Summer 8-2013

Iterative Solvers for Large, Dense Matrices

Eowyn Wilhelmina Cenek
University of Southern Mississippi

The University of Southern Mississippi

ITERATIVE SOLVERS FOR LARGE, DENSE MATRICES

by

Eowyn Wilhelmina Čenek

Abstract of a Dissertation
Submitted to the Graduate School
of The University of Southern Mississippi
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

August 2013

ABSTRACT

ITERATIVE SOLVERS FOR LARGE, DENSE MATRICES

by Eowyn Wilhelmina Čenek

August 2013

Stochastic Interpolation (SI) uses a continuous, centrally symmetric probability distribution function to interpolate a given set of data points, and splits the interpolation operator into a discrete deconvolution followed by a discrete convolution of the data. The method is particularly effective for large data sets, as it does not suffer from the problem of oversampling, where too many data points cause the interpolating function to oscillate wildly. Rather, the interpolation improves every time more data points are added. The method relies on the inversion of relatively large, dense matrices to solve $A_{nn}x = b$ for x. Based on the probability distribution function chosen, the matrix A_{nn} may have specific properties that make the problem of solving for x tractable.

The iterative Shulz Jones Mayer (SJM) method relies on an initial guess, which is iterated to approximate A_{nn}^{-1}. We present initial guesses that are guaranteed to converge quadratically for several classes of matrices, including diagonally and tri-diagonally dominant matrices and the structured matrices we encounter in the implementation of SI. We improve the method, creating the Polynomial Shulz Jones Mayer method, and take advantage of the more efficient matrix operations possible for Toeplitz matrices. We calculate error bounds and use those to improve the method's accuracy, resulting in a method requiring $\mathcal{O}(n \log n)$ operations that returns x with double precision. The use of SI and PSJM is illustrated in interpolating functions and images in grey scale and color.

The University of Southern Mississippi

ITERATIVE SOLVERS FOR LARGE, DENSE MATRICES

by

Eowyn Wilhelmina Čenek

A Dissertation
Submitted to the Graduate School
of The University of Southern Mississippi
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

Approved:

James Lambers
Director

Joseph Kolibal

Jeremy Lyle

Sung Lee

Susan A. Siltanen
Dean of the Graduate School

August 2013

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Joseph Kolibal, for his support and guidance, as he guided me through the process of writing a dissertation. Without his help, this dissertation would not have been written. I am deeply grateful for the many hours he spent encouraging, advising, and editing my work. My committee members – Dr Sung Lee, Dr James Lambers, and Dr Jeremy Lyle – have kindly donated time reviewing and editing my work, which has been greatly appreciated.

I would also lik e to thank Dr C. S. Chen, who encouraged me to pursue scholarships that provided financial support, including the USM Doctoral Assistantship and the NASA/Mississippi Space Grant Consortium Fellowship, the latter of which was renewed twice, and I would like to thank both the University of Southern Mississippi and the NASA/Mississippi Space Grant Consortium for the financial support they provided.

I made many friends in the mathematics graduate program at the University of Southern Mississippi; the time spent together was much appreciated. Susan Howell and Mary Ross, you listened kindly when I had questions about students and teaching the undergraduate courses. Deanna Leggett, Jason (Leihsin) Kuo, and Alex Cibotarica, I appreciated the times we spent working together. And Guangming Yao, who graduated when I was started the writing process, and who shared freely of her time and experiences.

This dissertation would not have been possible without the encouragement of Dr Lorna Stewart, at the University of Alberta, who encouraged me all these years ago to pursue graduate school. She supervised my M. Sc. in Computer Science, and introduced me to the beauty of a clean, efficient algorithm.

Lastly, I would like to thank my husband, Staff Sergeant Hollis M. Turnage, who has supported me throughout my studies at the University of Southern Mississippi; he encouraged me to apply, he encouraged me to see it through, and he offered support and comfort while serving his country in Afghanistan. My life would have taken a different path without him.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figure

LIST OF TABLES

Table

LIST OF ABBREVIATIONS

CDF	-	Cumulative Density Function
erf(x)	-	Error function $\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$
$G(x)$	-	Gaussian or normal probability distribution function
$g(\alpha)$	-	modifying function $g(\alpha) = 2\sqrt{\alpha}/n$
PDF	-	Probability distribution function
PSJM	-	Polynomial Shulz-Jones-Mayer
PSJMwIEC	-	Polynomial Shulz Jones Mayer with Iterative Error Correction
SI	-	Stochastic Interpolation
SJM	-	Shulz-Jones-Mayer

NOTATION

General Usage and Terminology

Notation used in this dissertation represents fairly standard mathematical and computational usage. These fields tend to use different preferred notations; these different notations have been reconciled whenever possible.

We use capital letters A, $B \cdots$, to denote matrices, lower case letters such as x and y to denote vectors, and denote functions as $f(x)$. We use indices to indicate the position in a vector, so that x_i is the i-th element in the vector, or the i-th vector in a set of vectors, and differentiate based on context. Matrix elements are denoted using the lower case rather than capital letter, so that a_{jk} is the element in the j-th row and k-th column of the matrix A.

Norms are typeset using double pairs of lines so that $\|A\|$ is the norm of A, $\|A\|_\infty$ is the ∞-norm of A, and $\|x\|$ is the norm of vector x. The absolute value of numbers is denoted using single lines, so that $|-2| = 2$. The floor $y = \lfloor x \rfloor$ of x is the largest integer y such that $y \leq x$.

Chapter 1

Introduction

Simply speaking, interpolation is the process of obtaining a function $p(x)$ that fits given data points so that $p(x_i) = f(x_i)$ for the known data points $\{(x_i, f(x_i))\}$; the quality of the function is measured as $\|f(x) - p(x)\|$ [4]. A related process of approximation finds a function $p(x)$ that minimizes $\|f(x) - p(x)\|$ without the requirement that $p(x_i) = f(x_i)$. Stochastic interpolation, introduced by Howard and Kolibal [17, 18], uses an $n \times n$ row stochastic matrix A_{nn} to interpolate the data at m discrete points, where usually $m \gg n$, by evaluating $A_{mn}A_{nn}^{-1}b$, where $b = [f(x_1), f(x_2), \ldots, f(x_n)]$. The efficiency of this method relies on the ability to calculate $A_{nn}^{-1}b$ quickly and accurately, regardless of the size n.

In this dissertation we will concentrate on the computational problem of inverting A_{nn} quickly, and accurately. Of particular interest are the constraints that working in software require; unlike theoretical mathematics, we assume, unless stated otherwise, that all arithmetic is calculated using double precision arithmetic. Our ultimate goal is to calculate $A_{mn}A_{nn}^{-1}b$ both quickly and accurately.

1.1 Operation Counts, Complexity, and $\mathcal{O}(\cdot)$ Notation

We will be studying algorithms operating on $m \times n$ matrices; we will be discussing both the correctness and the complexity of the algorithms. The complexity refers to the number of steps the algorithms must perform and depends on the size of the input, whether it be matrix, vector, or both. In the study of mathematics this is usually called the operations count, whereas in the study of computing science this count is usually referred to as the run-time complexity. Specifically in this dissertation, we will be counting the number of arithmetic operations performed, and will assume that addition and multiplication each take a single unitary step. Moreover, even when dealing with large matrices, we will also be assuming that all matrices and vectors can be stored wholly in memory, so that we do not account for page swapping in and out of memory, and that each number is stored in a location of unitary size. These latter assumptions are grounded on the fact that a single double precision number can be represented using only 8 bytes; hence a $10,000 \times 10,000$ matrix requires only 8×10^8 bytes, or 763 Megabytes. Since the precision we are working

with is fixed; we do not need to account for the space required to store different values, and thus the operations count will be equivalent to the run-time complexity.

The operations count, or run-time complexity, will be calculated using $\mathcal{O}(\cdot)$ notation, which characterizes functions according to their growth rates[7]. An algorithm that requires $\mathcal{O}(n^2)$ operations, or runs in $\mathcal{O}(n^2)$ time, will require at most an^2 operations, where a, b, and c are constants. While an $\mathcal{O}(n^3)$ algorithm may be faster for small n than an $\mathcal{O}(n^2)$ algorithm, since each algorithm has its own associated constants, as n grows sufficiently large the quadratic $\mathcal{O}(n^2)$ algorithm will run faster and perform fewer operations than the cubic $\mathcal{O}(n^3)$ algorithm. Thus we say that an algorithm F with operation count $\mathcal{O}(f(n))$ is preferable over an algorithm G with operation count $\mathcal{O}(g(n))$ if there exists some integer N such that $f(n) < g(n)$ for all $n > N$.

1.2 Structuring Memory

Array based

	1	2	3	4	5
1	a	0	b	0	0
2	0	0	0	c	0
3	0	d	0	0	0
4	0	e	0	0	0
5	f	0	0	0	g

vs List based

Row	values
1	(1,a), (3,b)
2	(4,c)
3	(2,d)
4	(2,e)
5	(1,f), (5,g)

Figure 1.1: Storing a matrix in memory; the array versus the list approach.

One other aspect we should consider is the storage of matrices and vectors in memory. Generally speaking there are two ways to store a matrix M in memory; the first is to store it as a two-dimensional array of values, the second is to store the matrix as a list of columns or rows, each of which contains a list of only those values that are non-zero. There are advantages to each approach. The two dimensional array will require the full n^2 memory slots, and we can recover the value of M_{ij} in one step. However, when multiplying Mx we have to check each value of each row of M, resulting in $\mathcal{O}(n^2)$ operations. In contrast, if M contains at most k non-zero values in every row (or column), then we can calculate Mx using only $\mathcal{O}(kn)$ operations, since we only have to consider the non-zero values in each row, but finding the value of M_{ij} will require $\mathcal{O}(k)$ comparisons, since we need to check each item in the list to see if it is the entry we are searching for, and we must check every entry in that row before we can conclude that M_{ij} is zero. We can improve this count by using a heap rather than an ordered list, but that still requires log comparisons.

1.3 Accuracy Requirements and Implementations

The algorithms implemented in this dissertation are implemented using fixed precision arithmetic, although in some cases the precision is manipulated to explore the effects of the fixed precision on the results. One concern is the difference between fixed precision arithmetic, as implemented on the computer, and the perfect precision arithmetic used in abstract mathematical proofs. As we will see, the limits of fixed precision arithmetic will affect both our results, and the path we can take to reach those results.

Algorithms have been implemented on three platforms: Maple, Matlab, and C using the lapack[19], gnuscl[15], and fftw[12] libraries. There are advantages and disadvantages to using each of the three platforms. The Maple software allows the programmer to define the precision, in terms of number of digits stored and used, but trades computational speed for accuracy since each arithmetic operation has to be computed explicitly in software rather than in hardware. Matlab and the C libraries are both written using double precision accuracy, so that the precision is fixed but the arithmetic operations are performed in hardware. Matlab is an excellent package for prototyping, but when dealing with the large matrices, where $n = 10,000$ and beyond, programming in C allowed us to conserve memory and improve the time required by the software. In general, Matlab has a powerful interface which hides much of the processing work; by programming and optimizing the operations directly, it is possible to implement the same algorithms with the same accuracy using less time (but more lines of code) by programming in C.

All software is executed on a departmental machine, which is an 8-CPU machine, each an Intel Xeon E5410 processor with a 12Mb cache, with 16 Gb of memory installed, running Ubuntu 10.04, and Linux kernel 2.6.32-46-generic. The installed software includes Maple 15 and Matlab 2013a, gcc version 4.4.3, fftw 3.2.2, and gnuscl 1.15.

1.4 Summary of Accomplishments

The effort to develop methods for solving large dense matrices has received considerable attention in the last few decades, primarily because of the large number of problems which arise in this setting. The research in this dissertation was motivated by the study of stochastic interpolation which is one such problem requiring the solving of large, dense matrices, and our desire to improve the efficiency of this interpolation method by solving these structured matrix problems quickly. We provide background in Chapters 2 and 3, introducing stochastic interpolation and Toeplitz matrices respectively, and provide a brief survey of known results.

We introduce the Shulz-Jones-Mayer algorithm which will be used to invert A_{nn} in

Chapter 4, this is a key step in implementing stochastic interpolation. The Shulz-Jones-Mayer algorithm uses an initial guess X_0 and an iterative process to converge A_{nn}^{-1} under certain circumstances. We provide a sufficient condition for a good initial guess X_0 and a way to calculate X_0 for a variety of classes of matrices, proving that with these initial guesses the algorithm will in fact converge quadratically. Unfortunately, when implemented this method is slower than the lapack library function.

We overcome this limitation in Chapter 5, where we improve on the Shulz-Jones-Mayer algorithm by observing that since we are actually interested in $x_k \approx A_{nn}^{-1}b$, we can calculate x_k directly as a polynomial function, as opposed to calculating $X_k \approx A_{nn}^{-1}$ and then $x_k = X_k b$. We analyze both the complexity in terms of the operations count required and the amount of expected error, and provide an iterative extension of the method which allows us to reduce the size of the error. Thus the Polynomial Shulz-Jones-Mayer algorithm allows us to calculate x_k using $\mathcal{O}(n\log n)$ operations for Toeplitz matrices A_{nn}, whereas super fast Toeplitz solvers require $\mathcal{O}(n\log^2 n)$ operations. In contrast to Chapter 4, this is a solid improvement.

The underlying motivation in developing a fast matrix solver is the desire to solve stochastic interpolations. These problems require the solution of large, full, and structured matrices, and the structure of the matrix depends on the probability distribution function used in the stochastic interpolation. To illustrate the usefulness of the solver in solving these stochastic interpolation problems, we discuss the interpolation of single variable functions, multivariable functions, and graphical images, both gray scale and color, in Chapter 6. For each of the three types of problems, we include a clear algorithm, and a proof of the complexity, or number of operations required, for each algorithm, based on the size of the input which are the data points over which we interpolate.

Chapter 2

Stochastic Interpolation

2.1 Motivation

In this dissertation we develop a fast solver for matrix problems where the matrix is both dense and structured. This fast solver is used to improve the speed of stochastic interpolation, which relies on large, dense, structured matrices. In this chapter we introduce stochastic interpolation itself, explaining the method, and specifically the class of matrices that stochastic interpolation uses. These matrices have a very specific structure, as we discuss in Sec. 2.4, which can be tweaked to produce Toeplitz matrices. It is this structure that is crucial; the solver we develop in Chapters 4 and 5 are particularly effective for the matrices that occur in stochastic interpolation problems. We will discuss known solvers for one such set of matrices, the set of Toeplitz matrices, in Chapter 3, before discussing the solvers we have studied and improved.

2.2 The Art of Interpolation

Interpolation is the art of reconstructing a function $f(x)$ given only a finite number of discrete data points $\{(x_i, f(x_i)\}$. We call the reconstructed function $P(x)$, and require that

$$P(x_i) = f(x_i), \qquad\qquad i = 0 \ldots n.$$

A given set of data points allows for many interpolating functions; to test how well a given $P(x)$ interpolates a known $f(x)$ we consider the error

$$f(x) - P(x),$$

where $x \neq x_i$. If $f(x)$ is not known, we can sometimes estimate the error by using a subset of the known data points to calculate $P(x)$, and using the remaining points to estimate the error. Some interpolation methods, including polynomial interpolation, will increase in error as too many points are used; in these cases choosing the subset carefully is very important. We can see in both Fig. 2.1 and 2.3 that successive interpolations of the function using Lagrange polynomials and increasing the number of data points results in increasing oscillations. The error can be bounded, for smooth functions, using the result from Stoer and Bulirsch[26]:

linear:	$P(x) = a_0\Phi_0(x) + a_1\Phi_1(x) + \cdots + a_n\Phi_n(x)$
polynomial:	$P(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$
trigonometric:	$P(x) = a_0 + a_1e^{xi} + a_2e^{2xi} + \cdots + e_n^{nxi}$
rational:	$P(x) = \dfrac{a_0 + a_1x + a_2x^2 + \cdots + a_nx^n}{b_0 + b_1x + b_2x^2 + \cdots + b_mx^m}$
	where Φ is defined using both $n+1$ a_i's and $m+1$ b_j's as parameters.
exponential:	$P(x) = a_0e^{\lambda_0 x} + a_1e^{\lambda_1 x} + a_2e^{\lambda_2 x} + \cdots + a_ne^{\lambda_n x}$
	where Φ is defined using $n+1$ a_i's and λ_i's as parameters.

Table 2.1: The relationship between $P(x)$ and Φ in various interpolation schemes.

Theorem 2.2.1. *If the function f has an (n+1)st derivative, then for every argument $\bar{x}$ there exists a number ξ in the smallest interval $I[x_0,\ldots,x_n,\bar{x}]$ which contains $\bar{x}$ and all locations x_i, satisfying*

$$f(\bar{x}) - P(\bar{x}) = \frac{\omega(\bar{x})f^{(n+1)}(\xi)}{(n+1)!}$$

where

$$\omega(x) = \prod_{i=0}^{n}(x - x_i).$$

This result implies that if f is smooth the error can be bounded, although it maybe bounded by ∞. Runge proved, in 1901, that using polynomial interpolation and equally spaced points, the error in interpolating the Runge function of Fig. 2.3 will grow to ∞ near the endpoints $\pm - 5$ [29], which we can see begin to develop as the number of data points grows from 3 to 9 in Fig. 2.3.

Polynomial interpolation is straightforward to calculate, and to analyze, but as we can see from the above examples, there are some definite weaknesses. Further interpolation schemes have been devised, including trigonometric, rational, and exponential interpolation [4]. Consider a family of single variable functions $\Phi(x;a_0,\ldots a_n)$, whose values a_i are determined so that $f(x_i) = P(x_i)$, and $P(x)$ is defined in terms of $\Phi(x;a_0,\ldots a_n)$, based on the interpolation method used. Table 2.1 lists some commonly used interpolation schemes.

Kolibal and Howard introduce stochastic interpolation in [17, 18]; this is an interpolation scheme that relies on a probabilistic weighting of the data combined with discrete deconvolution and convolution operators. A strong advantage of stochastic interpolation is

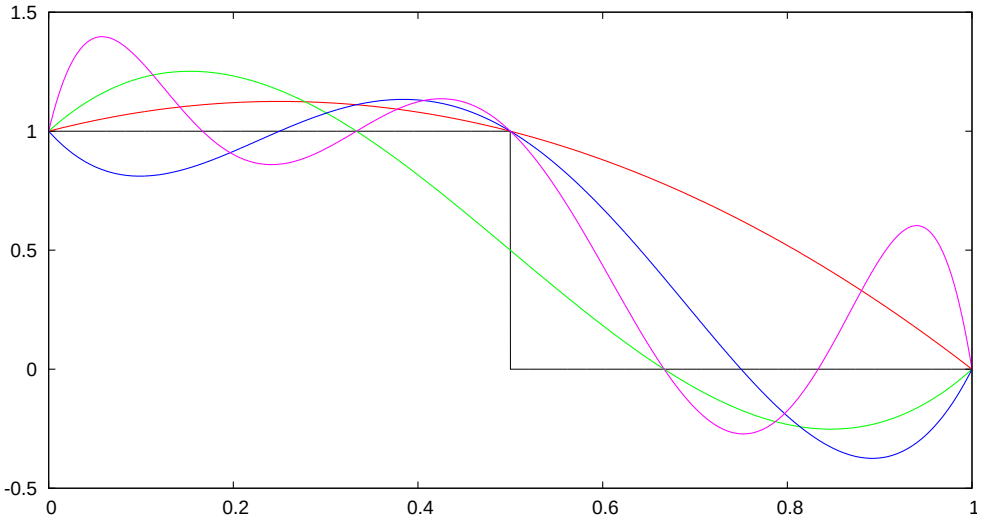


Figure 2.1: The function $f(x) = 1 - H(x - 1/2)$, $x \in [0,1]$, interpolated using Lagrange polynomials, showing the effect of using 3, 4, 5, or 7 data points.

