired with multi-threaded operations because you usually need to return a few variables and don't want to have to implement a simple class just to do it.

Another great use for this library is when you need to use a dictionary or map class in Java with a multi-indexed key. Before discovering this library, I would often map multiple maps within one another. This made it more complicated to check for the existence of keys as you would need to check for them at each nesting. While it is slower to use a "tuple" as a key, there's much less boiler plate code involved, and you can skip the creating of a map within a map step, which is frustrating. My only regret is not finding this library sooner, because there are still places in my code where I have nested Maps.

A.0.1.2 APACHE COMMONS CLI

When code is called from the command line, arguments need to be passed to the code so it knows what to do. To save yourself the headache of always reimplementing your own input parser, it's much easier to use Apache Commons CLI.

```
<dependency>
    <groupId>commons-cli</groupId>
    <artifactId>commons-cli</artifactId>
    <version>1.4</version>
</dependency>
```

I even wrote an easy to use abstract class that can be used as a base for other implementations. The abstract class is located here:

`com.mrsharky.helpers.InputParser_Abstract` with child classes based off of this located throughout the work.

While everyone wants to write fancy GUIs for things, most of the real work happens on the command line via bash scripts that can be automated by processes such as cron jobs. It is always a good idea be able to parse input arguments in your main classes, this library just makes the work trivial and easy.

A.0.2 Helpful Patterns

A.0.2.1 SYSTEM PROPERTIES

On a few occasions, there may be a desire to run some code with a certain condition in place that you don't want exposed. It might be for a test, it might be an option you don't want code up as in input argument. A helpful way around this is to set a system property via

`System.setProperty(propertyName, propertyValue)`. Later on, you can grab the value out of this property with a simple `System.getProperty(propertyName)`. If the property was not set, the method will return a null value.

This trick was used several times to generate results under “odd” conditions. This ensured that the code would still operate normally if the properties were not set. The reason to use this is that if you happen to forget to undo these “odd” conditions and you didn’t use this trick, then the normal operation of the code would not work correctly.

A.0.2.2 MULTI-THREADING

Multi-threading in Java used to be extremely difficult, and searching on-line for tutorials on the process can net you solutions based off of old usually complicated methods. The following is a simple pattern that I often use for executing code on multiple threads. You simply state what you want to run, wait for it to finish, and collect the results; you don’t have to write separate classes for the processing.

```
// Programmically get the number of threads available
int threads = Runtime.getRuntime().availableProcessors();

// Start a multi-threaded service
ExecutorService service = Executors.newFixedThreadPool(threads);

// Specify what these threads will return
List<Future<Triplet<Date, Double, double[]>>> futures = new ArrayList<
    Future<Triplet<Date, Double, double[]>>>();

for (int i = 0; i < 100; i++) {

    // Set whatever variables you want to use in your process as
    final
    final int iFinal = i;

    // Process the spherical harmonics multi-threaded
    Callable<Triplet<Date, Double, double[]>> callable = new Callable<
        Triplet<Date, Double, double[]>>() {
        public Triplet<Date, Double, double[]> call() throws Exception
        {

            // Call your process
            SomeClass values = new SomeClass(iFinal);
```

```

        Triplet<Date, Double, double[]> results = Triplet.with(
            values.getDate(), values.getDouble(), values.
                getDoubleArray());
        return results;
    }
};
futures.add(service.submit(callable));
}

// Wait for all the threads to complete
service.shutdown();
service.awaitTermination(Long.MAX_VALUE, TimeUnit.NANOSECONDS);

// Loop through the results and grab them
for (Future<Triplet<Date, Double, double[]>> future : futures) {
    Date date = future.get().getValue0();
    double value = future.get().getValue1();
    double[] valueArray = future.get().getValue2();

    // Do whatever you want with the gathered variables
    ...
}

```

It's also possible to completely skip the “for” loop at the end by creating a class where results are “synchronized” when written out to it. This is another pattern I recommend to learn. This “synchronized” trick is used in the code base discussed in this paper to keep the memory footprint small for the inverse discrete spherical transform. I did it for the multi-threaded version of the code so I wouldn't store this ever growing return type only to add them all together at the end. With this trick you can simply have the multiple threads add the values as it moves along, “synchronized” makes it so only one thread can run this process at a time and they can't accidentally hit it at the same time and fail to return the correct results.

In my research outside of this project I've also used this as a way to multi-thread work and skip steps if they wouldn't converge to a number better than what another thread has already done. For example it can be used when running a cross-validation to test different variable combinations in a model. By the time you've reached the 3rd of 5 cross validations for a particular variable combination, if you already know the results from this run will never be as good as that a prior run, you don't want to waste time and continue processing the 4th and 5th

cross validations. You can simply check what the value as in this “synchronized” variable and move on to the next variable combination. If however, you come up with a new “best” value, you can append the value and other threads will then know what the new number is to beat, potentially allowing you to skip more threads.

