OPTIMIZATIONS TO THE ORTHOGONAL MATCHING PURSUIT ALGORITHM FOR SPARSE BASIS REPRESENTATIONS OF PHOTOMETRIC REDSHIFT PDFS

By

Christopher Chan

Senior Thesis in Computer Engineering

University of Illinois at Urbana-Champaign

Advisor: Paris Smaragdis

April 2016

Abstract

In this thesis I investigate potential optimizations for the K-SVD algorithm (using Orthogonal Matching Pursuit) to create a sparse basis representation of probability density functions (PDFs), as implemented by NCSA research affiliate Matias Carrasco Kind and Professor Robert J. Brunner. The implementation these scientists engineered is currently being used to compress PDFs of photometric redshifts (i.e., distance estimates) for galaxies by about 90%. This implementation allows end-users to easily reconstruct the original PDF with accuracies better than 98%. As we continue to mine large, photometric sky surveys, photometric redshift PDF storage will need to scale appropriately; thus, meaningful advances in this algorithm's implementation will serve to demonstrably benefit our scientific ability to explore the Universe and to expand our cosmological understanding. However, the existing implementation of the algorithm is limited by run time—an issue that continues to grow more important as the amount of data surveys acquired becomes larger. The existing implementation utilizes SciPy, a scientific computing Python library. This past semester, I have explored this implementation by developing and testing alternative approaches to the core algorithms in C++, beginning with different linear algebra libraries. In my initial tests, I found that limitations in Eigen, a C++ linear algebra library, make it difficult to accurately reproduce both the results and the exaction speeds due to the optimizations that NumPy, the Python numerical library, already has implemented. Next, I pivoted to Armadillo, another C++ linear algebra library, where I discovered that the primary algorithm runs slightly quicker than its Python counterpart. This research is an ongoing project, and I am excited to continue my investigations into hardware assists, specifically in testing the efficiency of GPU-accelerated computation (NVBLAS). Once I have identified an optimization, I look forward to implementing Batch Orthogonal Matching Pursuit, an algorithm more suited for large sets of PDFs over a single dictionary, and, if time permits, an algorithm that can be extended to support two-dimensional PDF representations.

Subject Keywords: Signal Processing, PDF Compression, Photometric Redshift Estimations, NumPy, SciPy, Armadillo, Eigen

Acknowledgments

Special thanks to NCSA research affiliate Matias Carrasco Kind and Professor Robert J. Brunner for all their mentorship throughout the research.

Thank you to Professor Smaragdis for co-signing my ECE Senior Thesis.

Contents

Abbreviations/Definitions

- GPU - Graphics Processing Unit
- SVD - Singular Value Decomposition
- GCC - GNU Compiler Collection
- PDF - Probability Distribution Function
- NCSA - National Center for Supercomputing Applications
- BLAS - Basic Linear Algebra Subprograms
- NVBlas - Nvidia BLAS
- MKL - Intel’s Math Kernel Library

1. Introduction

In this chapter, I will introduce the issues surrounding the recent growth in galaxy surveys and photometric redshift PDFs, which are estimations of the cosmological redshift, or approximate distance, of galaxies by applying statistical techniques to photometric sources [1]. A redshift is an effect by which the frequency of a signal, such as a sound or light wave is affected by its' movement towards or away from the receiver [2]. More specifically, as a sound wave, for example, moves away from you, the frequency is stretched into lower frequencies [2].

1.1 Problem

As technology to mine photometric surveys continues to advance, the digital storage needed to house photometric redshift PDFs is becoming increasingly expensive. Photometric survey sizes have grown at a seemingly exponential rate. In the 1980s, the DPOSS survey was created at roughly 3 TB of data. It is expected that newer sky survey projects, such as SKA, will require data storage units of an estimated 4.6 exabytes [3]. Even further, the LSST, or Large Synoptic Survey Telescope, will observe tens of billions of objects and several billion galaxies [4]. As observations continue to grow, it has become a much more considerable challenge to store the data and process it within a reasonable amount of time. NCSA research affiliate Matias Carrasco Kind and Professor Robert J Brunner present the K-SVD algorithm (using Orthogonal Matching Pursuit) to create a sparse basis representation of probability density functions (PDFs) as a potential solution to the large storage complexity. In this paper, I will discuss my potential optimizations to Professor Brunner and Carrasco Kind's work.

1.2 Purpose

The need for photometric redshift PDFs is increasing. Large photometric surveys like the Dark Energy Survey are probing galaxies that are often too distant to be observed spectroscopically [1].

Furthermore, many cosmological measurements such as galaxy clustering, weak lensing, and more rely on the accuracy of measured distances to the galaxies; these measured distances rely on photometric redshift PDFs to get accurate results [1]. Carrasco Kind and Brunner's paper refers to a paper done by Meyers et al. that mentions that the measurement of a two-point angular quasar correlation function has been improved by nearly a factor of four by using full redshift PDFs instead of a single redshift estimation [1],[5]. In conclusion, astronomers are thus able to probe into much larger volumes of photometric survey data with these large scale estimations.