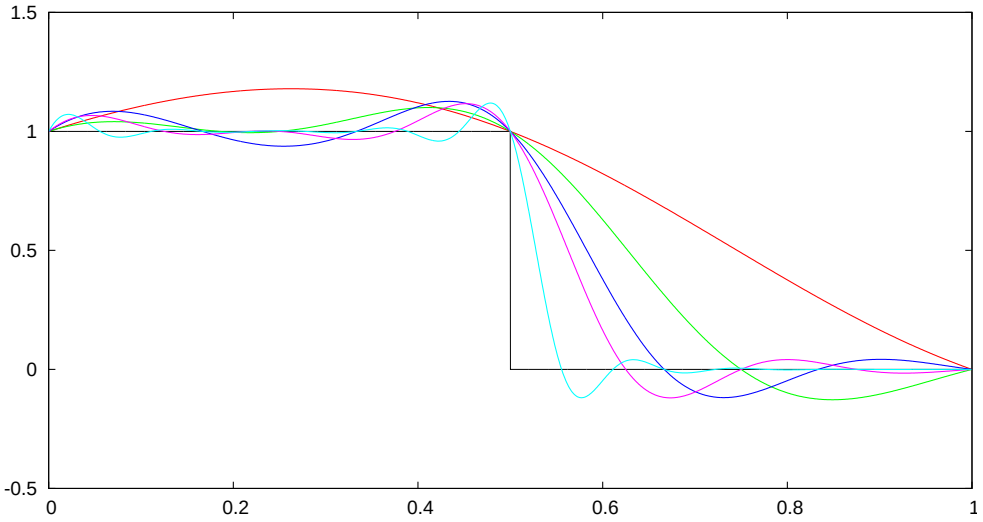


Figure 2.2: The step function $f(x) = 1 - H(x - 1/2)$, $x \in [0,1]$, interpolated using stochastic interpolation, showing the effect of using 3, 5, 7, 9, or 19 data points.

that it does not suffer from the excessive oscillation resulting from over-sampling that we see in Fig. 2.1 and 2.3. When using stochastic interpolation, increasing the number of data points, as seen in Fig. 2.2 and 2.4, improves the quality of the interpolating function $P(x)$. Stochastic interpolation is a deconvolution-convolution process, that can be calculated using two finitely sized matrices A_{mn} and A_{nn}. We provide a brief overview of the method in the next section.

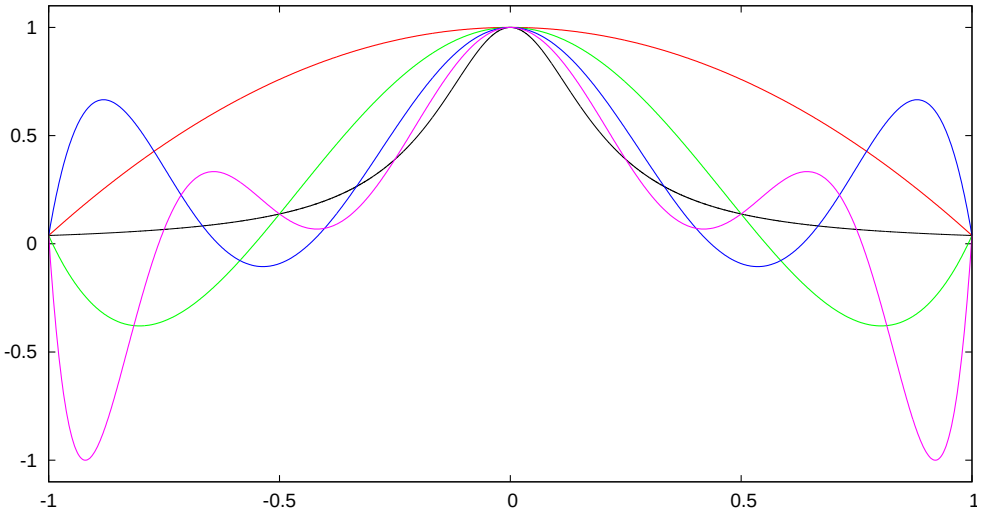


Figure 2.3: The Runge function $f(x) = 1/(1+25x^2)$, $x \in [-5,5]$, interpolated using Lagrange polynomials, showing the effect of using 3, 5, 7, or 9 data points.

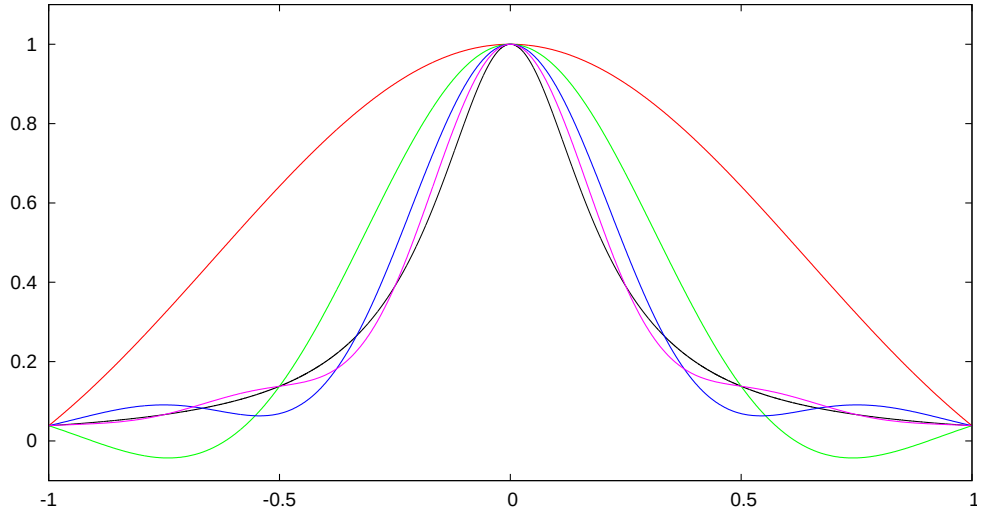


Figure 2.4: The Runge function $f(x) = 1/(1+25x^2)$, $x \in [-5,5]$, interpolated using stochastic interpolation, showing the effect of using 3, 5, 7, or 9 data points.

2.3 Constructing the Discrete Deconvolution and Convolution Operators

For a function f that we wish to approximate, we have a set of n sampling points $\{(x_k, f(x_k))\}$, $k = 1, \ldots, n$. We use these points to construct the function $K_n(x; f; \alpha)$ using the set of $n+1$

points $\{(y_k, f(x_k))\}$, where

$$y_k = \begin{cases} -\infty & k = 0 \\ \dfrac{x_k + x_{k+1}}{2} & k \in \{1, \ldots, n-1\} \\ \infty & k = n \end{cases}. \tag{2.1}$$

To do so we assume that f is piece wise constant over the intervals (y_{k-1}, y_k), and that $f(x) = f(x_0)$ for all $x < x_0$, and likewise $f(x) = f(x_n)$ for all $x > x_n$. Then $K_n(x; f; \alpha)$ is an extension of the Bernstein polynomials by changing the distribution from a binomial distribution to the Gaussian probability density function $G(x)$ so that

$$K_n(x; f; \alpha) = \sum_{k=1}^{n} \frac{f_k}{2} \left[G\left(\frac{y_{k+1} - x_j}{2g(\alpha)} \right) + G\left(\frac{y_k - x_j}{2g(\alpha)} \right) \right] \tag{2.2}$$

where $f_k = f(x_k)$ and $g(\alpha)$ is a smoothing function such that $g(\alpha(x)) > 0$ for all x. Typically $g(\alpha) = 2\sqrt{\alpha}/n$ with $0 < \alpha \leq 0.5$ being a constant, although this parameter may be modified depending on the problem domain. Note that K_n approaches the standard Bernstein polynomial as $n \to \infty$ for $g(\alpha) = \sqrt{x(1-x)}$ if G is the Gaussian probability density function [20].

We can view this approximation K_n as a linear, discrete operator acting on the data vector $f = (f_1, f_2, \ldots, f_n)$ and can recast this as a matrix vector multiplication. Let $\{x_j\}$, for $j = 1, \ldots, m$ be the set of points at which we want to evaluate K_n. Then we can define the matrix A_{mn} as

$$a_{jk} = \left[G\left(\frac{y_{k+1} - x_j}{2g(\alpha)} \right) - G\left(\frac{y_k - x_j}{2g(\alpha)} \right) \right] \tag{2.3}$$

and $K_m(x; f; \alpha) = A_{mn} f$. Observe that row i uses the Gaussian, or normal, distribution centered at x_i, as the x_i is fixed for each row i, and a_{ii} is the midpoint of the distribution so that $\sum_{k=1}^{i-1} a_{ik} < 0.5$, $\sum_{k=i+1}^{n} a_{ik} < 0.5$, and a_{ii} depends on the height of the distribution curve, which in turn is controlled by the variance $g(\alpha)$.

We can extend this approximation using a deconvolution-convolution process, where K_n constitutes a discrete convolution of the data $\{(x_k, f_k)\}$. To construct the deconvolution operator, we create the pre-image $p = (p_1, p_2, \ldots, p_n)$ of the data so that $A_{nn} p = f$. Hence the pre-image $\{(x_k, p_k)\}$ is the set of points that when approximated using K_n convolve to our original data $\{(x_k, f_k)\}$, and we use this pre-image to construct our interpolating function $K_m(x; p)$. Note that $K_n = A_{nn} f$ implies that $A_{nn}^{-1} K_n = A_{nn}^{-1} A_{nn} f = f$ so that $p = A_{nn}^{-1} f$, and hence

$$\begin{aligned} K_m(x; f; \alpha) &= A_{mn} p \\ &= A_{mn} A_{nn}^{-1} f. \end{aligned} \tag{2.4}$$

2.4 The A_{nn} Matrix

Interestingly, A_{nn} depends solely on the original sampling points and can be reused for any interpolation that uses the same number and spacing of data points, leading us to consider the goal of inverting A_{nn} quickly and efficiently. Thus, when the goal is to calculate $A_{mn}A_{nn}^{-1}f$ quickly, the key calculation is the calculation of $p = A_{nn}^{-1}f$. To calculate p and $A_{mn}p$ quickly we can take advantage of the structure of the matrices involved.

2.4.1 Centrosymmetric Matrices

We define the counter-identity matrix J as the square matrix whose entries are all zero, with the exception of the entries on the counter-diagonal, which are all 1. Then multiplying a matrix on the left by J results in reserving the rows of A, and multiplying by J on the right results in reversing the columns of A. A matrix A is centrosymmetric if $JAJ = A$, i.e.

$$JAJ = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ h & g & f & e \\ d & c & b & a \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ h & g & f & e \\ d & c & b & a \end{bmatrix} = A.$$

If the data points used to construct A_{nn} are evenly spaced, and the probability distribution function used is centrally symmetric, then the resulting row stochastic matrix is centrosymmetric. Hence if we consider solving $A_{nn}x = b$, we should first consider solving centrosymmetric matrices. Andrew [1] presents an algorithm which takes advantage of the structure of A_{nn}; by reducing the problem in size, a centrosymmetric matrix C can be decomposed so that

$$C = \begin{bmatrix} A & BP \\ PB & PAP \end{bmatrix},$$

where P is the matrix with ones only on the secondary diagonal, and zeroes elsewhere. Thus $P^2 = I$ and we can solve $Cx = y$ as

$$Cx = \begin{bmatrix} A & BP \\ PB & PAP \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix},$$

where the problem is reduced to solving

$$(A+B)z_1 = y_1 + Py_2$$
$$(A-B)z_2 = y_1 - Py_2,$$

and then set $x_1 = (z_1 + z_2)/2$ and $x_2 = P(z_1 - z_2)/2$. This algorithm still relies on some other algorithm, whether it be Gaussian elimination or some faster algorithm taking advantage of the structure of A and B, to solve the $n/2 \times n/2$ sub matrices. Hence the major speed improvement is found in either solving the sub matrices faster or speeding up the matrix-vector multiplications, as in Melman [21] and Fassbender and Ikramov [11]. Still, the bottleneck of solving $(A+B)z_1 = y_1 + Py_2$ and $(A-B)z_2 = y_1 - Py_2$ remains.

2.4.2 Toeplitz and Near Toeplitz Matrices

Toeplitz matrices are discussed in Chapter 3; essentially a Toeplitz matrix is constant along its diagonals. Consider the matrices

$$T = \begin{bmatrix} 0.5 & 0.4 & 0.3 & 0.2 & 0.1 \\ 0.4 & 0.5 & 0.4 & 0.3 & 0.2 \\ 0.3 & 0.4 & 0.5 & 0.4 & 0.3 \\ 0.2 & 0.3 & 0.4 & 0.5 & 0.4 \\ 0.1 & 0.2 & 0.3 & 0.4 & 0.5 \end{bmatrix}, \qquad M = \begin{bmatrix} 0.25 & 0.24 & 0.23 & 0.22 & 0.21 \\ 0.4 & 0.5 & 0.4 & 0.3 & 0.2 \\ 0.3 & 0.4 & 0.5 & 0.4 & 0.3 \\ 0.2 & 0.3 & 0.4 & 0.5 & 0.4 \\ 0.21 & 0.22 & 0.23 & 0.24 & 0.25 \end{bmatrix}.$$

T is a Toeplitz matrix but M is merely near-Toeplitz; the first and last row do not match the rest of the pattern because the values differ from those along the diagonals between them. If A_{nn} is initialized using evenly spaced x_i and the y_k values defined in (2.1), then the resulting matrix will be near-Toeplitz and row-stochastic. Consider for instance $A_{6,6}$ with $g(\alpha) = \sqrt{0.2}$;

$$A_{6,6} = \begin{bmatrix} 0.785 & 0.215 & 8.85e-03 & 3.86e-05 & 1.57e-08 & 5.59e-13 \\ 0.206 & 0.571 & 0.206 & 8.81e-03 & 3.86e-05 & 1.57e-08 \\ 8.81e-03 & 0.206 & 0.571 & 0.206 & 8.81e-03 & 3.86e-05 \\ 3.86e-05 & 8.81e-03 & 0.206 & 0.571 & 0.206 & 8.81e-03 \\ 1.57e-08 & 3.86e-05 & 8.81e-03 & 0.206 & 0.571 & 0.206 \\ 5.59e-13 & 1.57e-08 & 3.86e-05 & 8.85e-03 & 0.215 & 0.785 \end{bmatrix}.$$

we observe that $A_{6,6}$ is a positive, near-Toeplitz, row-stochastic, diagonally dominant matrix where the values are constant along the diagonal except in the first and last row, and where each row sums up to one. These properties will hold for any matrix A_{nn}; the difference in the first and last row is caused by the definition of $y_0 = -\infty$ and $y_n = \infty$, which introduces a non-even spacing of y_k values. Consider the probability distribution curve; using the current spacing of y-values as defined in (2.1), we effectively use a histogram where all columns except the first and last have constant width, but the first and last columns spread to a width of $-\infty$ and ∞ respectively, and the area under the total curve is guaranteed to be one.

If we redefine $y_0 = x_1 - (x_2 - x_1)/2$ and $y_n = x_n + (x_n - x_{n-1})/2$, the values of y_n are all evenly spaced, and $a_{j,k} = a_{j-1,k-1}$ in (2.3), so that A_{nn} will be a Toeplitz matrix but will

no longer be row-stochastic, since now the sum of each row represents the area under the curve of only a portion of the distribution, and as the x_i shift from left to right, the area considered will no longer be entirely centered. The effect of modifying A_{nn} thus is negligible, particularly if n is large. Considering that we are trying to solve $K_n = A_{nn}^{-1} f$, modifying A_{nn} before inversion will not affect the condition that $K_n(x) = f(x)$ for all original data points $(x, f(x))$.

Next, consider the diagonal structure of A_{nn}. The matrix A_{nn} depends on $g(\alpha)$ which in turn controls the narrowness of the distribution curve. Therefore the matrix A_{nn} will not always be diagonally dominant. However, if $g(\alpha)$ is a constant function, then the main diagonal $\{a_{ii}\}$ will always contain the largest value of all the diagonals, and the magnitude of the diagonal can be manipulated by varying the value of $g(\alpha)$; as $g(\alpha) \to 0$, the diagonal value $a_{ii} \to 1$. Using fixed precision arithmetic, this implies that as $a_{ii} \to 1$, fewer of the sub- and super-diagonals will have non-zero values. Using perfect precision, all entries in the matrix would be positive and non-zero, but as $|i - j|$ grows, and we consider values further away from the main diagonal, a_{ij} and a_{ji} will both shrink to zero. Thus the value of $g(\alpha)$ may affect the domain of influence of adjacent points when using fixed precision arithmetic; as $g(\alpha) \to 1$, more diagonals will have non-zero values, but the magnitude of the main diagonal will decrease, and conversely, as $g(\alpha) \to 0$, the magnitude of the main diagonal will increase but the number of non-zero entries in the matrix will decrease. In short; the more centrally peaked the distribution, i.e. as α becomes small, the fewer adjacent points we use in the interpolation and the flatter the curve, as α grows larger, the more adjacent points we use, but each is weighted less heavily.

2.5 Considering Other Probability Density Functions

The Gaussian probability density function is a symmetric probability density function, defined by its midpoint μ and variance α^2. The advantage of stochastic interpolation is precisely that it only requires a stochastic matrix, which means that any suitable probability density function can be utilized to generate the row-stochastic matrix. Other useful probability functions that are useful include the Laplace and Cauchy probability distributions. The Laplace probability density function has the advantage of being very narrowly distributed around μ, allowing greater control of the shape of the interpolant when working with rough data. Moreover, the advantage of the stochastic framework is that any combination of probability density functions, properly normalized, is also a probability density function and can also be used. Typically probability functions that are considered are continuous.

2.6 Summary and Assumptions

Matrices produced by the method of stochastic interpolation are large, dense, and very specifically structured, and can be tweaked to be Toeplitz matrices if the data consists of evenly spaced data points and the probability density function is centrally symmetric, as it is for both the Guassian and Laplacian probability density functions. Stochastic interpolation can be applied in many fields, and we are specifically interested in applying it to the problem of interpolating images.

For purposes of examining the Polynomial Shulz-Jones-Mayer method in the context of stochastic interpolation, we will use the Gaussian and Laplacian probability density function. We will assume that data points are evenly spaced, so that we can take advantage of the structural properties of A_{nn}. The main application that we consider is the interpolation of images. Images are typically sampled on a fixed grid, either in gray scale or in color, so that our assumption of evenly spaced data points is plausible. We also assume that the interpolant K_m has more points than the original data set, so that $m > n$.