BIBLIOGRAPHY

- [1] "the wolfram functions site - sphericalharmonic. <http://functions.wolfram.com/Polynomials/SphericalHarmonicY/16/ShowAll.html>.
- [2] E. ANDERSON, Z. BAI, J. DONGARRA, A. GREENBAUM, A. MCKENNEY, J. DU CROZ, S. HAMMARLING, J. DEMMEL, C. BISCHOF, AND D. SORENSEN, *Lapack: A portable linear algebra library for high-performance computers*, in Proceedings of the 1990 ACM/IEEE conference on Supercomputing, IEEE Computer Society Press, 1990, pp. 2–11.
- [3] J. ANGELL AND J. KORSHOVER, *Global temperature variations in the troposphere and stratosphere, 1958–1982*, Monthly Weather Review, 111 (1983), pp. 901–921.
- [4] G. ARFKEN, *Spherical harmonics*, Mathematical methods for physicists, 6th edn, section, 12 (2005), pp. 786–802.
- [5] M. ARMBRUST, R. S. XIN, C. LIAN, Y. HUAI, D. LIU, J. K. BRADLEY, X. MENG, T. KAFTAN, M. J. FRANKLIN, A. GHODSI, ET AL., *Spark sql: Relational data processing in spark*, in Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, ACM, 2015, pp. 1383–1394.
- [6] J. F. BLINN, *Simulation of wrinkled surfaces*, SIGGRAPH Comput. Graph., 12 (1978), pp. 286–292.
- [7] M. L. BRAUN, J. SCHABACK, M. L. JUGEL, AND N. OURY, *jblas-linear algebra for java*, 2015.
- [8] D. R. BRILLINGER, *Time series: data analysis and theory*, vol. 36, Siam, 2001.
- [9] R. CHAMBERLAIN AND J. SCHOMMER, *Using docker to support reproducible research*, DOI: <http://dx.doi.org/10.6084/m9.figshare.1101910> (2014).
- [10] CITRIX, *Citrix XenServer 7.0 Quick Start Guide*, 851 West Cypress Creek Road Fort Lauderdale, FL 33309 United States of America, 1.0 ed., May 2016.
- [11] G. COMPO, J. WHITAKER, N. SARDESHMUKH, P.D. ET AL MATSUI, R. ALLAN, X. YIN, B. GLEASON, R. VOSE, R. G., P. BESSEMOULIN, S. BRONNIMANN, M. BRUNET, R. CROUTHAMEL, G. A.N., G. P.Y., P. JONES, M. KRUK, A. KRUGER, G. MARSHALL, M. MAUGERI, H. MOK, O. NORDLI, T. ROSS, R. TRIGO, X. WANG, S. WOODRUFF, AND S. WORLEY, *The twentieth century reanalysis project*, Quarterly J. Roy. Meteorol. Soc., 137 (2011), pp. 1–28.
- [12] D. DEE, M. BALMASEDA, G. BALSAMO, R. ENGELEN, A. SIMMONS, AND J.-N. THÉPAUT, *Toward a consistent reanalysis of the climate system*, Bulletin of the American Meteorological Society, 95 (2014), pp. 1235–1248.
- [13] D. DEE, J. FASULLO, D. SHEA, AND J. WALSH, *The climate data guide: Atmospheric reanalysis: Overview and comparison tables*. <https://climatedataguide.ucar.edu/climate-data/atmospheric-reanalysis-overview-comparison-tables>.