2. Core Algorithm

2.1 Introduction to Algorithm

This thesis investigates potential speed optimizations for the K-SVD algorithm (using orthogonal matching pursuit) to create a sparse basis representation of probability density functions (PDFs), as implemented by NCSA research affiliate Matias Carrasco-Kind and Professor Robert J. Brunner. Their implementation is based on an optimized version of the K-SVD algorithm, developed by the Israel Institute of Technology. This optimized version utilizes orthogonal matching pursuit with Cholesky decompositions.

2.2 K-SVD

K-SVD is an algorithm based on the k-means algorithm used to construct dictionaries for sparse representations of signals. More specifically, the algorithm creates an over-complete dictionary that contains a satisfactory amount of prototype signals, which are referred to as atoms, to represent the data [6]. Over-complete dictionaries are key to the algorithm as they allow for higher sparsity of signal representation than other options such as orthogonal basis dictionaries [71]. Signals are then able to be described by linear combinations of these atoms, thus reducing the storage complexity [6]. Mathematically, the algorithm relies on a sparsity assumption, which can be described by the following mathematical statement:

$$\gamma = Argmin_{\gamma} \ ||\gamma||_0 \ Subject\ To \ ||x - D_{\gamma}||_2^2 \leq \epsilon \ .$$

γ is the sparse representation of x, ϵ is the error tolerance, and $|| \cdot ||_0$ is the pseudo norm which counts the number of non-zero entities [6]. In other words, we are trying to find a sparse representation of x that maintains a sparse vector and maintains a low error from the original signal. One method to solve the sparse approximation problem is to utilize the orthogonal matching pursuit algorithm [8].

2.3 Orthogonal Matching Pursuit with Cholesky Decomposition

Orthogonal matching pursuit with Cholesky decomposition presents a much faster alternative to K-SVD while achieving the same results. In a paper by primary author Ron Rubinstein, a Ph.D. graduate in Computer Science at Israel Institute of Technology focusing on image and signal modeling, he reports on a more efficient approach to K-SVD approximations and the use of batch orthogonal matching pursuit to perform sparse coding operations [8]. The goal of the algorithm is to represent a PDF using a set of basis functions defined by the following mathematical expression:

$$pz_k = D\delta_k + \varepsilon_k$$

D represents the dictionary of basis functions. This can also be described as a matrix of basis functions of size n x m, where m > n. Each d_j column in the sparse matrix represents a basis function that is l_2 normalized. I.e. $\sqrt{\sum_{k=1}^{n} |x_k|^2} = 1$ [8] .

The goal of orthogonal matching pursuit is to find, for each galaxy, an optimal vector $\delta_k$that minimizes the amount of non-zero entities to maintain the sparsity constraint and reduce the amount of residual error given ε_k[9]. Orthogonal matching pursuit does this by taking a given PDF, searching through the dictionary to find the basis function that best reduces the residual vector at the current iteration, then recalculating the residual vector ε_k excluding the new basis function[9].

Mathematically, at the high level, orthogonal matching pursuit boils down to the following:

i. Define the residual vector $\varepsilon_k = pz_k$. Create an empty set of basis functions B_k, an empty vector $\delta_k$and set i = 0, which represents the current iteration [94].

ii. Compute the current set of basis functions through matrix multiplication. Here, we want to find the column vector d_b from the dictionary D that maximizes the projection of the residual vector, ϵ_k. Thus, we come across the following mathematical expression [9]:

$$d_b^i = max\ |d_j^T \cdot \varepsilon_k^i|, \text{ where } d_j \epsilon D$$

Add the basis function selected from d_b^i to the set B_k[9].

iii. Orthogonally project the original PDF with all selected basis functions, where w_k is a temporary vector holding the coefficients of the selected basis functions.

$$w_k^i = B_k^T \cdot pz_k \text{ [9]}$$

iv. Update the residual vector by using the temporary vector.

$$\varepsilon_k^{i+1} = pz_k - B_k \cdot w_k^i \text{ [9]}$$

v. Check the stopping criterion $||\varepsilon_k^{i+1}||_2 < \varepsilon_{threshold}$. If the stopping criterion is met we then set $\delta_k = w_k^i$ and finally $pz_k = D \cdot \delta_k + \varepsilon_k^{i+1}$ where δ_k is sparse. If the stopping criterion is not met, we increment i and repeat steps 2-5. Steps 1-5, are repeated for every galaxy [9].

Finally, once the sparse basis representation is complete, Carrasco-Kind and Professor Brunner's work refers to compression techniques to further reduce the size of storing these sparse vectors. In order to do so, the paper refers to a compression technique where, for each basis function chosen, it compresses the basis function so that the first 16 bits refer to the scaling factor or amplitude of the basis function, while the next 16 bits refer to the basis function selected. This is valuable as we can shrink the size down for a single basis function to just a 32 bit or 4-byte integer.