Chapter 3

Solving Toeplitz Matrices

3.1 Motivation

In this chapter we discuss a specific class of matrices which is the set of Toeplitz matrices. These matrices occur in a large set of problems, and are correspondingly well-studied, and we provide a brief survey of known results of the problem of solving Toeplitz matrix problems. One of the advantages of using circulant matrices is that the matrix vector multiplication operator can be performed relatively quickly and efficiently, and we describe how to do so in Sec. 3.2.1. Since Toeplitz matrices can be embedded in circulant matrices, we can use this quick FFT-based method of multiplying matrices and vectors for our own improved solver, as discussed in Chapter 5.

3.2 Using Toeplitz Matrices

General solvers that solve $Ax = b$ for x, where A is an arbitrary matrix, can be improved if A is a matrix of a particular class. In this chapter we briefly discuss Toeplitz matrices, including both their properties and known results about solving $Ax = b$ for x when A is a Toeplitz matrix.

An $n \times n$ matrix T is a Toeplitz matrix if it is constant along its diagonals, i.e.

$$T_n = \begin{bmatrix} t_0 & t_{-1} & \dots & t_{2-n} & t_{1-n} \\ t_1 & t_0 & \dots & t_{3-n} & t_{2-n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ t_{n-2} & t_{n-3} & \dots & t_0 & t_{-1} \\ t_{n-1} & t_{n-2} & \dots & t_1 & t_0 \end{bmatrix},$$

and can be defined by the vector $t = [t_{n-1}, t_{n-2}, \dots, t_{-1}, t_0, t_1, \dots, t_{2-n}, t_{1-n}]$, which has length $2n-1$. A circulant matrix is a Toeplitz matrix such that $t_j = t_{j-n}$, so that

$$C_n = \begin{bmatrix} v_0 & v_{n-1} & \dots & v_2 & v_n \\ v_1 & v_0 & \dots & v_3 & v_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ v_{n-2} & v_{n-3} & \dots & v_0 & v_{n-1} \\ v_{n-1} & v_{n-2} & \dots & v_1 & v_0 \end{bmatrix},$$

can be defined by the vector $v = [v_0, v_1, \ldots, v_{n-1}]$, which has length n. We can denote C_n by $C_n(v)$, where v is the generating vector. A banded Toeplitz matrix is a Toeplitz matrix where only the diagonal and the the first r super- and sub-diagonals are non-zero. Generally $r \ll n$, so that the matrix is effectively sparse.

Toeplitz matrices occur in a large range of applications, including signals and image processing, the numerical solutions of partial differential and integral equations, and queuing networks. Depending on the application, the matrix may be more restricted; it may be required to be symmetric, or positive definite, or generated by real or complex generating functions. Gray, in his review of Toeplitz and circulant matrices [14], applies the known solvers he describes to the study of discrete random time processes. Signal processing is a huge field, and many textbooks have been written about the topic. Typically, one section is devoted to the study of Toeplitz matrices and the Fast Fourier Transform, while the rest of the book covers the manipulation of the data and interpretation of the results. Examples include both Dietrich's [10] review of signal processing for wireless communications, as well as Benesty, Sondhi, and Huang's [2] Handbook of Speech Processing (Springer), as well as many others.

One algorithm which is consistently covered, given its ability to speed up the signal processing applications, is the Fast Fourier Transform, and we will briefly describe how the FFT can be used to improve the efficiency of matrix vector multiplications if the matrix is Toeplitz or circulant.

3.2.1 Matrix Vector Multiplies and Fourier Transforms

Circulant matrices have a specific property, fast matrix vector multiplication, as described in [13] where it is shown that

$$C(v) = F_n^{-1}\text{diag}(F_n v)F_n, \tag{3.1}$$

where $C(v)$ is the circulant matrix of size $n \times n$ defined by v, and F_n is the discrete Fourier transform (DFT) matrix of order n. F_n is defined by

$$F_n = [f_{jk}], \qquad f_{jk} = \omega_n^{jk},$$

where

$$\omega_n = e^{-2\pi i/n} = \cos(2\pi/n) - i\sin(2\pi/n),$$

so that ω_n is an n-th root of unity as $\omega_n^n = 1$.

Thus the discrete Fourier transform of a vector b would be the vector $F_n b$. Generally, matrix-vector multiplications require $\mathcal{O}(n^2)$ operations. However, the structure of F_n is such that the matrix-vector multiplications $F_n b$ and $F_n^{-1} b$ can be performed recursively, requiring the far fewer $\mathcal{O}(n \log n)$ operations. Thus, the product $y = C(v)x$ can be calculated using the following four steps

$$\begin{aligned} \tilde{x} &= F_n x \\ \tilde{v} &= F_n v \\ z &= \tilde{x}. * \tilde{v} \\ y &= F_n^{-1} z, \end{aligned}$$

where $z = \tilde{x}. * \tilde{v}$ is calculated piece-wise, so that $z_i = \tilde{x}_i \tilde{v}_i$. Thus calculating z requires $\mathcal{O}(n)$ operations, and calculating each of the other three steps requires $\mathcal{O}(n \log n)$ operations, resulting in $\mathcal{O}(n \log n)$ operations overall.

We can use this result to calculate $y = T(t)x$, by augmenting the vector x to create the vector x' of length $2n-1$, by setting $x'_i = x_i$ if $0 \leq i < n$ and $x'_i = 0$ if $i \geq n$, and then embedding the $n \times n$ Toeplitz matrix T in a $2n-1 \times 2n-1$ circulant matrix $C(v)$ by setting $v = [t_0, t_1, \ldots, t_{n-1}, t_{1-n}, t_{2-n}, \ldots t_{-1}]$. Then $y' = C(v)x'$, and $y = [y'_0, y'_1, \ldots, y'_{n-1}]$.

As an example, consider the 3×3 matrix T, and its embedding in the 5×5 matrix C

$$T = \begin{bmatrix} 1 & 5 & 4 \\ 2 & 1 & 5 \\ 3 & 2 & 1 \end{bmatrix} \qquad C = \begin{bmatrix} 1 & 5 & 4 & 3 & 2 \\ 2 & 1 & 5 & 4 & 3 \\ 3 & 2 & 1 & 5 & 4 \\ 4 & 3 & 2 & 1 & 5 \\ 5 & 4 & 3 & 2 & 1 \end{bmatrix}$$

so that

$$y = \begin{bmatrix} 1 & 5 & 4 \\ 2 & 1 & 5 \\ 3 & 2 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_0 + 5x_1 + 4x_2 \\ 2x_0 + x_1 + 5x_2 \\ 3x_0 + 2x_1 + x_2 \end{bmatrix}$$

and

$$y' = \begin{bmatrix} 1 & 5 & 4 & 3 & 2 \\ 2 & 1 & 5 & 4 & 3 \\ 3 & 2 & 1 & 5 & 4 \\ 4 & 3 & 2 & 1 & 5 \\ 5 & 4 & 3 & 2 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} x_0 + 5x_1 + 4x_2 \\ 2x_0 + x_1 + 5x_2 \\ 3x_0 + 2x_1 + x_2 \\ 4x_0 + 3x_1 + 2x_2 \\ 5x_0 + 4x_1 + 3x_2 \end{bmatrix}.$$

Calculating y' will require $\mathcal{O}((2n-1)\log(2n-1)) = \mathcal{O}(n \log n)$ operations.

3.3 Toeplitz Solvers

In developing a Toeplitz solver, which solves $Tx = b$ for x, there are two main approaches. The direct solver attempts to solve x directly, by inverting T and calculating $x = T^{-1}b$. The indirect case, in contrast, typically uses pre-conditioners to modify T and make the problem more malleable. Here we present a brief survey of some known solvers; this is a very mature field.

Recursive Algorithms

Recursive methods for solving Toeplitz matrices have been known for quite some time; one early paper was published in 1964 by Trench [27] which provided an algorithm for solving Hermitian Toeplitz matrices. The algorithm was expanded by Zohar [31, 32], who showed that it could be used to solve any non-Hermitian Toeplitz matrices as long as the principal sub-matrices were all non-singular.

Trench's recursive method, which relied on solving subsets of the problem using principal sub-matrices, was adapted in what are known as Levinson-type algorithms. As discussed by Rost and Heinig in [16], these can be used to solve $T_n x = b$ for strongly non-singular matrices; T_n is strongly non-singular when both the matrix and all its principal sub-matrices are non-singular. If $T_k = [v_{i-j}]_{i,j=1}^{k}$, then T_k will occur twice as a substructure of T_{k+1} since

$$T_{k+1} = \begin{bmatrix} T_k & * \\ * & * \end{bmatrix} = \begin{bmatrix} * & * \\ * & T_k \end{bmatrix}.$$

Consider the two equations $T_k x_k^- = e_1$ and $T_k x_k^+ = e_k$. Then x_k^- and x_k^+ are the first and last columns of T_k^{-1}. Moreover

$$T_{k+1} \begin{bmatrix} x_k^- & 0 \\ 0 & x_k^+ \end{bmatrix} = \begin{bmatrix} e_1 & \gamma_k^- \\ \gamma_k^+ & e_k \end{bmatrix}$$

where $\gamma_k^- = [v_{-1}, \ldots, v_{-k}]x_k^+$ and $\gamma_k^+ = [v_k, \ldots, v_1]x_k^-$. This allows us to introduce the 2×2 matrix

$$\Gamma_k = \begin{bmatrix} 1 & \gamma_k^- \\ \gamma_k^+ & 1 \end{bmatrix}$$

which is both non-singular and easily invertible. This results in the following recursive result:

Theorem 3.3.1. *For $k = 1, 2, \ldots, n-1$, the vectors $x_k^{\pm}$ satisfy the recursion*

$$\begin{bmatrix} x_{k+1}^- & x_{k+1}^+ \end{bmatrix} = \begin{bmatrix} x_k^- & 0 \\ 0 & x_k^+ \end{bmatrix} \Gamma_k^{-1}.$$

This recursion can be started with $x_1^{\pm} = \dfrac{1}{v_0}$.

While the number of operations required for each step in the recursion is linear, there are n recursions total, resulting in an algorithm requiring $\mathcal{O}(n^2)$ operations. If the matrix T_n is symmetric or Hermitian, it is possible by dint of careful calculation to reduce the constants in the operation count, but the resulting algorithm remains $\mathcal{O}(n^2)$, as it is limited by the fact that the full n recursions need to be considered, and that every step involves vector operations, each of which require $\mathcal{O}(n)$ operations.

In contrast, the Schur algorithm solves Toeplitz matrices, among its many applications, by producing an LU decomposition of T_n. Once that decomposition is calculated, calculating $T_n^{-1}b$ is straightforward but still requires $\mathcal{O}(n^2)$ operations. The advantage of the Schur algorithm over the Levinson algorithm is that it lends itself very well to parallel computation, resulting in linear parallel complexity. The Schur algorithm uses three sub-matrices; T_k as defined above, as well as the the two $(n-k+1)\times k$ Toeplitz matrices

$$T_k^- = \begin{bmatrix} v_{k-n} & \dots & v_{1-n} \\ \vdots & & \vdots \\ v_0 & \dots & v_{1-k} \end{bmatrix} \text{ and } T_k^+ = \begin{bmatrix} v_{k-1} & \dots & v_0 \\ \vdots & & \vdots \\ v_{n-1} & \dots & v_{n-k} \end{bmatrix}.$$

We define $x_k^{\pm}$ as before, so that

$$\begin{bmatrix} T_k^- \\ T_k^+ \end{bmatrix} \begin{bmatrix} x_k^- & x_k^+ \end{bmatrix} = \begin{bmatrix} s_k^{--} & s_k^{-+} \\ s_k^{+-} & s_k^{++} \end{bmatrix}$$

where $s_{i,k}^{+\pm} = \begin{bmatrix} v_{k+i-2} & \dots & a_{i-1} \end{bmatrix} x_k^{\pm}$ and $s_{i,k}^{-\pm} = \begin{bmatrix} v_{k+i-1-n} & \dots & a_{i-n} \end{bmatrix} x_k^{\pm}$, and these are called the residual vectors. Since x_k^+ is a vector containing k elements, we can augment each vector $x_k^{+\prime} = \begin{bmatrix} x_k^+ \\ 0 \end{bmatrix}$ to have length n, and then construct the upper diagonal matrix $V = \begin{bmatrix} x_1^{+\prime} & x_2^{+\prime} & \dots & x_n^+ \end{bmatrix}$. Given this V, $L = T_n V$ is a lower triangular matrix. If $s_k^{++\prime} = \begin{bmatrix} 0 \\ s_k^{++} \end{bmatrix}$ is the augmented residual vector of length n for each k, then $L = \begin{bmatrix} s_1^{++\prime} & s_2^{++\prime} & \dots & s_n^{++} \end{bmatrix}$, and $T_n = LV^{-1}$ is a triangular factorization of T_n. L and V are calculated recursively. The Schur algorithm can be used to speed the Levinson algorithm up, but ultimately both algorithms require $\mathcal{O}(n^2)$ operations.

Böttcher and Grudsky [3] give a precise definition of $T^{-1}b$, if T is a banded Toeplitz matrix and b is a Laurent polynomial, so that

$$b(t) = \sum_{j=-r}^{s} b_j t^j$$

where $t \in \{\mathbb{C} : |t| = 1\}$. Expanding out the precise definition requires $\mathcal{O}(rn)$ operations where r is the number of non-zero diagonals.

Codevico, Heinig, and van Barel [6] introduce a super-fast $\mathcal{O}(n \log^2 n)$ direct solver, which uses real trigonometric transformations, rather than the Fast Fourier Transform. Their solver works for symmetric Toeplitz matrices. Since a symmetric Toeplitz matrix is also centrosymmetric, $T_n = J_n T_n J_n$ where J_n is a matrix with ones on the anti-diagonal and zeroes elsewhere, and the problem $T_n x = b$ can be reduced to two symmetric systems, each of which can be interpreted in terms of Chebyshev polynomials, which means they can be calculated using cosine transforms. However, due to some instability problems, they needed to include an iterative refinement which slowed the algorithm down, in part because deciding when to perform the refinement was non-trivial.

Chandrasekaran, Gu, Sun, Xia, and Zhu [5] utilize the FFT to transform the Toeplitz matrix into a Cauchy-like matrix, and then exploit the low rank property of Cauchy-like matrices; to whit that each off-diagonal block has low numerical rank. The Cauchy-like matrix is then approximated using a low-rank matrix called the sequentially semi-separable (SSS) matrix. If the matrix is in compact SSS form, it can be solved using $\mathcal{O}(p^2 N)$ operations, where p is the complexity of the semi-separable matrices. The challenge thus is computing the compact SSS matrices. Their algorithm requires $\mathcal{O}(n \log n + p^2 n)$ operations, and does not require the Toeplitz matrix to be either symmetric or positive definite.

Indirect Solvers

Indirect solvers work somewhat differently; they rely on a choice of suitable preconditioner and the use of the Preconditioned Conjugate Gradient method, and are often used for Toeplitz matrices generated by some function $f(x)$ over a domain $[a,b]$, so that the vector $v = [f(x_{n-1} \dots f(x_0) \dots f(x_{1-n})]$ determines the Toeplitz matrix T_n; the function is typically determined by the problem domain. Thus the efficacy of the method depends on the choice of preconditioner; Wen, Ching, and Ng [30] propose approximate inverse-free preconditioners, whose construction is based on the Gohberg-Semencul formula. If the Toeplitz matrix is generated by a positive bounded function and the matrix has the off-diagonal decay property, then these preconditioners work well. In contrast, Ng and Pan [23] consider the solution of $(T+D)x = b$ where T is a Toeplitz matrix and D is diagonal and positive. Their requirement is that the entries for T decay exponentially from the main diagonals. Chan and Yeung [5] discuss circulant preconditioners, where C_n is the circulant matrix that minimizes $\|B_n - T_n\|$ over all circulant matrices B_n. They consider Toeplitz matrices whose generating function is complex-valued rather than real.

Ng, Sun, and Jin [24] combine the recursive approach of the direct solvers and the indirect approach of the Preconditioned Conjugate Method. Preconditioners for T_m are calculated using $T_{m/2}^{-1}$ and the Gohberg-Semencul formula. Unlike the previous papers they do not rely on knowing the generating function f of T_n explicitly; as long as f is a non-negative, bounded, piecewise continuous even function with a finite number of zeros of even order, the conjugate gradient method is guaranteed to converge in $\mathcal{O}(\log n)$ steps, leading to an overall complexity of $\mathcal{O}(n\log^2 n)$.

Diagonally Dominant Matrices

Pan [25] specifically addresses diagonally dominant Toeplitz matrices, and proves that if the diagonally dominant matrix is both Toeplitz, Hermitian, and well-conditioned, so that $\log(\text{cond}(A)) = \mathcal{O}(\log n)$, there exists a numerically stable parallel fast algorithm to invert the matrix. The complexity, or operations count, of parallel algorithms is considered both in terms of the number of arithmetic operations and the number of processors required, and Pan's algorithm requires $\mathcal{O}(\log^2 n \log\log n)$ parallel arithmetic operations, and $n\log^2 n/\log\log n$ processors. Parallel algorithms can be modified to become sequential algorithms, most easily by using a round robin scheme in which each of the parallel operations is performed in turn. To calculate the sequential complexity, we consider both the number of arithmetic steps and the number of processors, and we can bound the sequential complexity from above by multiplying the two; we note that this is strictly an upper bound since altering the algorithm to become sequential removes the need to worry about load balancing or message passing between the processors that are executing in parallel. Thus Pan's algorithm requires $\mathcal{O}((\log^2 n \log\log n)(n\log^2 n/\log\log n)) = \mathcal{O}(n\log^4 n)$ arithmetic operations.

3.4 Summary

In summary, known solvers take advantage of the highly structured form of a Toeplitz matrix to solve the problem recursively using $\mathcal{O}(n^2)$ operations. These methods were improved upon by Codevico, Heinig and van Barel to result in a "super-fast" $\mathcal{O}(n\log^2 n)$ solver using real trigonometric transformations rather than the FFT, but only for symmetric Toeplitz matrices. Chandrasekharan et al use the FFT to transform the Toeplitz matrix, resulting in an $\mathcal{O}(n\log n + p^2 n)$ algorithm, where p is the complexity of the semi-separable matrices. Methods using the Preconditioned Conjugate Gradient Method require at least $\mathcal{O}(n\log^2 n)$ operations, and in some cases calculating the preconditioner requires more work than that. Lastly, Pan shows that he can explicitly solve diagonally dominant Hermitian matrices quickly, using parallel processors.

In Chapters 4 and 5 we will develop algorithms that approximately solve $Ax = b$ for x, where A is a Toeplitz matrix, using $\mathcal{O}(n \log n)$ operations. Specifically, we will consider diagonally and more generally dominant matrices, as well as the matrices generated by stochastic interpolation. We will also present an upper bound on the error.

Chapter 4

The Schulz-Jones-Mayer Algorithm

4.1 The SJM Approach

In this chapter we will discuss the Shulz-Jones-Mayer algorithm, including estimating the algorithm's error bounds when using fixed precision computation, and the run-time complexity. We also discuss the choice of good initial starting states for a variety of matrix classes. This chapter is primarily theoretical; the algorithm relies heavily on matrix-matrix multiplications, which are slow and cumbersome. In Chapter 5 we will modify the algorithm so as to replace the matrix-matrix multiplications with a summation of matrix-vector multiplications.