- [14] E. ESRI, *Shapefile technical description*, An ESRI White Paper, (1998).
- [15] B. S. GIESE AND S. RAY, *El niño variability in simple ocean data assimilation (soda), 1871–2008*, Journal of Geophysical Research: Oceans, 116 (2011).
- [16] K. GILL, *Bored? then create a planet*.
<http://web.archive.org/web/20161108172439/http://blog.apoapsys.com/2013/05/23/bored-then-create-a-planet/>, May 2013.
- [17] T. GUNARATHNE, T.-L. WU, J. QIU, AND G. FOX, *Mapreduce in the clouds for science*, in Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on, IEEE, 2010, pp. 565–572.
- [18] J. HANSEN, R. RUEDY, M. SATO, AND K. LO, *Global surface temperature change*, Reviews of Geophysics, 48 (2010).
- [19] M. HARROWER AND C. A. BREWER, *Colorbrewer.org: An online tool for selecting colour schemes for maps*, The Cartographic Journal, 40 (2003), pp. 27–37.
- [20] W. HEIDRICH AND H.-P. SEIDEL, *Realistic, hardware-accelerated shading and lighting*, in Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques, New York, NY, USA, 1999, ACM Press/Addison-Wesley Publishing Co., pp. 171–178.
- [21] H. HEWAGAMA, H. THARANGA, D. WIJESINGHE, AND K. ERLILVERL, *Algorithm optimization to select the best query execution plan for mariadb query optimizer*, (2013).
- [22] C. HOCK-CHUAN, *Java native interface (jni) - java programming tutorial*, 2014.
- [23] T. R. KARL, A. ARGUEZ, B. HUANG, J. H. LAWRIKORE, J. R. MCMAHON, M. J. MENNE, T. C. PETERSON, R. S. VOSE, AND H.-M. ZHANG, *Possible artifacts of data biases in the recent global surface warming hiatus*, Science, 348 (2015), pp. 1469–1472.
- [24] J. KEINER, S. KUNIS, AND D. POTTS, *Using nfft 3—a software library for various nonequispaced fast fourier transforms*, ACM Transactions on Mathematical Software (TOMS), 36 (2009), p. 19.
- [25] K.-Y. KIM, G. R. NORTH, AND S. S. SHEN, *Optimal estimation of spherical harmonic components from a sample with spatially nonuniform covariance statistics*, Journal of climate, 9 (1996), pp. 635–645.
- [26] G. E. KRASNER, S. T. POPE, ET AL., *A description of the model-view-controller user interface paradigm in the smalltalk-80 system*, Journal of object oriented programming, 1 (1988), pp. 26–49.
- [27] S. KUNIS AND D. POTTS, *Fast spherical fourier algorithms*, Journal of Computational and Applied Mathematics, 161 (2003), pp. 75–98.
- [28] J. H. LAWRIKORE, M. J. MENNE, B. E. GLEASON, C. N. WILLIAMS, D. B. WUERTZ, R. S. VOSE, AND J. RENNIE, *An overview of the global historical climatology network monthly mean temperature data set, version 3*, J. Geophys. Res., 116 (2011).
- [29] C. L. LAWSON, R. J. HANSON, D. R. KINCAID, AND F. T. KROGH, *Basic linear algebra subprograms for fortran usage*, ACM Transactions on Mathematical Software (TOMS), 5

- (1979), pp. 308–323.
- [30] K. LEI, Y. MA, AND Z. TAN, *Performance comparison and evaluation of web development technologies in php, python, and node.js*, in Proceedings of the 2014 IEEE 17th International Conference on Computational Science and Engineering, Washington, DC, USA, 2014, IEEE Computer Society, pp. 661–668.
 - [31] C. MARRIN, *Webgl specification*, Khronos WebGL Working Group, (2011).
 - [32] D. MERKEL, *Docker: Lightweight linux containers for consistent development and deployment*, Linux J., 2014 (2014).
 - [33] MISC., *3-j symbol*. Wikipedia, https://en.wikipedia.org/wiki/3-j_symbol, accessed February 2018, n.d.
 - [34] MISC., *Associated legender polynomials*. Wikipedia, https://en.wikipedia.org/wiki/Associated_Legendre_polynomials, accessed February 2018, n.d.
 - [35] MISC., *Clebsch-gordan coefficients*. Wikipedia, https://en.wikipedia.org/wiki/Clebsch%E2%80%93Gordan_coefficients, accessed February 2018, n.d.
 - [36] MISC., *List of logarithmic identities*. Wikipedia, https://en.wikipedia.org/wiki/List_of_logarithmic_identities, accessed February 2018, n.d.
 - [37] MISC., *Spherical harmonics*. Wikipedia, https://en.wikipedia.org/wiki/Spherical_harmonics, accessed February 2018, n.d.
 - [38] T. PETERSON AND R. VOSE, *An overview of the global historical climatology network temperature database*, Bulletin of the American Meteorological Society, 78 (1997), pp. 2837–2849.
 - [39] J. PIERRET AND S. S. SHEN, *4d visual delivery of big climate data: A fast web database application system*, Advances in Data Science and Adaptive Analysis, 9 (2017), p. 1750006.
 - [40] W. H. PRESS, *Numerical recipes 3rd edition: The art of scientific computing*, Cambridge university press, 2007.
 - [41] R. REW AND G. DAVIS, *Netcdf: an interface for scientific data access*, IEEE Computer Graphics and Applications, 10 (1990), pp. 76–82.
 - [42] C. F. ROPELEWSKI AND M. S. HALPERT, *North american precipitation and temperature patterns associated with the el niño/southern oscillation (enso)*, Monthly Weather Review, 114 (1986), pp. 2352–2362.
 - [43] N. SCHAEFFER, *Efficient spherical harmonic transforms aimed at pseudospectral numerical simulations*, Geochemistry, Geophysics, Geosystems, 14 (2013), pp. 751–758.
 - [44] C. E. SHANNON, *Communication in the presence of noise*, Proc. Institute of Radio Engineers, 37 (1949), p. 1021.