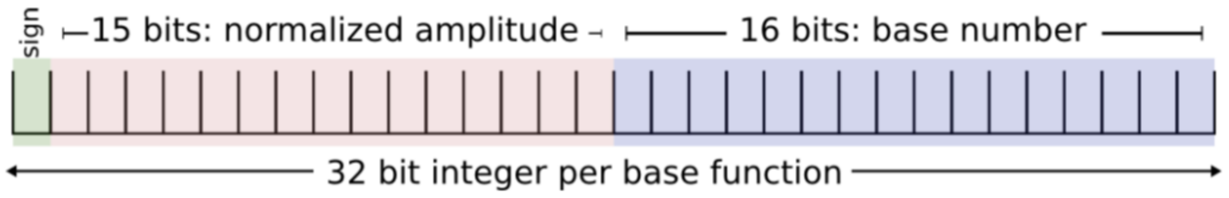
sign
15 bits: normalized amplitude
16 bits: base number
32 bit integer per base function

Figure 1: Taken from Carrasco-Kind and Professor Brunner's paper on Sparse Representation of Photometric Redshift PDFs: Preparing for Petascale Astronomy [9]. Depicts the compression technique for a single basis function.

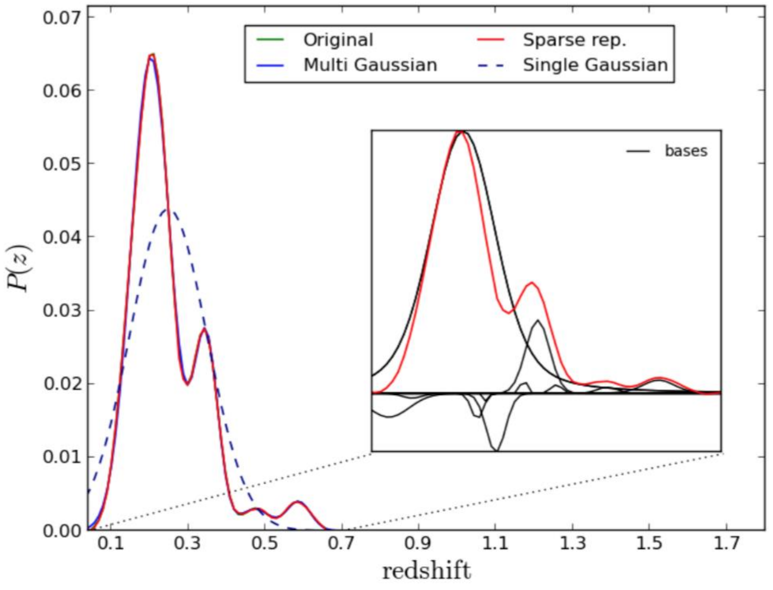

Figure 2: Taken from Carrasco-Kind and Professor Brunner's paper on Sparse Representation of Photometric Redshift PDFs: Preparing for Petascale Astronomy [9]. Depicts the bases functions selected and the sparse represented PDF.

This chapter goes over the high level design decisions I made to develop an optimized version of NCSA research affiliate Matias Carrasco Kind and Professor Robert J. Brunner's implementation of the orthogonal matching pursuit algorithm. This includes programming language decision, library decisions, and data.

3. Design Decisions

3.1 Original Implementation

The original implementation of the orthogonal matching pursuit algorithm is written in Python with NumPy/SciPy. Both NumPy and SciPy contain scientific and numerical tools for Python. Due to the fact that Python is an interpreted language, it is inherently slower than compiled languages such as C. In order to combat this, NumPy and SciPy have C implementations for many of their more algorithmically complex functions [10]. During my research, I spent a significant amount of time reading about NumPy optimizations to explain any run-time differences between NumPy and C++.

3.1.1 NumPy

NumPy relies on a very efficient memory model for its arrays that allows the programmer to assign specific portions of an array to a variable without copying data [11]. Ensuring that the code is only copying data when necessary is instrumental to ensuring an optimized run time.

NumPy also features vectorized operations that are implemented in C. As stated by Scipy's online tutorials, vectorized operations are utilized when operations are applied to every element in an entire NumPy array [12]. Van Der Falt, an author of the publication on NumPy performance, describes that instead of using a traditional for loop, one should use a vectorized operation as the operation is broadcast across the entire array for significant performance improvement [11].

For larger datasets, NumPy supports manipulating arrays stored on disk without copying all of its data to faster memory units such as RAM [11]. Van Der Falt refers to this technique as memory mapping [11]. The algorithm works by loading only the part of the large array that the programmer wants to access onto memory. It then performs any operations that the programmer requires and then, when the programmer calls flush(), flushes the modifications over to disk [11]. This is potentially quite valuable for the K-SVD algorithm for a large amount of basis functions or dictionary size that cannot be held on expensive RAM.