The Shulz-Jones-Mayer (or SJM) algorithm mentioned obliquely by Crandall [8, 9] relies on a suitable guess to iteratively calculate the inverse of a matrix A. Motivated by the Newton iteration method, SJM requires an initial matrix X_0 and calculates successive approximations of A^{-1} using the rule

$$X_{i+1} = 2X_i - X_i A X_i. \tag{4.1}$$

If the initial guess X_0 is well-chosen, then the successive X_i will satisfy the condition that

$$\lim_{k\to\infty} X_k = A^{-1}. \tag{4.2}$$

The following theorem shows that there is a sufficient condition on X_0, although there are cases where the limit converges without satisfying the theorem.

Theorem 4.1.1. *Let A be an invertible matrix. Then the Shulz-Jones-Mayer method converges quadratically when $\|I - X_0 A\| < 1$ for some initial guess X_0.*

Proof. Note that $\|I - X_k A\| = \left\|(A^{-1} - X_k)A\right\| \leq \left\|A^{-1} - X_k\right\| \|A\|$, which can be rewritten as

$$\frac{\|I - X_k A\|}{\|A\|} \leq \left\|A^{-1} - X_k\right\|. \tag{4.3}$$

Since $\|A\|$ is fixed, it suffices to consider the relative error, e_k, of the k-th iteration. Then

$$\begin{aligned} e_k &= \|I - X_k A\| \\ &= \|I - (2X_{k-1} - X_{k-1}AX_{k-1})A\| = \|I - 2X_{k-1}A + (X_{k-1}A)^2\| \\ &= \|(I - X_{k-1}A)(I - X_{k-1}A)\| \\ &\leq \|I - X_{k-1}A\|^2 = e_{k-1}^2, \end{aligned} \tag{4.4}$$

and we have convergence if $\lim_{k\to\infty} e_k/e_{k-1}^2 \to 0$. When the sequence converges,

$$e_k \leq e_{k-1}^2 \leq (e_{k-2}^2)^2 \leq ((e_{k-3}^2)^2)^2 \leq \cdots \leq e_0^{2^k} \tag{4.5}$$

and $0 \leq e_k \leq e_0^{2^k}$. Convergence is assured, and quadratic, if $\|I - X_0 A\| = e_0 < 1$, since then $\|I - X_k A\| \leq e_0^{2k}$, with $e_0^{2k} \to 0$ as $k \to \infty$. □

Theorem 4.1.1 assumes that the precision used in calculations is exact; for numerical implementations it is important that $\|I - X_0 A\| < \delta \ll 1$, for some $\delta > 0$, so that numerical errors will not overwhelm the computation. Specifically, when e_0 is close to 1, convergence typically requires many iterations; in a numerical implementation the convergence may stall due to round off errors.

4.1.1 Error Analysis

Consider the accuracy of SJM in a numerical implementation; we can bound the magnitude of the computational errors, assuming that X_0 satisfies Theorem 4.1.1. Using the notation and error analysis of matrix and vector operations by Mikhlin[22], we define $\delta(X)$ as the computational error, given a calculated matrix X, so that $\hat{X} = X + \delta(X)$ is the computed result and $\|\delta(X)\|$ is the magnitude of the computational error. The machine precision is represented by ε. The SJM algorithm is a recursive algorithm, so each step relies on the previous calculation rather than the initial state. Thus, using Mikhlin's bounds on the error of matrix addition and multiplication, each iteration the error is compounded by 2 matrix-matrix multiplications, a matrix scaling, and a matrix addition, resulting in an error estimate

$$\|\delta(X_{k+1})\| \leq \varepsilon\sqrt{n}\left(2\left\|\hat{X}_k\right\| + \left\|\hat{X}_k\right\|^2 \|A\|\right). \tag{4.6}$$

Calculating this error estimate for the $(k+1)$-th iteration relies on actually computing the first k iterations, which is far from ideal. Ideally we can bound the error before calculating the iterations, relying only on the initial states, so that we can use the error bound as a predictive value to fine-tune expectations. We will show in Chapter 5 that it is possible to

adapt SJM, in the process achieving a more useful computational error estimate. When using the SJM algorithm for stochastic interpolation, we are specifically interested in solving the related problem $Ax = b$ for x; if we use the method explicitly we calculate $x = A^{-1}b$ which is approximated as $x_k \approx X_k b$, with an error estimate of

$$\begin{aligned} \|\delta(x_k)\| &\leq \varepsilon\sqrt{n}\left(2\left\|\hat{X}_k\right\| + \left\|\hat{X}_k\right\|^2 \|A\|\right)\|b\| \\ &= \|\delta(X_{k+1})\|\,\|b\|\,. \end{aligned} \tag{4.7}$$

4.1.2 Complexity

The basic Shulz-Jones-Mayer algorithm, as stated, is not generally a speedy algorithm. In certain cases we can take advantage of the structure of the initial matrix A and initial guess X_0 to improve the number of operations required. The analysis below considers the general case only, although the proof can be modified for specific types of matrices. Since most classes of matrices are not closed under multiplication, the special cases arise only rarely. Given the quadratic convergence of the algorithm, note that the number of recursive steps k is usually a relatively small number, and as n grows, becomes simply a constant. We present a faster implementation in Chapter 5, and include this analysis for comparison's sake.

Lemma 4.1.2. *Given any $n \times n$ invertible matrix A and initial guess X_0 which satisfy Theorem 4.1.1, SJM will require $\mathcal{O}(kn^3)$ operations to complete k iterations.*

Proof. Each iteration of (4.1) requires two matrix-matrix multiplications, 1 matrix scaling, and 1 matrix subtraction. In general, a matrix-matrix operation requires $\mathcal{O}(n^3)$ operations[13]. Matrix scaling and subtraction can both be completed using n^2 operations each, so that the matrix-matrix multiplications are the limiting factor. Obviously, given k iterations, SJM will require $\mathcal{O}(kn^3)$ operations. □

Generally the number of iterations is often much smaller than the size of the matrix A, so that the factor k is negligible. However, if we can show that, for instance, A and every X_i are Toeplitz matrices, then we can take advantage of the matrix structure to speed the matrix-matrix multiplication up and require only $\mathcal{O}(n^2)$ operations[13] since each matrix can be represented by a $2n-1$ element vector. However, Toeplitz matrices are not closed under multiplication, and hence there is no guarantee that successive X_i will be Toeplitz matrices. Consider, for example, the following two 3×3 Toeplitz matrices, and their product:

$$A = \begin{bmatrix} 3 & 2 & 1 \\ 4 & 3 & 2 \\ 5 & 4 & 3 \end{bmatrix}, \quad B = \begin{bmatrix} 3 & 4 & 5 \\ 2 & 3 & 4 \\ 1 & 2 & 3, \end{bmatrix}, \quad AB = \begin{bmatrix} 14 & 20 & 26 \\ 20 & 29 & 38 \\ 26 & 38 & 50 \end{bmatrix},$$

and it is clear to see that the set of Toeplitz matrices is not closed under multiplication. We demonstrate in Chapter 5 that with some care we can nonetheless take advantage of the Toeplitz structure to solve $Ax = b$ for x using fewer operations.

4.2 Initial Guesses for a Variety of Classes of Matrices

Clearly, the effectiveness of the Shulz-Jones-Mayer algorithm depends on the initial guess X_0; if the initial guess is insufficiently close to the actual inverse of A so that Theorem 4.1.1 is not satisfied, then there is no guarantee that SJM will converge. We consider several different classes, including matrices exhibiting diagonal dominance, extended diagonal dominance, tridiagonal dominance, and lastly the matrices generated when using stochastic interpolation as described in Chapter 2 and present initial guesses X_0 for each of these classes of matrices.

4.2.1 Diagonally Dominant Matrices

Consider first matrices exhibiting diagonal dominance; a matrix A is diagonally dominant if for every row i, $i = 1, \ldots, n$, the following inequality holds:

$$|a_{i,i}| \leq \sum_{j=1, j\neq i}^{n} |a_{i,j}|. \tag{4.8}$$

We can scale any diagonally dominant matrix so that the values along the diagonal of the scaled matrix all lie in $[0,1]$. This scaling can be done with a simple diagonal matrix, and that scaling matrix X_0 will satisfy the condition $\|I - X_0 A\| < 1$, as we show below. We first used D^{-1}, where D is the diagonal submatrix of A, but found that on average results improve drastically if we instead use the maximum diagonal value as a scaling factor. Note that $a_{i,i} \neq 0$ for all $i = 1, 2, \ldots, n$, since A is diagonally dominant, and hence if $a_{i,i} = 0$, the entire row would contain only zeroes, resulting in a non-invertible matrix.

Theorem 4.2.1. *Given any $n \times n$ invertible diagonally dominant matrix A, define $m = \max_{i=1..n} |a_{i,i}|$, and the diagonal matrix $X_0 = [x_{i,j}]$ such that $x_{i,i} = sgn(a_{i,i})/m$ and $x_{i,j} = 0$ for $i \neq j$.*

Then $\|I - X_0 A\|_\infty < 1$ and SJM converges.

Proof. Since $x_{i,i} = \operatorname{sgn}(a_{i,i})/m$, it follows that X_0 scales each column of A by $\pm 1/m$, and changes the signs of the diagonal values so that all are positive. The resulting matrix will remain diagonally dominant, with diagonal values $(XA)_{i,i} \in (0,1]$.

Now define

$$\sigma(A)_i = \sum_{i=j}^{n} |a_{i,j}|, \tag{4.9}$$

which is the sum of the i-th row of A. Then $\sigma(X_0A)_i = \sigma(A)_i/m$. Given that A is diagonally dominant, and $0 \leq (X_0A)_{i,i} \leq 1$, it follows that $\sigma(X_0A)_i < 2(X_0A)_{i,i}$. Hence

$$\sigma(I - X_0A)_i = (1 - X_0A)_{i,i} + \sum_{j \neq i} |(X_0A)_{i,j}| < 1 \tag{4.10}$$

and thus

$$\|I - X_0A\|_\infty = \max_i \sigma(I - X_0A)_i < 1, \tag{4.11}$$

and by Theorem 4.1.1, SJM will converge given A with this initial guess X_0. □

4.2.2 Extended Diagonal Dominance

We next extend the result in Theorem 4.2.1 to extended diagonal dominance, where a matrix A has extended diagonal dominance if cyclically shifting all the columns would result in a diagonally dominant matrix. Consider the following matrix which is clearly not diagonally dominant,

$$A = \begin{bmatrix} 1 & 1 & 1 & 5 & 1 \\ 1 & 1 & 1 & 1 & 5 \\ 5 & 1 & 1 & 1 & 1 \\ 1 & 5 & 1 & 1 & 1 \\ 1 & 1 & 5 & 1 & 1 \end{bmatrix} \tag{4.12}$$

but has an extended diagonal that begins in column 4 on row 1, and wraps around after two rows. Then when A is multiplied by the shifting matrix

$$S = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}, \tag{4.13}$$

the resulting matrix SA is diagonally dominant. We can generalize this further, as extended dominance, by allowing the shifted columns to be in any order, as long as there is only one row-dominant value in every column. This is a variation of the Rook's problem in chess, where the rooks are placed on a square board so that no two rooks share a row or column, but we replace the rooks with the dominant values in every row. Thus for a given matrix

A we can construct a permutation matrix αS that will reorder, and scale, the columns to result in a diagonally dominant matrix with values on the diagonal all in $[0,1]$. Since the permutation matrix S is orthogonal, as the rows and columns are orthogonal unit vectors, it follows that $S^T = S^{-1}$, and $(\alpha S)^{-1} = (1/\alpha)S^T$. Thus once A has been modified we can solve $\alpha SAx = b$ as $x = (\alpha SA)^{-1}b = A^{-1}(1/\alpha)S^T b$.

Consider the example

$$A = \begin{bmatrix} 1 & 5 & 1 & 1 & 1 \\ 1 & 1 & 1 & 5 & 1 \\ 5 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 5 \\ 1 & 1 & 5 & 1 & 1 \end{bmatrix}, \tag{4.14}$$

where the dominant values are stored in positions $\{(1,2),(2,4),(3,1),(4,5),(5,3)\}$. Let A' be the modified matrix containing only the dominant values, with zeroes for all other positions. Then the permutation matrix $S = (1/m)(A')^T$, where $m = \max_{i,j}|a_{i,j}|$. So for this example

$$S = \begin{bmatrix} 0 & 0 & 0.2 & 0 & 0 \\ 0.2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.2 \\ 0 & 0.2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.2 & 0 \end{bmatrix}, \tag{4.15}$$

and

$$SA = \begin{bmatrix} 0 & 0 & 0.2 & 0 & 0 \\ 0.2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.2 \\ 0 & 0.2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.2 & 0 \end{bmatrix} \begin{bmatrix} 1 & 5 & 1 & 1 & 1 \\ 1 & 1 & 1 & 5 & 1 \\ 5 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 5 \\ 1 & 1 & 5 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0.2 & 0.2 & 0.2 & 0.2 \\ 0.2 & 1 & 0.2 & 0.2 & 0.2 \\ 0.2 & 0.2 & 1 & 0.2 & 0.2 \\ 0.2 & 0.2 & 0.2 & 1 & 0.2 \\ 0.2 & 0.2 & 0.2 & 0.2 & 1 \end{bmatrix} \tag{4.16}$$

Theorem 4.2.2. *Given any invertible $n \times n$ matrix A, with extended dominance, let A' be the modified matrix containing only the dominant values of A, let $m = \max_{i,j}|a_{i,j}|$, and let $S = (1/m)(A')^T$. Then SJM will converge if $X_0 = S$.*

Proof. For each row i in A, let (i,j) be the position of the maximum value $\max_j |a_{i,j}|$. Then set $(A')_{i,j} = -1(\text{sgn}(a_{i,j}))$, and all other values zero. Then A' will contain exactly one non-zero value in each row and column, since A has extended dominance. Let $m = \max_{i,j}|a_{i,j}|$, and set $S = (1/m)(A')^T$. Thus S will simultaneously scale A and reorder the columns of A, resulting in a diagonally dominant matrix, with all values on the diagonal lying in $[0,1]$.

Hence, by the same argument as used in Theorem 4.2.1, it follows that $\|I - SA\| < 1$ and SJM will converge using $X_0 = S$ as the initial guess. □

4.2.3 Tri-Diagonally Dominant Matrices

Further expanding the concept of dominance leads us to consider tridiagonal dominance. Whereas a matrix A is diagonally dominant if the absolute value of the diagonal is greater than the sum of the absolute values of the non-diagonal entries on each row, the matrix A is tri-diagonally dominant if the sum in each row of the absolute values of the diagonal and super- and sub-diagonal values is greater than the sum of the remaining entries. In short, A is tri-diagonally dominant if

$$\sum_{k=i-1,k>0}^{i+1,k\leq n} |a_{ik}| > \sum_{k=1}^{i-2} |a_{ik}| + \sum_{k=i+2}^{n} |a_{ik}| \tag{4.17}$$

holds for row i of matrix A, $i = 1 \ldots n$. We can decompose the matrix $A = D_3 + M$, where D_3 consists of the three diagonals, including the sub- and super diagonal. Then, assuming D_3 is non-singular, the inverse D_3^{-1} is known and can be calculated using the elegant and concise formulation described by Usmani [28].

Consider the tridiagonal matrix

$$D_3 = \begin{pmatrix} a_1 & b_1 & & & \\ c_1 & a_2 & b_2 & & \\ & c_2 & \ddots & \ddots & \\ & & \ddots & \ddots & b_{n-1} \\ & & & c_{n-1} & a_n \end{pmatrix}. \tag{4.18}$$

Then D_3^{-1} is defined as

$$(D_3^{-1})_{ij} = \begin{cases} (-1)^{i+j} b_i \ldots b_{j-1} \theta_{i-1} \phi_{j+1} / \theta_n, & \text{if } i \leq j \\ (-1)^{i+j} c_j \ldots c_{i-1} \theta_{j-1} \phi_{i+1} / \theta_n, & \text{if } i > j \end{cases} \tag{4.19}$$

where

$$\begin{aligned} \theta_0 &= 1 \\ \theta_1 &= a_1 \\ \theta_i &= a_i \theta_{i-1} - b_{i-1} c_{i-1} \theta_{i-2}, \text{ for } i = 2, \ldots, n \end{aligned}$$

and

$$\begin{aligned} \phi_{n+1} &= 1 \\ \phi_n &= a_n \\ \phi_i &= a_i \phi_{i+1} - b_i c_i \phi_{i+2}, \text{ for } i = n-1, \ldots, 1. \end{aligned}$$

We can calculate D_3^{-1} explicitly using Algorithm 1; by pre-calculating the values we can calculate the inverse D_3^{-1} using only $\mathcal{O}(n^2)$ operations.

Algorithm 1 Calculate D_3^{-1}

Require: The $n \times n$ matrix D_3 is invertible

1: $\theta_0 = 1$, $\theta_1 = a_1$
2: **for** $i = 2$ to n **do**
3: $\quad \theta_i = a_i\theta_{i-1} - b_{i-1}c_{i-1}\theta_{i-2}$
4: **end for**
5: $\phi_{n+1} = 1$, $\phi_n = a_n$
6: **for** $i = n-1$ down to 1 **do**
7: $\quad \phi_i = a_i\phi_{i+1} - b_ic_i\phi_{i+2}$
8: **end for**
9: ▷ Now we construct the matrix B such that
10: ▷ $B_{i,j} = b_ib_{i+1}\cdots b_j$, for $i \leq j$, and C likewise
11: $B_{1,1} = b_1$, $C_{1,1} = c_1$
12: **for** $i = 2$ to $n-1$ **do**
13: $\quad B_{1,i} = b_iB_{1,i-1}$, $C_{1,i} = c_iC_{1,i-1}$
14: **end for**
15: **for** $i = 2$ to $n-1$ **do**
16: $\quad B_{i,i} = b_i$, $C_{i,i} = c_i$
17: $\quad$ **for** $j = i+1$ to $n-1$ **do**
18: $\quad\quad B_{i,j} = b_jB_{i,j-1}$, $C_{i,j} = c_jC_{i,j-1}$
19: $\quad$ **end for**
20: **end for**
21: ▷ Final Step: Calculating D_3^{-1}
22: **for** $i = 1$ to n **do**
23: $\quad$ **for** $j = 1$ to n **do**
24: $\quad\quad$ **if** $i \leq j$ **then**
25: $\quad\quad\quad D^{-1}_{3_{i,j}} = (-1)^{i+j}B_{i,j-1}\theta_{i-1}\phi_{j+1}/\theta_n$
26: $\quad\quad$ **else**
27: $\quad\quad\quad D^{-1}_{3_{i,j}} = (-1)^{i+j}C_{j,i-1}\theta_{j-1}\phi_{i+1}/\theta_n$
28: $\quad\quad$ **end if**
29: $\quad$ **end for**
30: **end for**
31: **return** D_3^{-1}

Lemma 4.2.3. *Let D_3 be a tri-diagonal matrix defined by the three vectors $a = \begin{bmatrix} a_1 & a_2 & \dots a_n \end{bmatrix}$, $b = \begin{bmatrix} b_1 & b_2 & \dots b_{n-1} \end{bmatrix}$, and $c = \begin{bmatrix} c_1 & c_2 & \dots c_{n-1} \end{bmatrix}$, as described in (4.18). Then the inverse D_3^{-1} can be calculated using $\mathcal{O}(n^2)$ operations.*

Proof. Consider Algorithm 1, which calculates D_3^{-1}. Lines 3 and 7 both require 4 operations, and are called $n-1$ times, resulting in $8(n-1) = 8n-8$ operations. The algorithm uses two additional matrices, B and C to store the results of multiplying subsets of $\{b_i\}$ and $\{c_i\}$ together; in (4.19) the products of multiple consecutive values are used. Hence

$B_{i,j}$ is initialized so that $B_{i,j} = b_i b_{i+1} \dots b_j$, where $i \leq j$, and C is set up similarly. By building these two matrices row by row, we can rely on previous calculations, so that the total number of operations required is $2(n-2)$ for the loop in lines 11 to 13, and $\sum_{i=2}^{n-1} \sum_{j=i+1}^{n-1} 2 = \sum_{i=2}^{n-1} 2n - 2i - 2 = n^2 - 5n + 6$ for the nested for loops in lines 14 to 19.