- [45] S. S. SHEN, T. M. SMITH, C. F. ROPELEWSKI, AND R. E. LIVEZEY, *An optimal regional averaging method with error estimates and a test using tropical pacific sst data*, Journal of Climate, 2 (1998), pp. 2340–2350.
- [46] S. S. P. SHEN, N. TAFOLLA, T. M. SMITH, AND P. A. ARKIN, *Multivariate regression reconstruction and its sampling error for the quasi-global annual precipitation from 1900 to 2011*, Journal of the Atmospheric Sciences, 71 (2014), pp. 3250–3268.
- [47] C. A. SMITH, G. P. COMPO, AND D. K. HOOPER, *Web-based reanalysis intercomparison tools (writ) for analysis and comparison of reanalyses and other datasets*, Bulletin of the American Meteorological Society, 95 (2014), pp. 1671–1678.
- [48] T. M. SMITH, R. W. REYNOLDS, T. C. PETERSON, AND J. LAWRIKMORE, *Improvements to noaa’s historical merged land-ocean surface temperature analysis (1880-2006)*, Journal of Climate, 21 (2008), pp. 2283–2296.
- [49] G. TEAM ET AL., *2016: Giss surface temperature analysis (gistemp)*, NASA Goddard Institute for Space Studies. Dataset accessed Feb 2018 at <https://data.giss.nasa.gov/gistemp/>.
- [50] D. W. THOMPSON AND J. M. WALLACE, *The arctic oscillation signature in the wintertime geopotential height and temperature fields*, Geophysical research letters, 25 (1998), pp. 1297–1300.
- [51] UCAR, *NetCDF-Java (version 4.5.5)*, March 2015.
- [52] VARIOUS, *Esgf - earth system grid federation*. Brochure, 2017.
- [53] VARIOUS, *ojsolve is open source java code to do mathematics, linear algebra and optimisation.*, 2017.
- [54] R. S. VOSE, D. ARNDT, V. F. BANZON, D. R. EASTERLING, B. GLEASON, B. HUANG, E. KEARNS, J. H. LAWRIKMORE, M. J. MENNE, T. C. PETERSON, ET AL., *Noaa’s merged land-ocean surface temperature analysis*, Bulletin of the American Meteorological Society, 93 (2012), pp. 1677–1685.
- [55] E. W. WEISSTEIN, *“associated legendre polynomial.” from mathworld—a wolfram web resource*. <http://mathworld.wolfram.com/AssociatedLegendrePolynomial.html>.
- [56] E. W. WEISSTEIN, *“double factorial.” from mathworld—a wolfram web resource*. <http://mathworld.wolfram.com/DoubleFactorial.html>.
- [57] E. W. WEISSTEIN, *“spherical harmonic.” from mathworld—a wolfram web resource*. <http://mathworld.wolfram.com/SphericalHarmonic.html>.
- [58] R. C. WHALEY AND J. J. DONGARRA, *Automatically tuned linear algebra software*, in Proceedings of the 1998 ACM/IEEE conference on Supercomputing, IEEE Computer Society, 1998, pp. 1–27.
- [59] T. WHITE, *Hadoop: The definitive guide*, O’Reilly Media, Inc., 2012.
- [60] U. M. WIDENIU, *MariaDB 10.0.27*, August 2016.
- [61] M. WOO, J. NEIDER, T. DAVIS, AND D. SHREINER, *OpenGL programming guide: the official guide to learning OpenGL, version 1.4*, Addison-Wesley Longman Publishing Co.,

Inc., 2003.

- [62] R. XIN AND J. ROSEN, *Project tungsten: Bringing apache spark closer to bare metal*, 04 2015.
- [63] M. ZAHARIA, *An architecture for fast and general data processing on large clusters*, Morgan & Claypool, 2016.
- [64] M. ZAHARIA, M. CHOWDHURY, T. DAS, A. DAVE, J. MA, M. MCCAULEY, M. J. FRANKLIN, S. SHENKER, AND I. STOICA, *Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing*, in Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, USENIX Association, 2012, pp. 2–2.
- [65] M. ZAHARIA, R. S. XIN, P. WENDELL, T. DAS, M. ARMBRUST, A. DAVE, X. MENG, J. ROSEN, S. VENKATARAMAN, M. J. FRANKLIN, ET AL., *Apache spark: a unified engine for big data processing*, Communications of the ACM, 59 (2016), pp. 56–65.
- [66] F. W. ZWIERS AND S. S. SHEN, *Errors in estimating spherical harmonic coefficients from partially sampled gcm output*, Climate Dynamics, 13 (1997), pp. 703–716.