3.2 Choice of C++

Average Run Time (Seconds)

Programming Language	Compiler	**Time**
C++	GCC-4.9.0	0.73
Fortran	GCC-4.9.0	0.76
Fortran	Intel Fortran 14.0.3	0.95
Java	JDK8u5	1.95
Julia	0.2.1	1.92
Matlab	2014a	7.91
Python	Pypy 2.2.1	31.90
Python	Cpython 2.7.6	269.31
Mathematica	9.0,base	588.57
Matlab	2014a	1.19
Rcpp	3.3.1	2.66
Python	Numba 0.13	1.18
	Cython	1.03
Mathematica	9.0, Idiomatic	1.67

Figure 3: Describes the average run time for each programming language for the stochastic growth model from Professor Fernandez-Villaverde's work on financial programming languages [13].

For the optimized implementation this publication focuses on, the code was developed in C++ [13]. This decision was based on a variety of factors. At the high level, I investigated the differences in performance times of the various programming languages. In a paper developed by Professor Fernandez-Villaverde, he investigates the use of multiple programming languages to implement a probabilistic model known as stochastic neoclassical growth model [13]. This model relies on many similar mathematical functions that orthogonal matching pursuit requires such as max, summation, and linear algebra operations. In their study, they found that C++ using GCC-4.9.0 as a compiler maintained the fastest run time on average [13]. Although Fortran with Intel's compiler came in at a close second by only .03 seconds, this difference could cause issues as the data scales.

3.3 Choice of Linear Algebra Library

Choosing the proper linear algebra library in C++ can have a dramatic effect on run time. My decision in choosing a particular linear algebra library largely relied on investigating what optimizations and support for a wide variety of linear algebra operations the library contains. After looking through documentation, I came across two major linear algebra libraries largely recommended by my peers, Eigen and Armadillo [14],[15].

3.3.1 Eigen

Eigen presents enticing optimizations. The library supports explicit vectorization, optimizations in chained operations such as multiplication, a wide variety of compiler support, and support for other linear algebra libraries it can call underneath such as: BLAS, LaPACK, Intel MKL, etc. [16]. These libraries are highly optimized and are able to perform multi-threaded mathematical routines for x86 machines [16]. In addition, Eigen's algorithms are able to utilize multiple cores via OpenMP, a parallel programming library in C/C++ [16].

Above all, utilizing Eigen as a top level library over optimized and well documented libraries is especially valuable as Eigen's syntax and supported functions are similar to those in NumPy/SciPy. Decompositions that Orthogonal Matching Pursuit requires (Cholesky and Triangular), are both available in Eigen [16]. Furthermore, basic linear algebra functions that NumPy contains such as linspace, sum, and min have nearly identical syntax and function in Eigen [14].

3.3.2 Armadillo

Similarly, Armadillo is a library that contains efficient optimizations. Armadillo supports optimizations in chained operations, SIMD vectorisation (vectorizes elementary operations such as matrix addition into SSE2, SSE3, SSE4, or AVX instructions), and support for various high speed linear algebra such as BLAS or multithreaded libraries such as OpenBLAS, Intel MKL, or AMD ACML [15]. Furthermore, for my research as I am using a Macintosh, Armadillo supports Mac OS X's accelerate framework which leverages optimizations in the Mac architecture for further optimizations [17]. Described later, Armadillo also contains support for NVIDIA NVBLAS which is a GPU accelerated version of BLAS [15]. For computers with quick GPUs, this could allow users to run any algorithm with Armadillo even more quickly.

Table 1 Examples of syntax conversion

Armadillo	**Numpy**
A.cols(p,q)	A[:,p:q]
A[p:q,r:s]	A(span(p,q),span(r,s))

Again, Armadillo also contains syntax and functionality that closely mirror NumPy/SciPy. Armadillo's website contains documentation for syntax conversion between Armadillo and Matlab, which contains similar syntax to NumPy/SciPy [18]. Explained later in the paper, Armadillo's data structures follow NumPy/SciPy's more closely which has proven quite valuable for OMP's implementation.

3.4 Data

In this section, I introduce an overview over the data that I used to test if my results maintain the same consistency that the original implementation outputs. The dataset, provided by Dr. Brunner and Kind's Github repository, holds a subset of photo-z PDFs for different galaxies.

The dataset, saved in a NumPy data frame, originates from the Canada-France-Hawaii Telescope Lensing Survey known as CHFTLens [19]. CHFTLens is an accumulation of images over five years via a 340-megapixel camera. This sophisticated dataset contains about 10 million galaxies that are typically six billion light years away [19]. For my focus, the original implementation Drs. Brunner and Kind developed utilizes a dataset that takes 49,686 galaxies from CHFTLens and then utilizes an algorithm called TPZ which uses decision trees with random forest to develop redshift PDFs for each of the galaxies [9].

Because the data is saved in a NumPy data frame, relying on C++ standard libraries do not apply here. Instead, I rely on Cnpy, an open source C++ library that reads NumPy data frame files onto memory in an easy to use array of bytes [20].

The data is shaped by 200 floating points, each equally spaced from 0 to 2. As stated earlier, the goal of the implementation is to reduce this data size down to just 20 4-byte integers.

4. Results

I developed multiple implementations of the orthogonal matching pursuit algorithm with Numba, a Python library that allows the code to be compiled in LLVM, with Eigen, and with Armadillo [21]. I discovered that each implementation has their pros and cons that I will discuss in the following sections.