At this point all the pre-calculations have been completed, and D^{-1} can be calculated directly. Note that $(-1)^{i+j}$ does not have to be calculated directly, but can be replaced with an if statement, so that $(-1)^{i+j} = -1$ if $i+j$ is odd, and 1 otherwise. Hence the sign can be calculated with a single operation, and the total number of operations required in lines 24 and 26 is 4, and on each iteration of the for loops either line 24 or line 26 will be used, but not both. Thus the total number of operations in lines 21 to 29 is

$$\sum_{i=1}^{n} \sum_{j=1}^{n} 4 = 4n^2. \tag{4.20}$$

Combining all these results, the total number of operations required is $\mathcal{O}(n^2)$ since $(8n-8) + 2(n-2) + (n^2 - 5n + 6) + (4n^2) = 5n^2 + 5n - 4$.

□

Naturally, calculating D_3 using fixed, finite precision will result in some computational error, which depends in part on the size of the matrix. To illustrate the effect of the matrix size on the computational error, at various levels of precision, we generated random tri-diagonal matrices of various sizes such that $a_i, b_i, c_i \in [0,1]$ and then measured the distance between the identity matrix and the calculated inverse matrix multiplied by the original matrix. In short, we calculated $\left\| I - D_3 D_3^{-1} \right\|$. Using perfect precision, $\left\| I - D_3 D_3^{-1} \right\| = 0$. By implementing Algorithm 1 in Maple, we were able to calculate the matrices using both perfect precision and controlled single, double, and quadruple precision. Fig. 4.1 illustrates the accuracy achievable; for small matrices the accuracy of the inverse primarily depends on the precision used. However, as matrices grow larger, we can see the effect of multiplying many floating point numbers together, so that when $n = 1000$ there is a significant degrading in accuracy. Hence when implementing SJM, in Theorem 4.2.4, where the initial guess is $X = D_3^{-1}$, it is important to distinguish between the inverse calculated with perfect precision, and the inverse calculated with computer precision; it is the latter we need in the condition that $\|XM\| < 1 - \left\| I - D_3 D_3^{-1} \right\|$.

Theorem 4.2.4. *Given a tridiagonally dominant matrix $A = D_3 + M$, where D_3 is tridiagonal and invertible, SJM will converge if $\|X_0 M\| < 1$, with initial guess $X_0 = D_3^{-1}$.*

Proof. Let A be a tridiagonally dominant matrix, such that $A = D_3 + M$, and D_3 is both tri-diagonal and invertible. Then by Theorem 4.1.1, SJM will converge if $\|I - X_0 A\| < 1$.

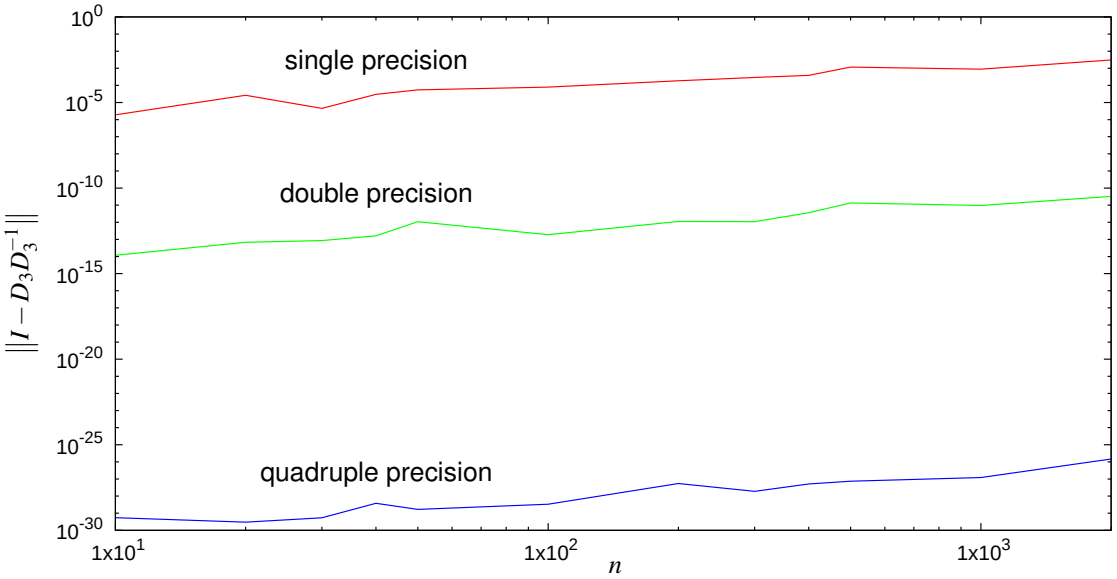


Figure 4.1: Computation error $\left\|I - D_3 D_3^{-1}\right\|$ for random tridiagonal matrices, with $a_i, b_i, c_i \in [0,1]$ with size n varying from 10 to 2000, implemented using single, double and quadruple precision in Maple.

Consider

$$\begin{aligned} \|I - X_0 A\| &= \|I - X_0(D_3 + M)\| \\ &= \|I - X_0 D_3 - X_0 M\| \\ &= \left\|I - D_3^{-1} D_3 - X_0 M\right\| \\ &= \|X_0 M\|. \end{aligned}$$

Hence $\|I - X_0 A\| < 1$ if and only if $\|X_0 M\| < 1$. □

We can further generalize this result, since the proof relies only on being able to calculate the inverse of D_3; in general, if we can decompose $A = B + C$ into two matrices, and B has a known inverse B^{-1}, then B^{-1} will make a sufficient guess if $\left\|B^{-1}C\right\| < 1$.

Corollary 4.2.5. *Let A be an invertible matrix such that $A = B + C$ and B^{-1} is known. Then SJM will converge, with B^{-1} as initial guess X_0, if $\left\|B^{-1}C\right\| < 1$.*

Proof. Let A be an invertible matrix, such that $A = B + C$ and B^{-1} is known, and et $X_0 = B^{-1}$.

Then by Theorem 4.1.1, SJM will converge if $\|I - X_0 A\| < 1$. Consider

$$\begin{aligned}\|I - X_0 A\| &= \|I - X_0(B+C)\| \\ &= \|I - X_0 B - X_0 C\| \\ &= \left\|I - B^{-1}B - X_0 C\right\| \\ &= \|X_0 C\|.\end{aligned}$$

Hence $\|I - X_0 A\| < 1$ if and only if $\|X_0 M\| < 1$. □

4.2.4 Gaussian Matrices

Consider the matrix A as defined in (2.3), in Sec. 2.3, using a Gaussian distribution so that

$$a_{jk} = \frac{1}{2}\left[\operatorname{erf}\left(\frac{y_{k+1} - x_j}{2\sqrt{\alpha}/n}\right) - \operatorname{erf}\left(\frac{y_k - x_j}{2\sqrt{\alpha}/n}\right)\right]. \tag{4.21}$$

The resultant matrix is row stochastic. Assuming that the initial data points $\{x_k, f_k\}_{k=1\ldots n}$ were evenly spaced, the matrix will be near Toeplitz, with only the first and last row differing.

Consider the matrix X_0 defined by

$$x_{jk} = \frac{1}{2}\left[1/\operatorname{erf}\left(\frac{y_{k+1} - x_j}{2\sqrt{\alpha}/n}\right) - 1/\operatorname{erf}\left(\frac{y_k - x_j}{2\sqrt{\alpha}/n}\right)\right]. \tag{4.22}$$

Then the product $X_0 A$ will depend on the size and spacing of the original inputs x_k, as well as the mollifier α, and does not depend on the $\{f_k\}$, so that the same matrix can be used for many problems. If the points are evenly spaced, then $X_0 A$ can easily be calculated explicitly, and we can show that X_0 is a valid initial guess for carefully chosen α.

Theorem 4.2.6. *Given a matrix A constructed using (4.21) where the data points consist of n even spaced points, and initial guess X_0 constructed using (4.22), using a sufficiently small α so that $(X_0 A)_{i,i} < 1$, SJM will converge.*

Proof. First, from (2.1) and the fact that the x_j are evenly spaced between 0 and 1, so that $x_j = j/n$, we have

$$y_k - x_j = \begin{cases} -\infty & \text{if } k = 0 \\ \dfrac{2k+1}{2n} - \dfrac{2j}{2n} & \text{if } 0 < k < n \\ \infty & \text{if } k = n \end{cases} \tag{4.23}$$

and hence

$$\begin{aligned}(X_0A)_{i,j} &= \sum_{k=1}^{n}(X_0)_{i,k}A_{k,j} \\ &= \sum_{k=1}^{n}\frac{1}{4}\left[1/\text{erf}\left(\frac{y_{k+1}-x_i}{2\sqrt{\alpha}/n}\right)-1/\text{erf}\left(\frac{y_k-x_i}{2\sqrt{\alpha}/n}\right)\right]\left[\text{erf}\left(\frac{y_{j+1}-x_k}{2\sqrt{\alpha}/n}\right)-\text{erf}\left(\frac{y_j-x_k}{2\sqrt{\alpha}/n}\right)\right] \\ &= \frac{1}{4}\sum_{k=1}^{n}\left[1/\text{erf}\left(\frac{2k+3-2i}{4\sqrt{\alpha}}\right)-1/\text{erf}\left(\frac{2k+1-2i}{4\sqrt{\alpha}}\right)\right] \\ &\qquad \left[\text{erf}\left(\frac{2j+3-2k}{4\sqrt{\alpha}}\right)-\text{erf}\left(\frac{2j+1-2k}{4\sqrt{\alpha}}\right)\right] \qquad (4.24)\end{aligned}$$

using (4.23). For the sake of clarity, let

$$f(a,b) = \text{erf}\left(\frac{2a+3-2b}{4\sqrt{\alpha}}\right) \qquad (4.25)$$

using (4.23), and we can rewrite (4.24) as

$$\begin{aligned}(X_0A)_{i,j} &= \frac{1}{4}\sum_{k=1}^{n}\left[\frac{1}{f(k,i)}-\frac{1}{f(k-1,i)}\right][f(j,k)-f(j-1,k)] \\ &= \frac{1}{4}\sum_{k=1}^{n}\left[\frac{f(j,k)-f(j-1,k)}{f(k,i)}-\frac{f(j,k)-f(j-1,k)}{f(k-1,i)}\right]. \qquad (4.26)\end{aligned}$$

Now to show that $\|I - X_0A\| < 1$, it suffices to show that the following conditions hold:

$$(X_0A)_{i,i} > \sum_{j=1,j\neq i}^{n}|(X_0A)_{i,j}|, \qquad (4.27)$$

$$\|(X_0A)\| < 2. \qquad (4.28)$$

Using substitution and (4.24), we see that

$$(X_0A)_{i,i} = \frac{1}{4}\sum_{k=1}^{n}\left[\frac{f(i,k)-f(i-1,k)}{f(k,i)}-\frac{f(i,k)-f(i-1,k)}{f(k-1,i)}\right]. \qquad (4.29)$$

Consider $A_{k,j}$; since $y_{j+1}-x_k > y_j-x_k$, it follows that $\text{erf}\left(\frac{y_{j+1}-x_k}{2\sqrt{\alpha}/n}\right) > \text{erf}\left(\frac{y_j-x_k}{2\sqrt{\alpha}/n}\right)$ and hence $A_{k,j} > 0$. Conversely, $(X_0)_{i,k} < 0$, if $i \neq k$. Hence $|(X_0A)_{i,j}| < 0$ when $i \neq j$. Thus we can consider

$$\begin{aligned}\sum_{j=1}^{n}|(X_0A)_{i,j}| &= \sum_{j=1,j\neq i}^{n}\left|\frac{1}{4}\sum_{k=1}^{n}\left[\frac{f(j,k)-f(j-1,k)}{f(k,i)}-\frac{f(j,k)-f(j-1,k)}{f(k-1,i)}\right]\right| \\ &= -\frac{1}{4}\sum_{j=1,j\neq i}^{n}\sum_{k=1}^{n}\left[\frac{f(j,k)-f(j-1,k)}{f(k,i)}-\frac{f(j,k)-f(j-1,k)}{f(k-1,i)}\right] \\ &= -\frac{1}{4}\sum_{k=1}^{n}\sum_{j=1,j\neq i}^{n}\left[\frac{f(j,k)-f(j-1,k)}{f(k,i)}-\frac{f(j,k)-f(j-1,k)}{f(k-1,i)}\right]\end{aligned}$$

and we can simplify the resulting equation by reordering and canceling terms, resulting in

$$\sum_{j=1}^{n} |(X_0A)_{i,j}| = -\frac{1}{4}\sum_{k=1}^{n}\left[\frac{f(i-1,k)-f(0,k)+f(n,k)-f(i,k)}{f(k,i)} - \frac{f(i-1,k)-f(0,k)+f(n,k)-f(i,k)}{f(k-1,i)}\right]$$

$$= \frac{1}{4}\sum_{k=1}^{n}\left[\frac{(f(i,k)-f(i-1,k))-(f(n,k)-f(0,k))}{f(k,i)} - \frac{(f(i,k)-f(i-1,k))-(f(n,k)-f(0,k))}{f(k-1,i)}\right]$$

$$= (X_0A)_{i,i} - \frac{1}{4}\sum_{k=1}^{n}\left[\frac{(f(n,k)-f(0,k))}{f(k,i)} - \frac{(f(n,k)-f(0,k))}{f(k-1,i)}\right]. \tag{4.30}$$

Then, assuming $\|X_0A\| < 2$, it follows that

$$\begin{aligned}\|I - X_0A\| &= \max_{i=1,\ldots,n} 1-(X_0A)_{i,i} + \sum_{j=1,j\neq i}^{n} |(X_0A)_{i,j}| \\ &< \max_{i=1,\ldots,n} 1-(X_0A)_{i,i} + (X_0A)_{i,i} \\ &< 1. \end{aligned} \tag{4.31}$$

□

The structure of $A = A_{nn}$ is discussed in Sec. 2.4, and A is near-Toeplitz. We can modify A slightly, to make it Toeplitz, by modifying (4.23) so that

$$y_k - x_j = \begin{cases} -\dfrac{1}{2n} & \text{if } k = 0 \\ \dfrac{2k+1-2j}{2n} & \text{if } 0 < k < n \\ \dfrac{2n+1}{2n} & \text{if } k = n \end{cases} \tag{4.32}$$

and the resulting matrices A and X_0, as defined by (4.21) and (4.22) will both be Toeplitz matrices. We can still use the modified A and X_0 for stochastic interpolation, as we will in Chapter 6, but the structure of Toeplitz matrices allows for improvement in the number of operations required to add and multiply matrices, since the entire $n \times n$ matrix is reduced to a vector of length $2n-1$.

4.3 Summary

The SJM algorithm is easy to implement, and to understand, but suffers from computational complexity. The matrix-matrix multiplications require a relatively large number of operations, and estimating error has to be done iteratively. In Chapter 5 we will present solutions to both of those challenges, resulting in a faster, more flexible algorithm, whose errors can be estimated using only the magnitude of the original matrix A and initial guess X_0. We have also presented valid initial guesses for a variety of graph classes, and these initial guesses will guarantee convergence of both SJM, and the modified SJM presented in Chapter 5.

Chapter 5

The Polynomial Schulz-Jones-Mayer Algorithm

5.1 Extending SJM – The PSJM Algorithm

Whilst the Shulz-Jones-Mayer algorithm is known to converge, for certain classes of matrices, the algorithm, requiring $\mathcal{O}(kn^3)$ operations, is no faster than traditional methods such as Gaussian elimination. By reframing the original problem from inversion to solving for x, we are able to adapt the SJM algorithm which results in a much faster algorithm for the solution we actually seek to compute. Our considerations in this chapter are two-fold; to improve the operations count and error estimation for SJM, and to experimentally verify the efficacy of the initial guesses presented for a variety of cases of matrices in Sec. 4.2.

Using SJM, the solution x for the problem $Ax = b$ can be approximated by $x_k = X_k b$, for some sufficiently large k and good initial guess X_0. If the desired value is x, however, it is not necessary to calculate X_k explicitly; consider that

$$\begin{aligned} X_{k+1} &= 2X_k - X_k A X_k \\ &= 2(2X_{k-1} - X_{k-1}AX_{k-1}) - (2X_{k-1} - X_{k-1}AX_{k-1})A(2X_{k-1} - X_{k-1}AX_{k-1}) \\ &= (4I - 6(X_{k-1}A) + 4(X_{k-1}A)^2 - (X_{k-1}A)^3)X_{k-1}. \end{aligned}$$

We can continue continue rewriting X_k in terms of previous x_{k-j} until we reach the initial guess, X_0, allowing us to rewrite X_k as

$$X_k = \left(\sum_{i=0}^{2^k-1} \alpha_{k,i} (X_0 A)^i \right) X_0, \tag{5.1}$$

and

$$x_k = \left(\sum_{i=0}^{2^k-1} \alpha_{k,i} (X_0 A)^i \right) X_0 b, \tag{5.2}$$

where $X_k \approx A^{-1}$ for sufficiently large k when the iteration converges, and the coefficients $\alpha_{k,i}$ are constants that do not rely on the initial matrix. Table 5.1 illustrates the coefficients for the first three iterations. Note that the number of coefficients doubles at each iteration, and can be calculated using dynamic programming, starting with the case where $k = 0$ and

$$\begin{bmatrix} 1 & & & & & & & \\ 2 & -1 & & & & & & \\ 4 & -6 & 4 & -1 & & & & \\ 8 & -28 & 56 & -70 & 56 & -28 & 8 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ (X_0A) \\ (X_0A)^2 \\ (X_0A)^3 \\ (X_0A)^4 \\ (X_0A)^5 \\ (X_0A)^6 \\ (X_0A)^7 \end{bmatrix} = \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \end{bmatrix}$$

Table 5.1: (5.1) written as a matrix multiplication, with coefficients $\alpha_{k,i}$ for the first three iterations.

working up to any level chosen. Notation-wise, $\alpha_{k,i}$ represents the coefficient, in the k-th iteration, of the term with with variable X_0A and exponent i.

Clearly coefficients at each iteration rely on the previous iteration, and we can use a recurrence relation to calculate them.

Lemma 5.1.1. *To calculate* $X_k = \left(\sum_{i=0}^{2^k-1} \alpha_{k,i}(X_0A)^i\right) X_0$*, the* $\alpha_{i,j}$ *are defined as follows:*

$$\alpha_{0,0} = 1, \tag{5.3}$$

$$\alpha_{i,0} = 2\alpha_{i-1,0}, \tag{5.4}$$

$$\alpha_{i,j} = 2\alpha_{i-1,j} - \sum_{k=0}^{j-1} \alpha_{i-1,j-k-1}\,\alpha_{i-1,k}, \ \textit{for } j = 1,\ldots,2^i-1, \tag{5.5}$$

$$\alpha_{i-1,j} = 0 \ \textit{ if } j \geq 2^{i-1} \tag{5.6}$$

Proof. Consider first that $X_0 = (X_0A)^0X_0$, so $\alpha_{0,0} = 1$. Similarly, $X_1 = 2X_0 - X_0AX_0 = (2-(X_0A)^1)X_0$, so that $\alpha_{1,0} = 2$ and $\alpha_{1,1} = -1$. Now assume that the lemma holds for all $j \leq k$, and consider the j-th coefficient of $X_{k+1} = 2X_k - X_kAX_k$. The j-th coefficient of $2X_k = 2\alpha_{k,j}$, but

$$\begin{aligned} X_kAX_k &= \left(\sum_{i=0}^{2^k-1} \alpha_{k,i}(X_0A)^i\right) X_0A \left(\sum_{i=0}^{2^k-1} \alpha_{k,i}(X_0A)^i\right) X_0 \\ &= \left(\sum_{i=0}^{2^k-1} \alpha_{k,i}(X_0A)^{i+1}\right) \left(\sum_{i=0}^{2^k-1} \alpha_{k,i}(X_0A)^i\right) X_0 \end{aligned}$$

so that by combining all the terms with degree j, the j-th coefficient of $X_kAX_k = \sum_{i=0}^{j-1} \alpha_{k,j-i-1}\alpha_{k,j}$. Thus

$$\alpha_{k+1,j} = 2\alpha_{k,j} - \sum_{i=0}^{j-1} \alpha_{k,j-i-1}\alpha_{k,i}, \tag{5.7}$$

which after rewriting with appropriate variables is (5.6). □

5.2 Run-time Complexity

When considering the number of operations required for k iterations of PSJM, we have to consider two calculations; first the calculation of the coefficients $\{\alpha_{k,i}\}$, and secondly the sum (5.1). The coefficients are constant, and do not rely on the size or values of the matrix A and can be calculated once, stored, and thenceforward retrieved as needed. That one-time calculation, to depth k will require $\mathcal{O}(2^{k+2})$ calculations, but will result in calculating all the coefficients for iterations $1,\ldots k$. As such, we will not consider the complexity of calculating $\{\alpha_{k,i}\}$ when considering the number of operations required to calculate x_k.

Lemma 5.2.1. *Calculating the coefficients $\alpha_{i,j}$ for the first k iterations will require $\mathcal{O}(2^{k+2})$ operations.*

Proof. We will first consider the number of operations required to calculate the coefficients for a given iteration i. If $i = 0$ then no calculations are required. Now, assuming $i > 0$ there are two counts we need to consider.

Recall that

$$\alpha_{i,0} = 2\alpha_{i-1,0},$$
$$\alpha_{i,j} = 2\alpha_{i-1,j} - \sum_{k=0}^{j-1} \alpha_{i-1,j-k-1}\,\alpha_{i-1,k}, \text{ for } j = 1,\ldots,2^i-1.$$

Thus the first coefficient requires 1 operation. Each of the other $2^i - 1$ coefficients will require one multiplication and one subtraction, plus the number of operations required to complete the summation. Each term in the summation requires one multiplication and one addition to increase the cumulative total. Thus for coefficient j the number of operations required is $2 + 2j = 2(j+1)$ operations. Then for each iteration the total number of operations can be bounded as

$$1 + \sum_{j=1}^{2^i-1} 2(j+1) \leq 2^{i+1}, \tag{5.8}$$

and the total number of operations can be bounded by

$$\sum_{n=1}^{k} 2^{n+1} \leq 2^{k+2}.$$

Furthermore, storing the coefficients will require $\mathcal{O}(2^{k+1})$ memory locations, although the magnitude of the coefficients grows quite rapidly. □

Clearly, calculating the coefficients $\{\alpha_{k,i}\}$ does not depend on the actual input, but is independent. In contrast, (5.1) depends on the initial $n \times n$ matrix A, initial guess X_0, and vector b, and the number of operations required depends primarily on n, the size of A. Assuming the conditions of Theorem 4.1.1 are satisfied, PSJM calculates, after k iterations, $x_k = X_k b \approx A^{-1}b$ which is an approximation of the correct answer.

Lemma 5.2.2. *Calculating* $x_k = \left(\sum_{i=0}^{2^k-1} \alpha_{k,i}(X_0A)^i\right) X_0 b$ *requires at most* 2^k *matrix vector multiplies and* 2^k *vector additions.*

Proof. We prove this by induction. First consider the base case: if $k = 0$ then $x_0 = X_0 b$, which requires exactly one matrix vector multiply.

Now assume the lemma holds for all $k < n$, and let $b_i = (X_0A)^i X_0 b$. Note that b_i can be calculated using $2i+1$ matrix-vector multiplies, by performing the multiplications from right to left; any other order of operations results in matrix-matrix multiplies, which are operationally more expensive, and thus to be avoided. However, if we calculate the b_i dynamically, storing the intermediate values, then calculating b_{i+1} will require only 2 matrix-vector multiplies more than those required for b_i. Hence the total number of matrix-vector multiplies required to calculate the vectors $b_0, b_1, \ldots, b_i$ is $1+2i$.

Consider the case where $k = n$. At this stage, the vectors b_i such that $i < 2^{n-1}$ have already been calculated, but we need to calculate the additional b_i such that $2^{n-1} \leq i < 2^n$, or 2^{n-1} additional vectors. Each successive b_i can be calculated using two matrix-vector multiplies and the already calculated b_{i-1}, resulting in an additional $2 \times 2^{n-1} = 2^n$ matrix vector multiplies. Hence the total number of matrix-vector multiplies required is $2^k - 1$, or $\mathcal{O}(2^k)$.

Once the various intermediate vectors have been calculated, the sum $x_k = \sum_{i=0}^{2^k-1} \alpha_{k,i} b_i$ requires the addition of 2^k vectors, each multiplied by a non-zero coefficient. □

Actually calculating x_k can be done with less memory than this proof suggests, by maintaining at every step of the summation the current vector b_j, $j = 0 \ldots 2^k - 1$, as well as the current approximation $\sum_{i=0}^{j} \alpha_{k,i} b_i$. This allows us to save on space, but still requires the same number of matrix-vector multiplications and additions.

We can use Lemma 5.2.2 when counting the number of operations, specifically additions and multiplications, required to calculate x_k. The actual operations count will differ based on the structure of both X_0 and A. For instance, if X_0 is a simple scaling matrix consisting of a diagonal matrix with non-zero values on the diagonal, then every matrix-vector multiplication of $X_0 b'$, where b' is an intermediate vector, would only require n operations. Similarly, if A and X_0 are Toeplitz, we can take advantage of a speedier matrix-vector multiplication using the FFT technique.

Lemma 5.2.3. *Given any $n \times n$ invertible matrix, an initial guess X_0 which satisfies Theorem 4.1.1, and a vector b, PSJM will require $\mathcal{O}(2^k n^2)$ operations to calculate x_k.*

Proof. Matrix-vector multiplies, calculated explicitly, require $\mathcal{O}(n^2)$ operations. Calculating vector-vector additions, on the other hand, requires only $\mathcal{O}(n)$ operations. Lemma 5.2.2 shows that calculating x_k will require at most 2^k matrix vector multiplies and 2^k vector additions. Hence, PSJM will require $\mathcal{O}(2^k n^2 + 2^k n) = \mathcal{O}(2^k n^2)$ operations. □

Using a fixed precision platform, the number or iterations used is usually at most 5, so that $2^k = 32$ becomes a constant resulting in a method requiring $\mathcal{O}(n^2)$ operations since $\mathcal{O}(\cdot)$ notation does not include constants. The crucial factor in determining how much work, or how many operations, are required by the algorithm depends strictly on the number of operations required to perform each matrix-vector multiplication.

5.2.1 Matrix Vector Multiplication in $\mathcal{O}(n \log n)$ time

Now consider the case when X_0 and A are both Toeplitz matrices; we can take advantage of the scheme described in Sec. 3.2.1 that embeds the matrix in a circulant matrix, and then uses the FFT to calculate to calculate the matrix-vector multiplication. This produces a matrix-vector multiply that only requires $\mathcal{O}(n \log n)$ operations.

Lemma 5.2.4. *Given any $n \times n$ invertible Toeplitz matrix, an initial guess X_0 which is a Toeplitz matrix and satisfies Theorem 4.1.1, and a vector b, PSJM will require $\mathcal{O}(2^k n \log n)$ operations to calculate x_k.*

Proof. Since A and X_0 are Toeplitz matrices, we can take advantage of the FFT based matrix-vector multiplication method which requires $\mathcal{O}(n \log n)$ operations. From Lemma 5.2.2 we know that calculating x_k will require at most 2^k matrix vector multiplies and vector additions. The additions can be completed using $\mathcal{O}(n)$ operations, and the matrix vector multiplies now only require $\mathcal{O}(n \log n)$ operations, as described in Sec. 3.2.1. Hence PSJM will require $\mathcal{O}(2^k n \log n)$ operations. □

5.3 Error Analysis

To analyze the possible sources of error in PSJM, we must look both at the initial matrices A and X_0 and vector b, and their manipulation, as well as the coefficients $\alpha_{k,i}$. For the convergence of $x_k \to x$, we require perfect computational accuracy which is naturally not possible on a computational platform using only a limited amount of precision. Standard packages such as Matlab and lapack are implemented using double precision arithmetic.

Depth k	Degree n	$\max \lvert\alpha_{k,i}\rvert$
1	$1 = 2^1 - 1$	2
2	$3 = 2^2 - 1$	6
3	$7 = 2^3 - 1$	70
4	$15 = 2^4 - 1$	12870
5	$31 = 2^5 - 1$	601080390
6	$63 = 2^6 - 1$	1832624140942590534

Table 5.2: The degree and maximum coefficient $\max \lvert\alpha_{k,i}\rvert$ of the monic polynomial used to calculate x_k, using PSJM.

A number stored as a double precision floating point number is stored as $(-1)^s \times c \times b^q$, where s is a one-bit sign signifier, c is the significant and q is the exponent. The exponent b is typically 2. Hence the number 12345 would be stored as $(-1)^0 \times 1.2345 \times 10^4$ as a decimal number or $(-1)^0 \times 1.1000000111001 \times 2^{1101}$. Since we have room to store 52 binary digits for the significant, the number 12345 can be represented without rounding but any integer larger than $2^{52} = 4503599627370496$ cannot be represented accurately, since there are insufficient digits available in the significant. Thus, even when working with integers, as the coefficients $\alpha_{k,i}$ are, we must still worry about rounding.

Thus one possible source of error are the coefficients $\alpha_{k,i}$ as k becomes larger; as we can see from Table 5.2, the size of the maximum coefficient grows at such a rapid pace that computation using double precision becomes flawed at the sixth iteration. Consider that double precision can represent $2^{52} = 4503599627370496$ integers accurately, allowing for 52 digits accuracy in binary representation, but only approximately 16 digits in decimal documentation. Thus the number 1832624140942590534, which is the maximum coefficient for the sixth iteration and requires 19 digits in decimal form or 61 digits in binary form, cannot be represented in memory without significant rounding. This source of error is wholly unrelated to the actual values of the initial data, but is the reason why we bound $k \leq 5$.

The other source of error comes from the computational errors that compound as (5.2) is calculated and result from computational errors in the addition, subtraction, and multiplication of matrices. These errors were studied by Mikhlin[22], and yield useful analytical estimates, and a possible way to improve the accuracy of the method, when applied to PSJM. As in Sec. 4.1.1, given a matrix X, let $\delta(X)$ be the computational error, so that $\hat{X} = X + \delta(X)$ is the computed result and $\|\delta(X)\|$ is the size of the computational error. The PSJM algorithm relies only on the initial matrices. Thus the approximate solution is

calculated as $x_k = X_k b$, and it follows that the error is

$$\begin{aligned}\|\delta(x_k)\| &\leq \varepsilon\sqrt{n}\left[\sum_{i=0}^{2^k-1}|\alpha_{k,i}|\left(\|X_0\|\,\|A\|\right)^i\right]\|X_0\|\,\|b\| \\ &= \varepsilon\sqrt{n}\sum_{i=0}^{2^k-1}|\alpha_{k,i}|\,\|X_0\|^{i+1}\,\|A\|^i\,\|b\|\,. \end{aligned} \tag{5.9}$$

We can simplify this error bound even further when A is diagonally dominant and X_0 is a diagonal matrix; in this case, since $\|I - X_0 A\| < 1$, it follows that $\|X_0\|\,\|A\| \leq 2$, resulting in

$$\begin{aligned}\|\delta(x_k)\| &\leq \varepsilon\sqrt{n}\left[\sum_{i=0}^{2^k-1}|\alpha_{k,i}|2^i\right]\|X_0\|\,\|b\| \\ &= \varepsilon\sqrt{n}\;\alpha\;(2^{2^k}-1)\,\|X_0\|\,\|b\|\,, \end{aligned} \tag{5.10}$$

where $\alpha = \max_i |\alpha_{k,i}|$. Thus we can estimate the error based only on the size of the initial guess X_0 and vector b, which leads us to a way to improve the algorithm further.

5.4 PSJM with Iterative Error Correction

To improve the accuracy of PSJM, so that double precision arithmetic yields double precision results, we can rely on the error estimate (5.10); while we cannot modify the initial guess, we can iteratively modify the size of the vector b, by repeatedly solving $Ax_i = b_i$ for diminishing vectors b_i. By combining the resulting solutions x_i, we obtain a solution for $Ax = b$. We call this refinement of the method PSJM with Iterative Error Correction, or PSJMwIEC.

Algorithm 2 Calculating x_k iteratively

Require: A, X_0, b, k and the number of iterative corrections m

1: $x = 0$, $b' = b$
2: **for** $i = 1 \ldots m$ **do**
3: $\quad x_k^i = \text{PSJM}(A, X_0, b', k)$
4: $\quad x = x + x_k^i$
5: $\quad b' = b' - Ax_k^i$
6: **end for**
7: **return** x

Lemma 5.4.1. *Given a square matrix A, an initial guess X_0 such that $\|I - X_0 A\| < 1$, and a vector b, we can improve the result of the PSJM algorithm by solving $Ax_k = b$ and $Ax' = b - Ax_k$, and returning $x_k + x'_k$.*

Proof. Given that $\|I - X_0 A\| < 1$, we know from Theorem 4.1.1 that X_k will converge to A^{-1}. Since $x_k = X_k b$ and $A^{-1}b = x$, this implies that $Ax_k = AX_k b$ converges to $AA^{-1}b = b$. Hence $\|b - Ax_k\| \to 0$, as k grows large, so $\|b - Ax_k\| < \|b\|$. Let k be fixed, and use the PSJM algorithm to solve $Ax' = b - Ax_k$ for x'_k, so that $Ax'_k = b - Ax_k - Ax'_k = b - A(x_k + x'_k)$.

Now consider $\left\|A[x - (x_k + x'_k)]\right\| = \left\|b - A(x_k + x'_k)\right\| = \left\|(b - Ax_k) - Ax'_k\right\| < \|b - Ax_k\|$; since $\left\|b - A(x_k + x'_k)\right\| < \|b - Ax_k\|$, it follows that $\left\|x - (x_k + x'_k)\right\| < \|x - x_k\|$. □

Algorithm 2 will require at most $m(2^k + 1)$ matrix vector multiplies and $m(2^k + 2)$ vector additions, as a result of running PSJM, at a depth of k iterations m times. If k is relatively small and bounded, we can conclude from Lemma 5.2.3 that the total number of operations required is $\mathcal{O}(mn^2)$ in the general case, or using Lemma 5.2.4 $\mathcal{O}(mn \log n)$ when A and X_0 are Toeplitz matrices.

5.5 Experimental Results

In this section we will discuss the experimental results; after briefly describing the methodology of our experiments, we will consider three distinct cases. Since this research was motivated by the use of stochastic interpolation, which requires the use of near diagonally dominant Toeplitz matrices, we first consider the diagonally dominant matrices, specifically diagonally dominant Toeplitz matrices. Next we consider tri-diagonally dominant matrices, which extends the range of matrices we can cover, and lastly we discuss Gaussian matrices, which are the actual matrices used in stochastic interpolation. Gaussian matrices are typically Toeplitz or row stochastic, and positive. While we discuss several variations of diagonal dominance in Sec. 4.2, especially the concept of extended diagonal dominance in Theorem 4.2.2, we will focus strictly on diagonally dominant matrices as the extensions all rely on reordering the rows of A (and values of b) to create a matrix that is diagonally dominant. Depending on the method used to store the matrix in memory, whether array based or list based, this reordering will require $\mathcal{O}(n^2)$ memory operations if the matrix is stored as an array or $\mathcal{O}(n)$ memory operations if the array is stored as a list of lists. When counting the number of operations, however, we generally explicitly consider only arithmetic operations so that these memory operations that occur as values are moved and swapped around in memory are left uncounted. These memory operations are uncounted because they are, relatively speaking, much faster than the arithmetic operations and thus the arithmetic operations form the bottleneck when studying the speed or complexity of an algorithm.

5.5.1 Methodology

We implemented our testing software in C, using the Gnu Scientific Library (gnuscl) [15], the lapack [19] library, and, in the cases where we were dealing with Toeplitz matrices, the fftw library subroutine [12]. The code is implemented in double precision; gnuscl supports greater precision for the basic matrix and vector manipulations, including initialization, addition and scalar multiplication, but relies on lapack for optimized matrix-matrix and matrix-vector multiplication. The lapack package was first written in Fortran 77, but later upgraded to Fortran 90 in 2008. The programming language Fortran is widely used, suited as it is to numeric computation, but Fortran 90 offers only single and double precision, thus limiting the precision of the libraries. Like Fortran, C is also an imperative programming language with similar memory management routines; the choice to use C rather than Fortran for the front end allowed us to structure the code in a way that simplified adding test cases by using function pointers.

The testing software is designed in a modular fashion. There are separate routines for the creation of random matrices A under a variety of parameters that include the type of matrix A should be, such as diagonally dominant and tridiagonally dominant matrices and the Gaussian matrices used for stochastic interpolation. Secondly, there are separate routines to create the initial guess X, based on the matrix A; for a diagonally dominant matrix A, X consists of a scaling matrix, whereas for a tridiagonally dominant matrix X is initialized as the inverse of D_3, where D_3 is the matrix including the diagonal and super and sub diagonals of A. The specific calculation of the initial guesses is described in Sec. 4.2. Thirdly, there are separate routines to calculate the PSJM results, based on the structure of A. For instance, if A is diagonally dominant, then X is simply a scaling matrix, and each matrix vector multiply involving X can be replaced with a simple vector scaling, resulting in an optimized number of operations. In contrast, if A and X are both Toeplitz matrices, then the matrix vector multiply can use the FFT method, resulting in a reduction from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$ operations per matrix-vector multiply. Sec. 3.2.1 explains how the matrix-vector multiply can be performed faster, using an embedding of the Toeplitz matrices in circulant matrices.