4.1 Numba

Numba is a Python library that generates optimized machine code using the LLVM compiler architecture at run time [21]. Here, I utilized Numba to compile the orthogonal matching pursuit algorithm from the original code in Python. Because most of the algorithm relies on NumPy and SciPy calls, the code is mostly optimized already, as these calls are implemented in C and maintained by an avid open source community [11]. Instead, we utilize Numba to remove the remaining Python code, the for loop. By using Numba I am thus able to optimize the loop in machine code leading to faster results.

4.2 Eigen

Eigen presented an easy way of converting the Python code to C++; however, it lacked certain functionality that led Eigen to being an invalid solution. The algorithm employs a progressive Cholesky Factorization which allows for performance boosts [8]. Thus, the algorithm relies on a Cholesky solver that, given a cholesky factorization matrix and a column vector, will solve the system of equations [8]. Because Eigen only allows the user to call the solve function when the underlying data structure is some sort of decomposition, it disallows users to call solve on a simple matrix that is assumed to already be factorized. This led to a lot of issues, until finally I realized that this was no longer a viable option.

4.3 Armadillo

Armadillo, the other C++ linear algebra library, provides a more complete implementation than Eigen provided for this problem set, but it came with its own problems [8]. Armadillo provides a very comprehensive assortment of functions that allow the user to perform various linear algebra computations. Implementing the orthogonal matching pursuit algorithm was mostly straightforward. One downside was that the Cholesky solver in Python relies on forward and back substitution to solve for the equation [22].

The function call cholesky_solve instead works as follows [22]:

If the matrix is upper triangular, $A = UU^T$where U is upper triangular, the solution X is calculated by

$U^TY = B \rightarrow UX = Y$

If the matrix is lower triangular $A = UL$where U is upper triangular, the solution X is calculated by

$LY = B \rightarrow UX = YL^TX = Y$

Thus, I had to implement this function in the order of matrix multiplication the above mathematical statements describe.

4.4 Running Time Comparisons

Here, I focus on a comparison of the performance differences between Python with Numba, Python, and C++ with Armadillo. Figure 5 shows the performance difference over an average of three iterations for each of the implementations. In the end, the C++ code presents a quicker solution, with running time improvements of about 14%. Although the test data was only one hundred galaxies/PDFS, the data sets that astronomers mine are much larger. Thus, shaving 2 seconds here may result in significant performance gains for larger, more realistic, data sets.

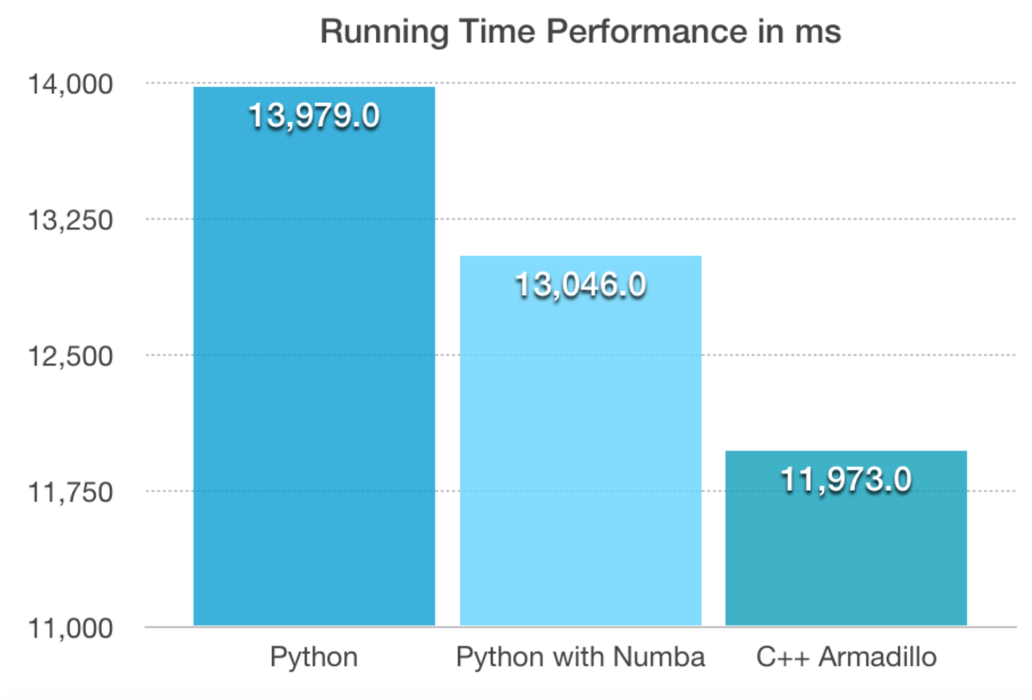

Figure 5: Average running time performance comparison of the three implementations in milliseconds.

4.5 Comparison of Sparse Basis PDF to Original PDFs

This section will focus on comparing the sparse representation of the photometric redshift PDF's across the three implementations with the original PDFs to see if there are any differences in error performance. Because Python with Numba and Python are inherently the same code, we only need to compare one of their results.

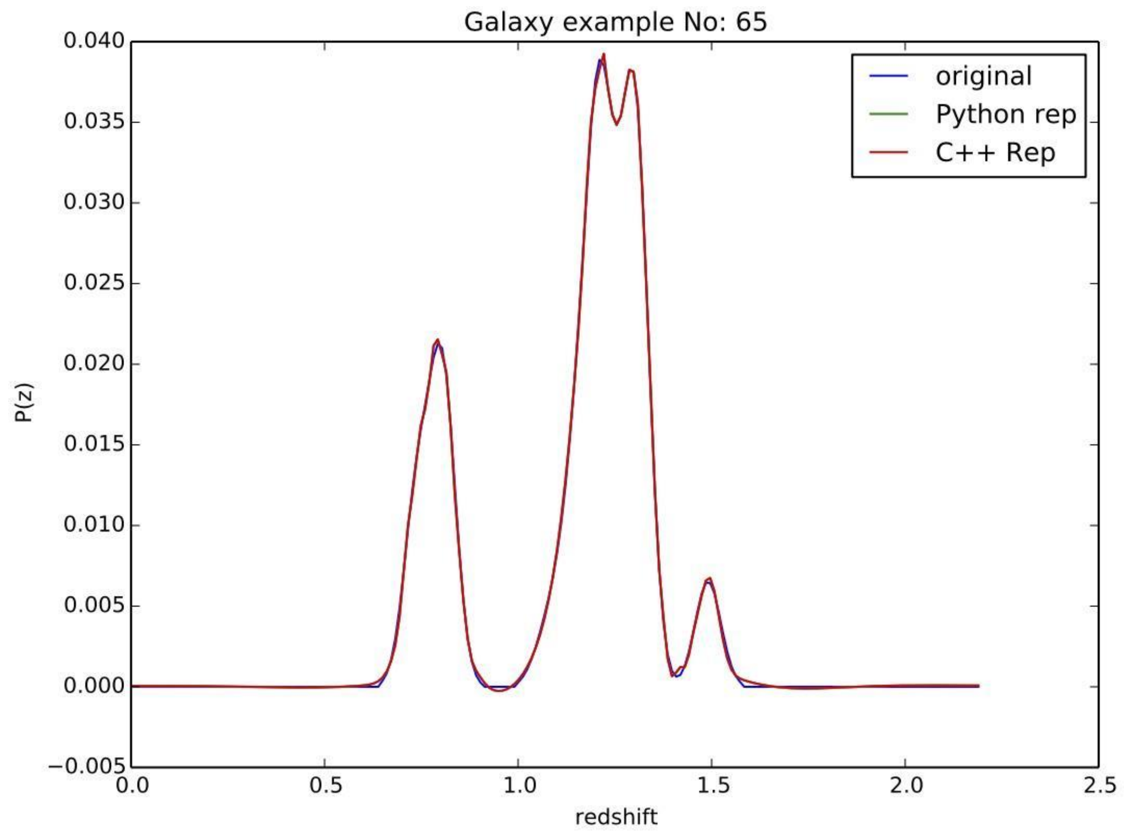

Figure 6: A random galaxy example showing the original PDF, the Python sparse representation, and the C++ sparse representation. Plot generated from Matias Kind Sparse-Z implementation [23].

Figure 7 is a difference plot for the first galaxy between the original PDF and the C++ sparse basis representation. As we can see, the sparse basis representation is not perfect, and loses some information at earlier redshift values.

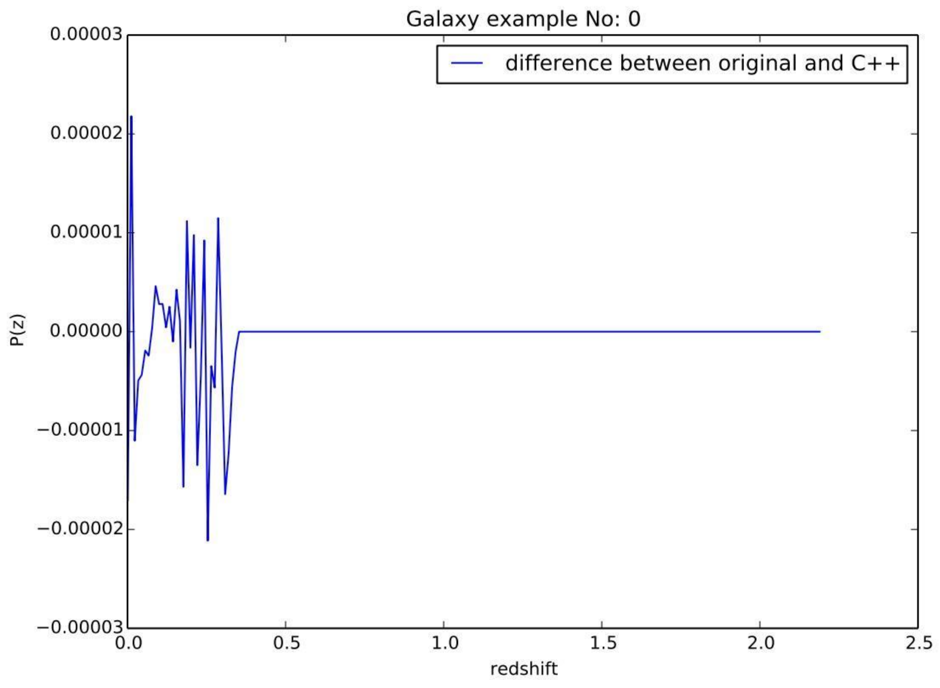

Figure 7: Difference plot between original PDF and C++ Sparse Basis representation