For each test run three main values are recorded; the difference between the lapack and PSJM solution, the residual error, and the amount of time required. Thus we define the error as $\|x_l - x_k\|_\infty$, where x_l is the solution generated by the lapack libraries and x_k is the solution generated by PSJM after k iterations. The residual error is calculated as $\|b - Ax_k\|_\infty$. The time required is calculated using the system clock, by recording when PSJM starts and stops, and returning the difference between the two times.

Generally, the SJM method is not tested, since it is much slower even than the lapack

library routine; for $k = 3,4$, both SJM and PSJM yield exactly the same results using double precision, while PSJM provides an upper bound for the error when $k = 5$. The slight divergence is due to larger number of matrix operations required for PSJM. As discussed in Sec. 5.3, PSJM is not reliable for $k = 6$ when using double precision due to the inability to store the coefficients $\alpha_{k,j}$ accurately.

5.5.2 Implementing PSJM

Recall, when implementing PSJM, that the goal is to calculate (5.2), which is

$$x_k = \left(\sum_{i=0}^{2^k-1} \alpha_{k,i} (X_0 A)^i \right) X_0 b.$$

To do so, the implementation maintains two vectors simultaneously; $XAXb = (XA)^j Xb$, for $j = 0 \dots 2^k - 1$, and $xk = \left(\sum_{i=0}^{j} \alpha_{k,i} (X_0 A)^i \right) X_0 b$, for $j = 0 \dots 2^k - 1$. The intermediate vectors are calculated using Algorithm 3, which minimizes the number of calculations required by never recalculating a value. Even so, it is possible to improve the number of calculations required by implementing faster matrix-vector multiplies in lines 4 and 5, depending on the structure of A and X. Brute-forcing the matrix vector multiplications requires the use of $\mathcal{O}(n^2)$ calculations; if either matrix is Toeplitz, however, this step can be implemented in $\mathcal{O}(n \log n)$ calculations, and if X is sufficiently simple, line 5 can be subsumed in line 4 or line 6.

Algorithm 3 Calculating x_k

Require: A and X are $n \times n$ matrices, b is a vector of size n
Require: The coefficients $\alpha(k,i)$ are known.
1: $XAjXb = X.b$ ▷ The variable $XAjXb$ stores the intermediate vector $(XA)^j Xb$
2: $xk = \alpha(k,0) \times XAjXb$ ▷ The variable xk stores the partial vector $\left(\sum_{i=0}^{j} \alpha_{k,i} (X_0 A)^i \right) X_0 b$
3: **for** $j = 1 \dots 2^k - 1$ **do**
4: $\quad XAjXb = A.XAjXb$
5: $\quad XAjXb = X.XAjXb$
6: $\quad xk = xk + \alpha(k,j) \times XAjXb$
7: **end for**
8: **return** xk

5.5.3 Diagonally Dominant Matrices

When the matrix A is diagonally dominant, the initial guess X is a scaling matrix. Using Theorem 4.2.1, we initialize X so that, if $m = \max_{i=1\dots n} |A[i,i]|$, then $X[i,i] = \text{sgn}(A[i,i])/m$,

with all other entries 0. If A has no negative values on the diagonal, we can simplify X further, by setting $X[i,i] = 1/m$, so that X simply scales A. Then the calculations in lines 4 and 5 can be combined, using the gnuscl routine `gsl_blas_dgemv`, which takes as parameters the matrix and vector that will be multiplied together, as well as scalar coefficients, so that `gsl_blas_dgemv(CblasNoTrans,1/m,A,b,0,c)` calculates $c = (1/m)Ab + 0c$. This allows us to reduce the amount of calculations significantly. We see the effects of these choices in Figures 5.1 through 5.4 and Table 5.3.

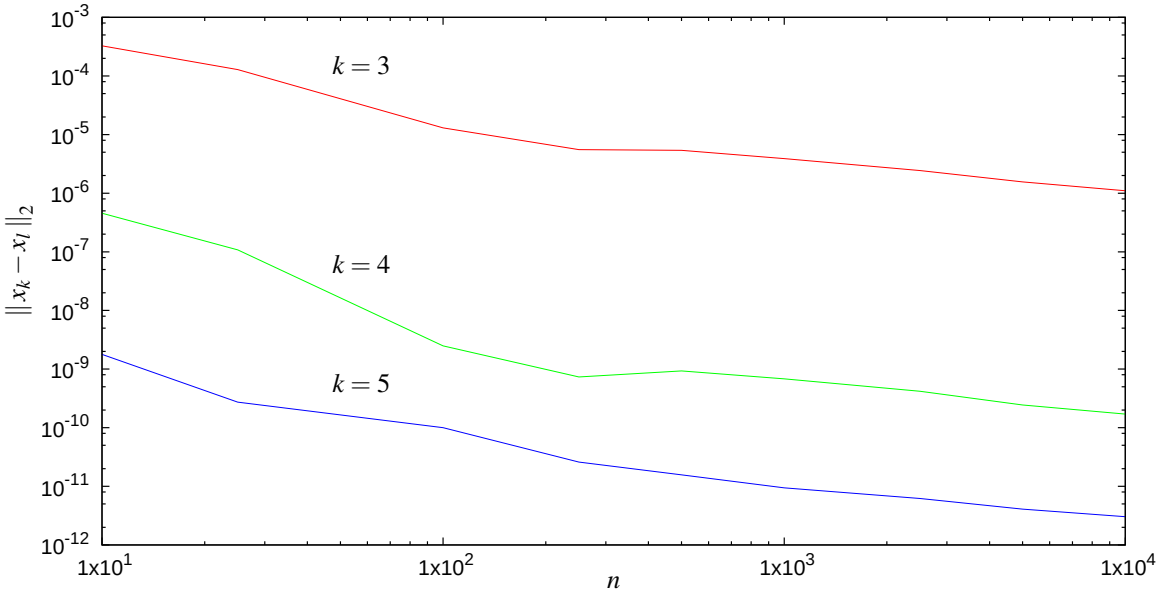


Figure 5.1: The error $\|x_l - x_k\|_2$, where x_k is computed using PSJM and x_l is the solution computed by the lapack library, as the size of the matrix is varied from $n = 10$ to $n = 10,000$ and the depth of recursion varies from 3 to 5, averaged over 50 diagonally dominant Toeplitz matrices at each size.

To generate the results in Fig. 5.1, we used random diagonally dominant matrices that were also Toeplitz matrices. Thus we can use the method described in Sec. 3.2.1 to perform the matrix vector multiplication in $\mathcal{O}(n \log n)$ time, calculating the result x_k rapidly, as we can see in Fig. 5.2. Note from the slope of the curves and the comparison $n \log n$ slope, that we are achieving the $\mathcal{O}(n \log n)$ timing we expect, using CPU time as a proxy for the number of arithmetic operations. We can improve the accuracy by applying the iterative error correction, as described in Sec. 5.4, as in Figures 5.3 and 5.4, where we see that applying the iterative error correction allows us to improve the error dramatically; after only one round of iterative error correction the error is halved, and a second round of iterative correction decreases the error further. This allows us to take advantage of the quadratic convergence of the technique, while overcoming the challenge of overly large coefficients

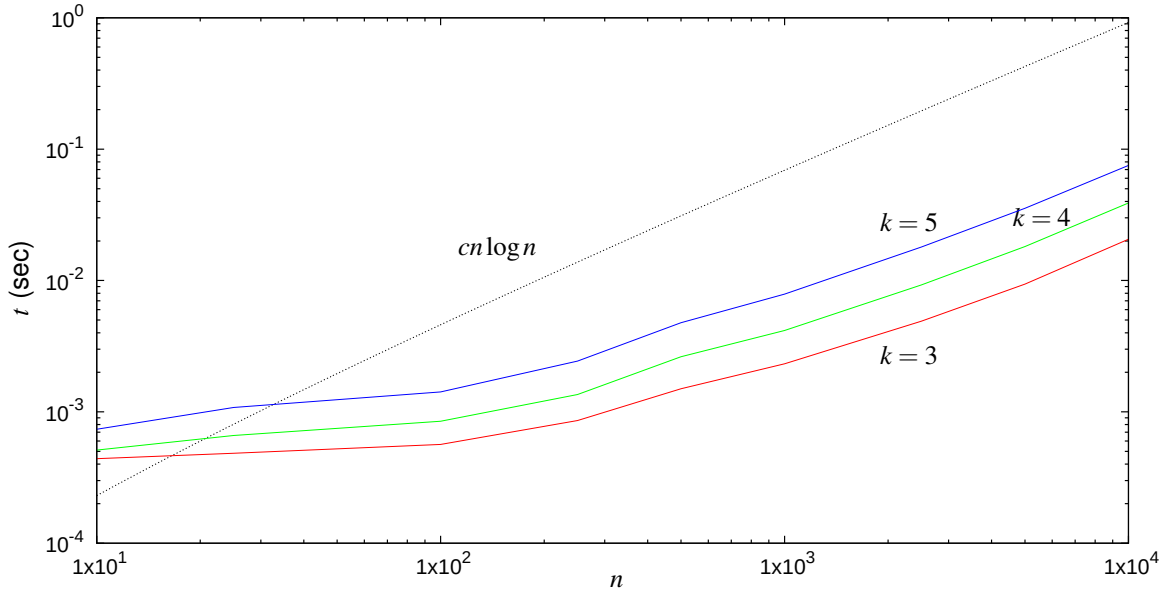


Figure 5.2: The amount of time required, in seconds, to calculate x_k for matrices whose size ranges from $n = 10$ to $n = 10,000$, averaged over 50 random diagonally dominant matrices at each size. Also included is the line $y = n\log n$, downshifted c units, as a comparison.

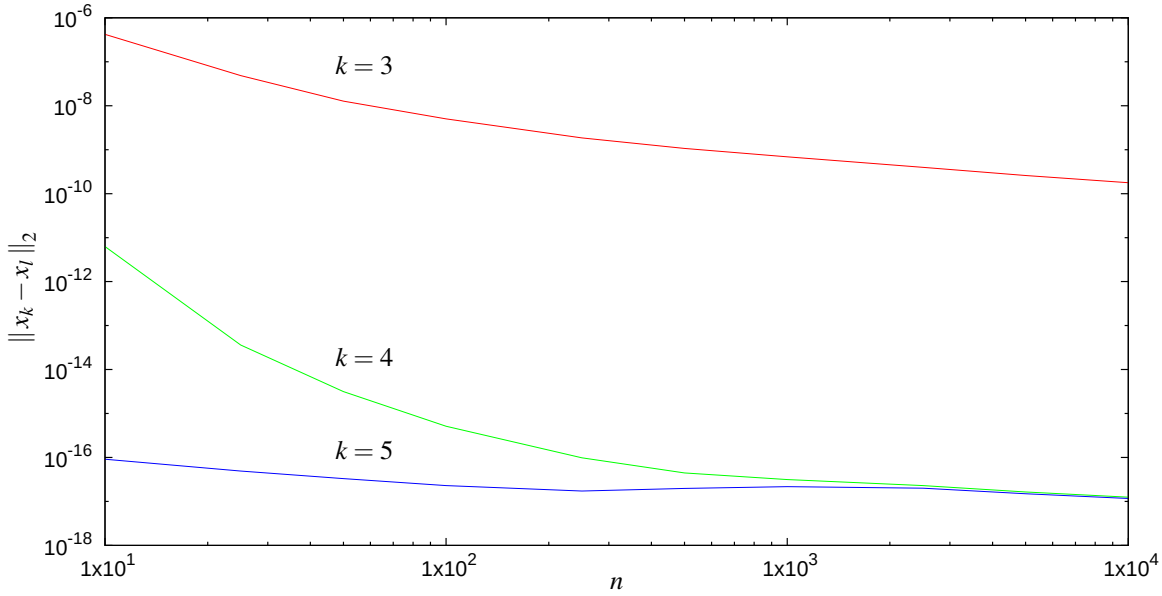


Figure 5.3: The error $\|x_l - x_k\|_2$, where x_k is computed using PSJM with one iterative error correction and x_l is the solution computed by the lapack library, as the size of the matrix is varied from $n = 10$ to $n = 10,000$ and the depth of recursion varies from 3 to 5, averaged over 50 diagonally dominant Toeplitz matrices at each size.

$\alpha(k,i)$ that cannot be stored in fixed precision memory.

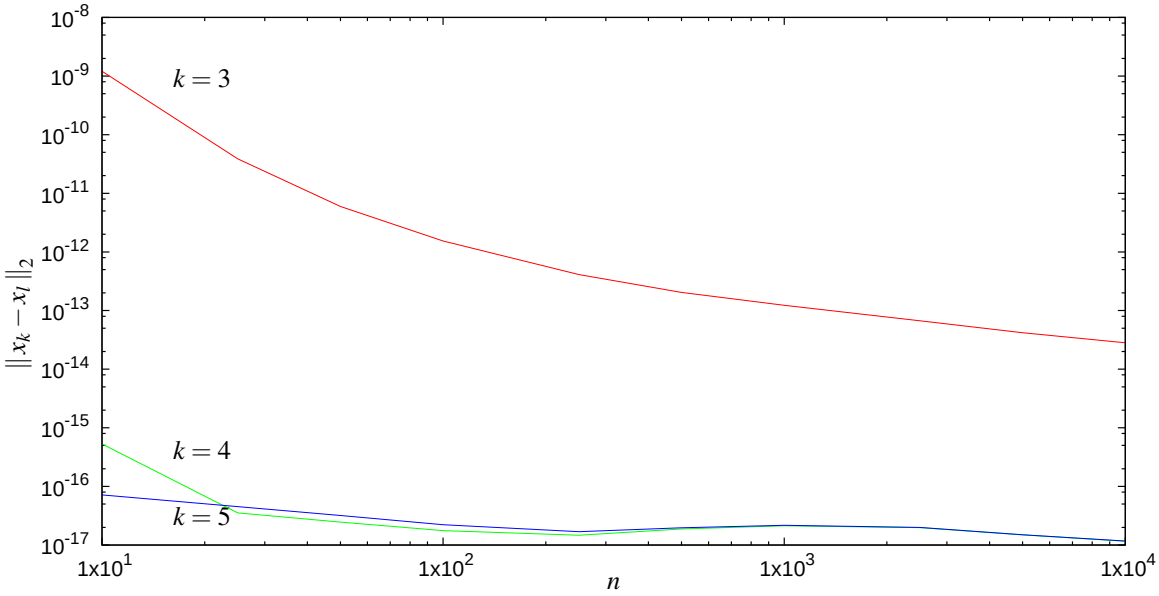


Figure 5.4: The error $\|x_l - x_k\|_2$, where x_k is computed using PSJM with two iterative error corrections and x_l is the solution computed by the lapack library, as the size of the matrix is varied from $n = 10$ to $n = 10,000$ and the depth of recursion varies from 3 to 5, averaged over 50 diagonally dominant Toeplitz matrices at each size.

n	k	Number of Iterative Corrections	Time (seconds)	$\|x_k - x_l\|_2$
1000	3	0	0.001881647	3.89×10^{-06}
		1	0.006025781	6.87×10^{-10}
		2	0.009785614	1.23×10^{-13}
1000	4	0	0.003716688	6.87×10^{-10}
		1	0.009754291	3.15×10^{-17}
		2	0.015440168	2.17×10^{-17}
1000	5	0	0.007422156	8.94×10^{-12}
		1	0.017100935	2.17×10^{-17}
		2	0.026493812	2.17×10^{-17}

Table 5.3: A comparison of the time required to calculate x_k using PSJM, with and without iterative error corrections, as the depth of the recursion varies from 3 to 5, averaged over 50 diagonally dominant Toeplitz matrices of size 1000×1000.

5.5.4 Tri-Diagonally Dominant Matrices

If a matrix A is diagonally dominant, it is guaranteed by Theorem 4.2.1, to have a known, and easily computable, initial guess X. In contrast, not every tridiagonally dominant matrix

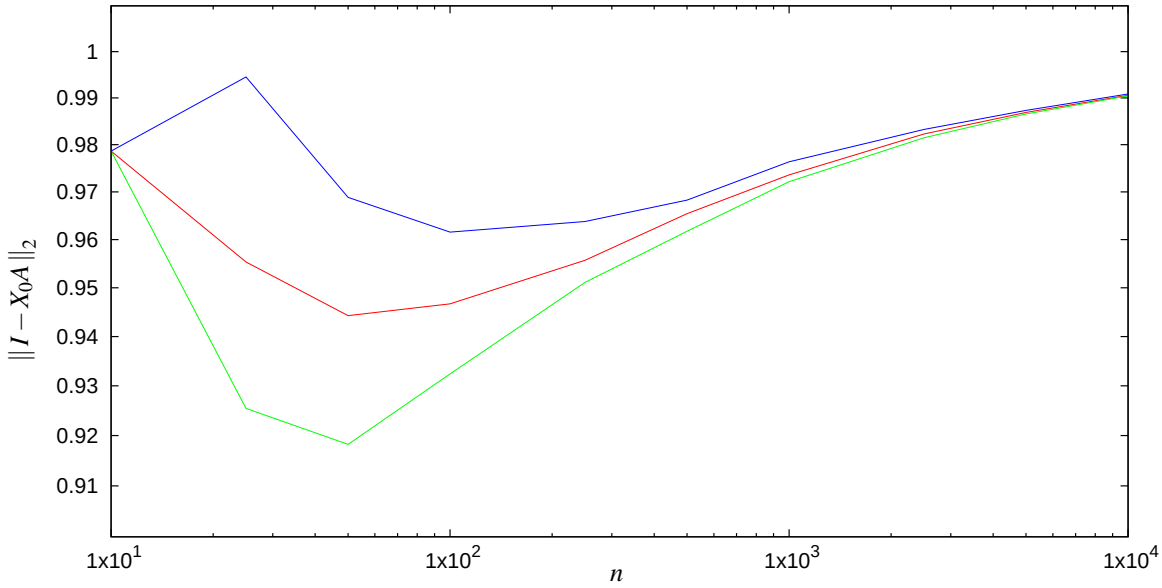


Figure 5.5: The minimum, maximum, and average values for the accuracy of the initial guess, calculated as $\|1 - X_0A\|_2$, over 20 tridiagonally dominant matrices at each size.

is invertible. Consider, for instance, the matrix

$$A = \begin{bmatrix} 0 & 2 & 0 & \dots & 0 \\ 2 & 0 & 2 & \ddots & \vdots \\ 0 & 2 & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 2 \\ 0 & \dots & 0 & 2 & 0 \end{bmatrix}$$

is clearly tridiagonally dominant but not invertible; if A were invertible, its inverse could be calculated using (4.19), and the zeroes on the diagonal would result in a division by zero error. Thus we turn our attention to matrices whose tridiagonal sub-matrix is invertible, as well as dominant, and we further consider the requirement that, if $A = D_3 + M$, the initial guess $X = D_3^{-1}$ must also satisfy $\|XM\| < 1$, as in Theorem 4.2.4. Thus we need to consider invertible tridiagonally dominant matrices who also satisfy $\|XM\| < 1$. Furthermore, even if A is a Toeplitz matrix, the initial X will not be Toeplitz, so that we cannot take advantage of the faster matrix-vector multiplication. This leaves us with an algorithm requiring $\mathcal{O}(n^2)$ operations per matrix-vector multiplication. Figures 5.5 through 5.7 and Table 5.4 illustrate the results of studying tridiagonally dominant matrices.