Figure 8 compares the mean squared error between the original PDF and C++'s sparse basis representation for each galaxy. Notice that for some PDFs there are larger MSE difference spikes than for others. This could result from the greater complexity of some signals.

Mean Squared Error Difference

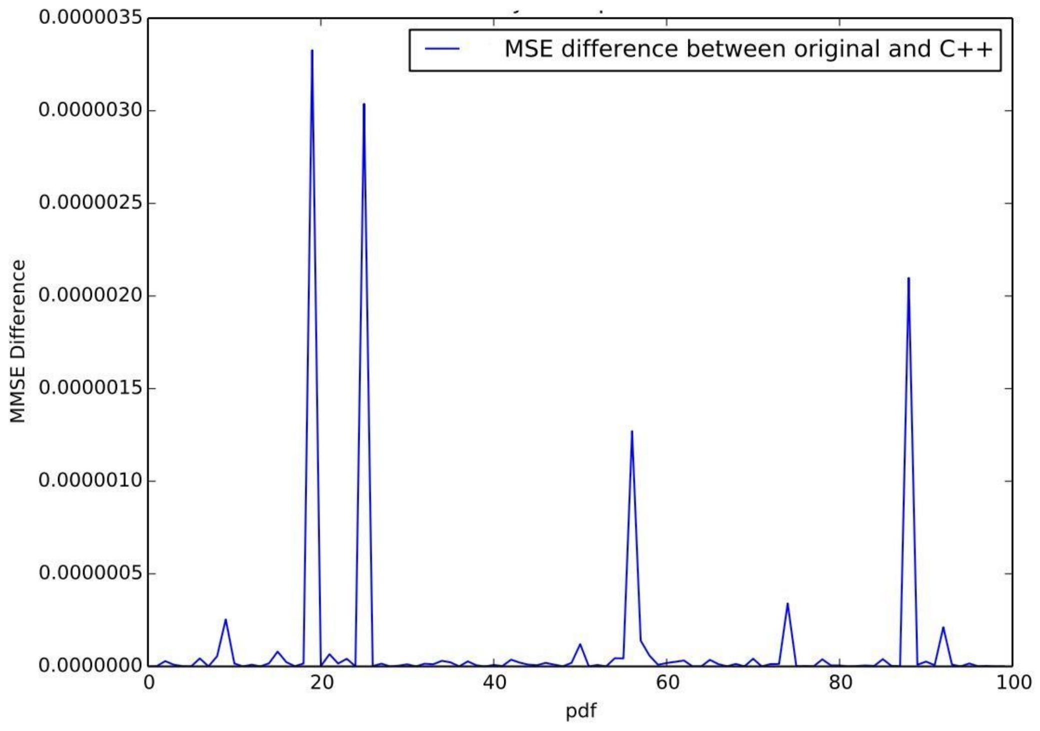

Figure 8: Mean squared error difference plot between original PDF and C++ code

For all galaxies, the aggregate mean squared error difference between the original PDF and the C++ code is$1.245347 \cdot 10^{-5}$. Interestingly enough, the original Python implementation results in an aggregate mean squared error of $1.132906 \cdot 10^{-5}$. Although the additional error in the C++ code is quite small, we believe that the code might give slightly different results in terms of the basis functions selected.

In conclusion, we notice that the C++ code is slightly less accurate than the original Python code. Although the C++ solution is quicker, the Python code remains the most accurate. However, this error may be acceptable within the errors of measured photo-z as the difference is significantly smaller than the data we are using.

5. Conclusion

The research presented in this paper is only a start to the greater project of delivering faster performance to the orthogonal matching pursuit algorithm. Although the optimized C++ code with Armadillo provides performance gains to the Python code, it is only 14%. We believe that this code could be sped up even further through a variety of other methods. In the future, I plan on continuing my work and testing out other techniques such as NVBlas, a NVIDIA GPU accelerated library, which my C++ implementation supports. Another exciting, potentially significant fix would be to test out the new beta for Intel MKL, a linear algebra library for Python. If we are able to get performance boosts in linear algebra operations by utilizing different libraries, the code could potentially be significantly faster than the original Python implementation. In the meantime, Numba provides a temporary speed boost to the problem. Although the time difference is small, we can see that if we apply this algorithm to many PDF's, there could be significant performance gains.

References

[1] M. Carrasco Kind and R. Brunner, "TPZ: photometric redshift PDFs and ancillary information by using prediction trees and random forests", *Monthly Notices of the Royal Astronomical Society*, vol. 432, no. 2, pp. 1483-1501, 2013.

[2] "Cool Cosmos", *Coolcosmos.ipac.caltech.edu*, 2016. [Online]. Available: http://coolcosmos.ipac.caltech.edu/cosmic_classroom/cosmic_reference/redshift.html. [Accessed: 26- Apr- 2016].

[3] Y. Zhang and Y. Zhao, "Astronomy in the Big Data Era", *CODATA*, vol. 14, no. 0, p. 11, 2015.

[4] L. Telescope, "LSST Scientist FAQs", *Lsst.org*, 2016. [Online]. Available: http://www.lsst.org/scientists/scientist-faqs. [Accessed: 26- Apr- 2016].

[5] A. Myers, M. White and N. Ball, "Incorporating photometric redshift probability density information into real-space clustering measurements", *Monthly Notices of the Royal Astronomical Society*, vol. 399, no. 4, pp. 2279-2287, 2009.

[6] M. Aharon, M. Elad and A. Bruckstein, "K-SVD: An Algorithm for Designing Overcomplete Dictionaries for Sparse Representation", *IEEE Transactions on Signal Processing*, vol. 54, no. 11, pp. 4311-4322, 2006.

[7] Wang Tian Jing, Zheng Bao Yu and Yang Zhen, "An overcomplete dictionary design algorithm for sparse representation of piecewise stationary signals," *2012 18th Asia-Pacific Conference on Communications (APCC)*, Jeju Island, 2012, pp. 427-430.

[8] Rubinstein, R., Zibulevsky, M. and Elad, M., Efficient Implementation of the K-SVD Algorithm using Batch Orthogonal Matching Pursuit Technical Report - CS Technion, April 2008.

[9] Matias Carrasco Kind, Robert J Brunner, “Sparse representation of photometric redshift probability density functions: preparing for petascale astronomy”, Monthly Notices of the Royal Astronomical Society, Volume 441, Issue 4, p.3550-3561, 2014.

[10] "Numpy C Code Explanations — NumPy v1.10 Manual", *Docs.scipy.org*, 2016. [Online]. Available: http://docs.scipy.org/doc/numpy/reference/internals.code-explanations.html. [Accessed: 26- Apr- 2016].

[11] S. van der Walt, S. C. Colbert and G. Varoquaux, "The NumPy Array: A Structure for Efficient Numerical Computation," in *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22-30, March-April 2011.

[12] "2.4. Optimizing code — Scipy lecture notes", *Scipy-lectures.org*, 2016. [Online]. Available: http://www.scipy-lectures.org/advanced/optimizing/index.html#profiling-python-code. [Accessed: 26- Apr- 2016].

[13] S. Boragan Aruoba, Jesus Vernandez-Villaverde. "A Comparison of Programming Languages in Economics".

[14] "Eigen Quick Reference", *Eigen.tuxfamily.org*, 2016. [Online]. Available: https://eigen.tuxfamily.org/dox/AsciiQuickReference.txt. [Accessed: 26- Apr- 2016].

[15] "Armadillo: C++ linear algebra library", *Arma.sourceforge.net*, 2016. [Online]. Available: http://arma.sourceforge.net. [Accessed: 26- Apr- 2016].

[16] "Eigen", *Eigen.tuxfamily.org*, 2016. [Online]. Available: http://eigen.tuxfamily.org/index.php?title=Main_Page. [Accessed: 26- Apr- 2016].

[17] "Accelerate(7) Mac OS X Manual Page", *Developer.apple.com*, 2016. [Online]. Available: https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man7/Accelerate.7.html. [Accessed: 26- Apr- 2016].

[18] "Armadillo Matlab Conversion Table", *Arma.sourceforge.net*, 2016. [Online]. Available: http://arma.sourceforge.net/docs.html#syntax. [Accessed: 26- Apr- 2016].

[19] "News CFHT - Astronomers reach new frontiers of dark matter", *Cfht.hawaii.edu*, 2016. [Online]. Available: http://www.cfht.hawaii.edu/en/news/CFHTLens/. [Accessed: 26- Apr- 2016].

[20] "Github Repository cnpy", *GitHub*, 2016. [Online]. Available: https://github.com/rogersce/cnpy. [Accessed: 26- Apr- 2016].

[21] "Numba — Numba", *Numba.pydata.org*, 2016. [Online]. Available: http://numba.pydata.org. [Accessed: 26- Apr- 2016].

[22] "NAG Fortran Library Routine Document", *Fortran*, 2016. [Online]. Available: http://www.nag.com/numeric/FL/manual/pdf/F07/f07hef.pdf. [Accessed: 26- Apr- 2016].

[23] M. Carrasco Kind, "mgckind/SparsePz", *GitHub*, 2016. [Online]. Available: https://github.com/mgckind/SparsePz. [Accessed: 09- May- 2016].