Before we can consider the effectiveness of the algorithm, we must first consider the effectiveness of the initial guess. Table 5.4 considers the number of iterations required before reaching an answer with single precision when $0.9 \leq \|I - X_0A\| < 1$. We can see that if $\|I - X_0A\| = 0.99$, eleven iterations will be required before reaching single precision

$\|I - X_0A\|$	k	$\|I - X_0A\|^k$	$\|I - X_0A\|^5$
0.99	11	1.15×10^{-09}	7.25×10^{-01}
0.98	10	1.04×10^{-09}	5.24×10^{-01}
0.97	10	2.85×10^{-14}	3.77×10^{-01}
0.96	9	8.37×10^{-10}	2.71×10^{-01}
0.95	9	3.93×10^{-12}	1.94×10^{-01}
0.94	9	1.74×10^{-14}	1.38×10^{-01}
0.93	8	8.54×10^{-09}	9.81×10^{-02}
0.92	8	5.37×10^{-10}	6.94×10^{-02}
0.91	8	3.27×10^{-11}	4.89×10^{-02}
0.90	8	1.93×10^{-12}	3.43×10^{-02}

Table 5.4: Number of iterations k required before $\|I - X_kA\| < 10^{-14}$, given $\|I - X_0A\|$ and quadratic convergence, as well as the bound on the error after 5 iterations.

accuracy, and after five iterations, the effective limit of PSJM when using double precision, will reach an accuracy bounded by 0.725, a handicap we can overcome by using iterative error correction as required. To consider the range of $\|I - X_0A\|$ we can expect, we generated 20 matrices at each size n, where n varied from 10 to $10,000$, and calculated the minimum, maximum, and mean of $\|I - X_0A\|$ for all matrices at that size. Fig. 5.5 shows that there is some variation when n, the size of A, is small, but that the minimum, mean, and maximum grow close to 1 as n grows large. Experimentally, we see that the results are not quite as dire as the worst-case scenario suggests, and Fig. 5.6 shows the measured error after 5 iterations. Furthermore, iterative error correction allows for further improvements as we can see in Fig. 5.7, and continuing iterative corrections would result in further convergence to a reasonable precision, even as n grows large.

5.5.5 Gaussian Matrices

Gaussian matrices are initialized using (4.21); if the spacing is even, then the matrix A is determined by its size n and the mollifier α, and the initial guess X is defined by (4.22). Thus $X_k = 2X_{k-1} - X_{k-1}AX_{k-1}$ is considered constant for any set of n equally spaced data points, but x_k will vary, depending on the value of b. Figures 5.8 through 5.12 illustrate the effects of varying b, and the time required to calculate x_k.

Since b is randomly populated with uniformly distributed values between 0 and 1, the expected value for each entry is 0.5, and $\|b\| \to 0.5n$ as n grows large. Thus, in Fig. 5.8, which shows the minimum, mean, and maximum error for 20 random vectors b of each size n, the minimum and maximum errors come very close together. As we can see in

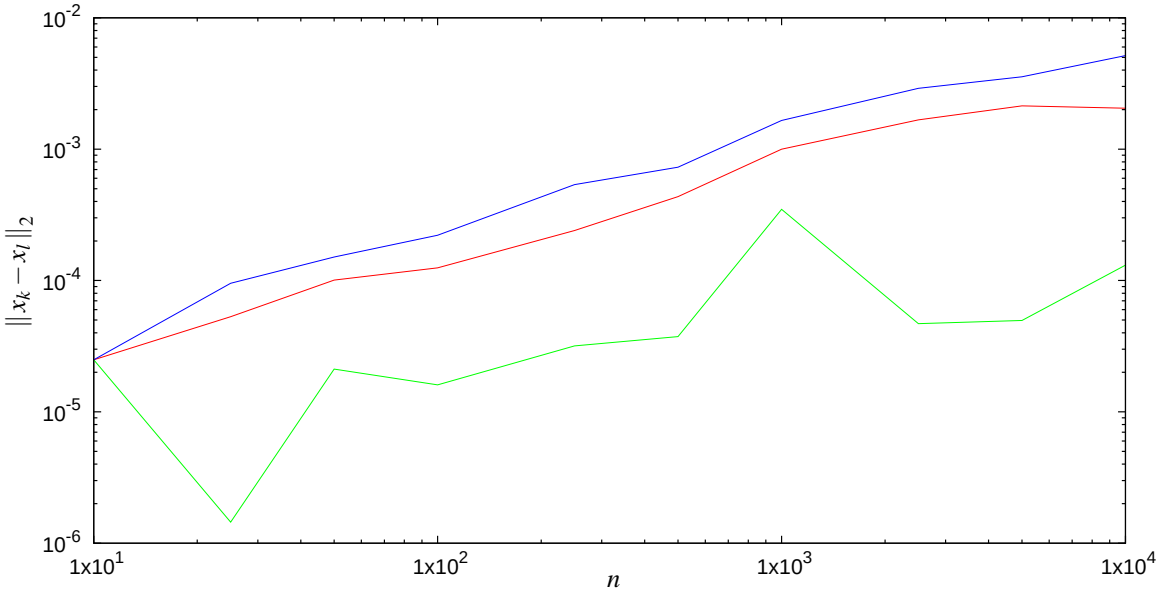


Figure 5.6: The minimum, maximum, and average error $\|x_l - x_k\|_2$, where x_k is computed using PSJM and x_l is the solution computed by the lapack library, as the size of the matrix is varied from $n = 10$ to $n = 10,000$ and the depth of recursion is 5, averaged over 20 tridiagonally dominant matrices at each size.

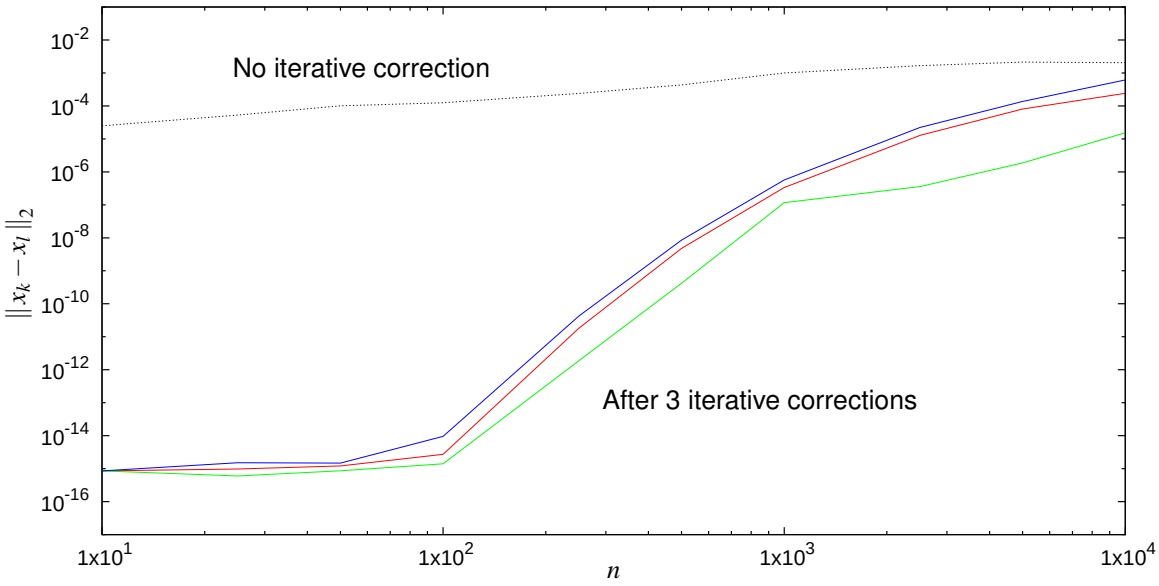


Figure 5.7: The minimum, maximum, and average error $\|x_l - x_k\|_2$, where x_5 is computed using PSJMwIEC and 3 iterative corrections, and x_l is the solution computed by the lapack library, as the size of the matrix is varied from $n = 10$ to $n = 10,000$, averaged over 20 tridiagonally dominant matrices at each size. The average $\|x_l - x_k\|$ without iterative error correction is included for comparison.

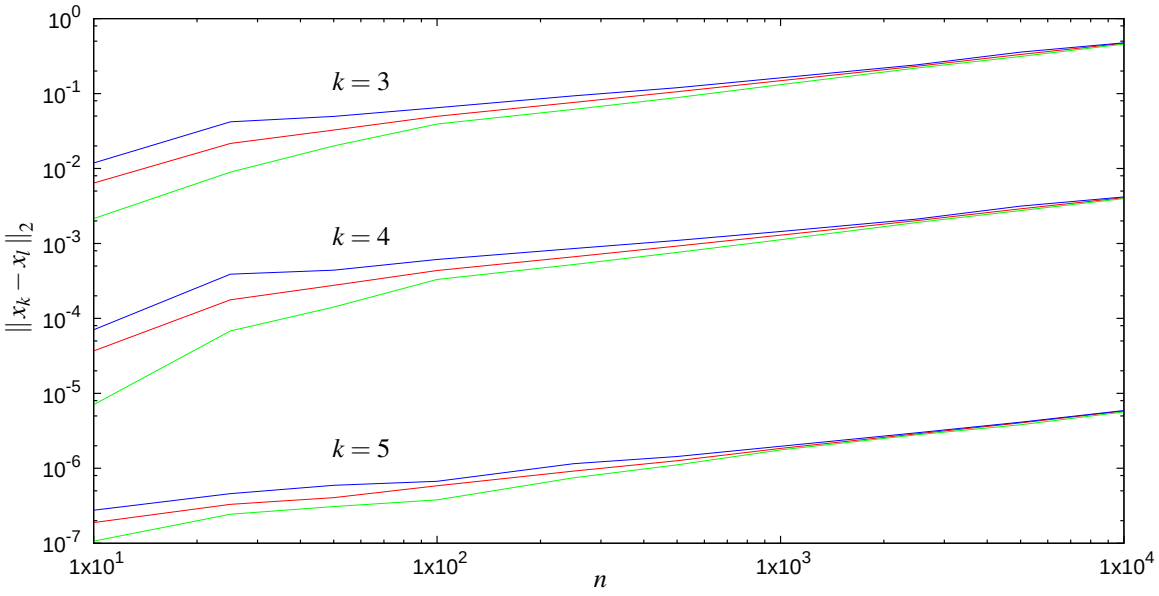


Figure 5.8: The error $\|x_k - x_l\|_2$, where x_k is computed using PSJM and x_l is the solution computed by the lapack library, as the size of the matrix is varied from $n = 10$ to $n = 10,000$ and the depth of recursion varies from 3 to 5, using the Gaussian matrix for evenly spaced data points, and averaged over 20 random vector b. For each depth of recursion, the minimum, maximum, and average errors are shown.

Fig. 5.9 which illustrates the error after applying three iterative error corrections, iterative error correction is particularly effective for Gaussian matrices, even for large n. Contrast this with tridiagonally dominant matrices in Fig. 5.7, where it is clear that as n grows large, many more iterative corrections will be required.

By modifying A and X_0 slightly, using (4.23) to evaluate (4.21) and (4.22), we are left with two Toeplitz matrices A and X_0 which differ from the original matrices only in the first and last row. Taking advantage of the Toeplitz structure, the number of operations required by PSJM can be reduced significantly to $\mathcal{O}(n \log n)$, as per Lemma 5.2.4. There is a secondary cost in initializing A and X_0 and setting up the FFT routines, and for small n, the $\mathcal{O}(n^2)$ implementation returns results more rapidly. Where Fig. 5.11 shows the time required for the non-Toeplitz version, after 2 iterative corrections, Fig. 5.12 shows the time required for the Toeplitz version using the FFT library. As we can see from Fig. 5.9 and 5.10, the error is equivalent for both implementations, leaving time as the deciding factor. Fig. 5.12 particularly shows the difference, as the dotted line shows the amount of time used to solve the simplest case, where $k = 3$, for the $\mathcal{O}(n^2)$ implementation, where the colored lines show the time required using the Toeplitz matrix and the FFT library.

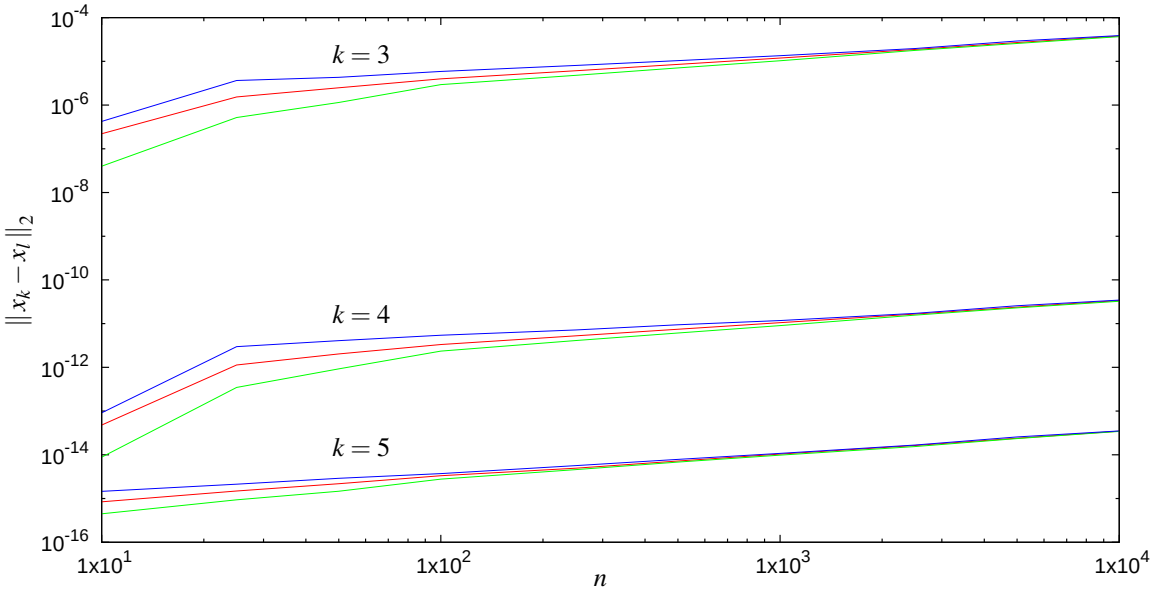


Figure 5.9: The error $\|x_l - x_k\|_2$, where x_k is computed using PSJM with 3 iterative error corrections and x_l is the solution computed by the lapack library, as the size of the matrix is varied from $n = 10$ to $n = 10,000$ and the depth of recursion varies from 3 to 5, using the Gaussian matrix for evenly spaced data points, and averaged over 20 random vector b. For each depth of recursion, the minimum, maximum, and average errors are shown.

5.6 Summary

As we can see from the experimental results using diagonally dominant, tridiagonally dominant, and Gaussian matrices, the PSJM algorithm offers several definite advantages over the SJM algorithm: PSJM runs faster, by removing the need for matrix-matrix multiplications and replacing them with matrix-vector multiplications. As long as $n > 2^k$, even one matrix matrix multiplication, requiring $2n^3$ operations, will require more work than 2^k matrix vector multiplications, each requiring $2n^2$ operations. Granted, when the matrix is centrosymmetric or Toeplitz we can reduce the number of operations the matrix-matrix multiplication requires, but considering we will still need to perform 2 matrix matrix multiplications and one matrix sum at every iteration, SJM will be faster only if $n \ll 2^k$, which makes for very small values of n.

However, when implementing PSJM using fixed precision computation, the number of possible iterations is small due to the rapid growth of the coefficients. Specifically, using double precision, we can use at most 5 iterations. This weakness implies that the initial guess X_0 has to be very good, so that $\|I - X_0 A\| \ll 1$. We overcome this weakness by using the error bounds established in (5.9) and observing that the error bound depends only on $\|A\|$, $\|X_0\|$, and $\|b\|$; thus iterative error correction as introduced in Sec. 5.4 reduces the

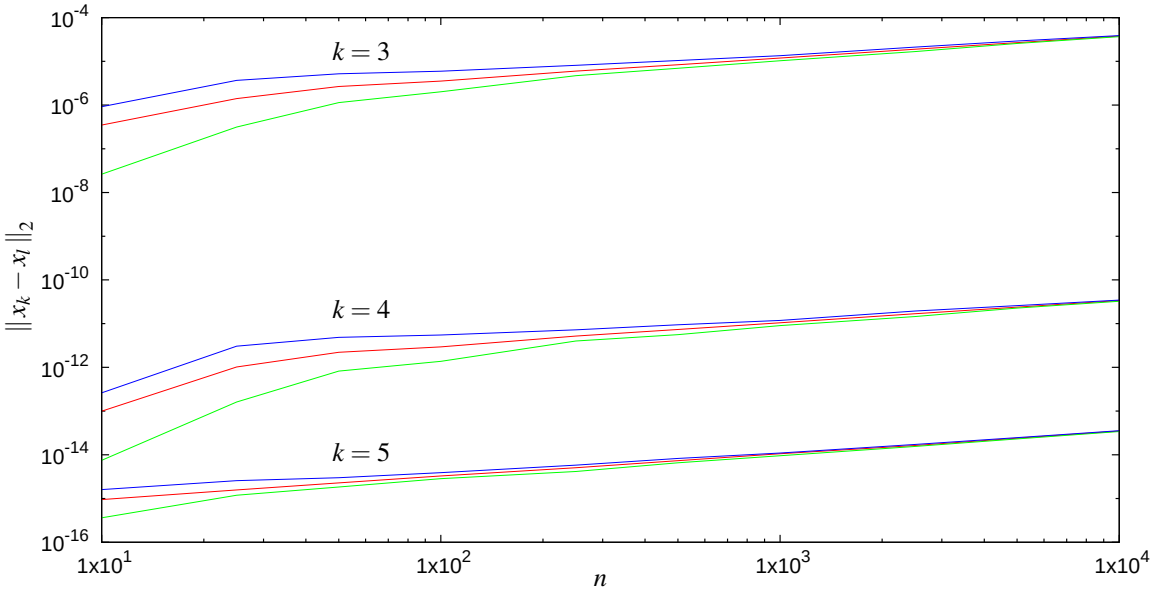


Figure 5.10: The error $\|x_k - x_l\|_2$, where x_k is computed using PSJM with 2 iterative error corrections and x_l is the solution computed by the lapack library, as the size of the matrix is varied from $n = 10$ to $n = 10,000$ and the depth of recursion varies from 3 to 5, using the Toeplitz Gaussian matrix for evenly spaced data points, and averaged over 20 random vector b. For each depth of recursion, the minimum, maximum, and average errors are shown.

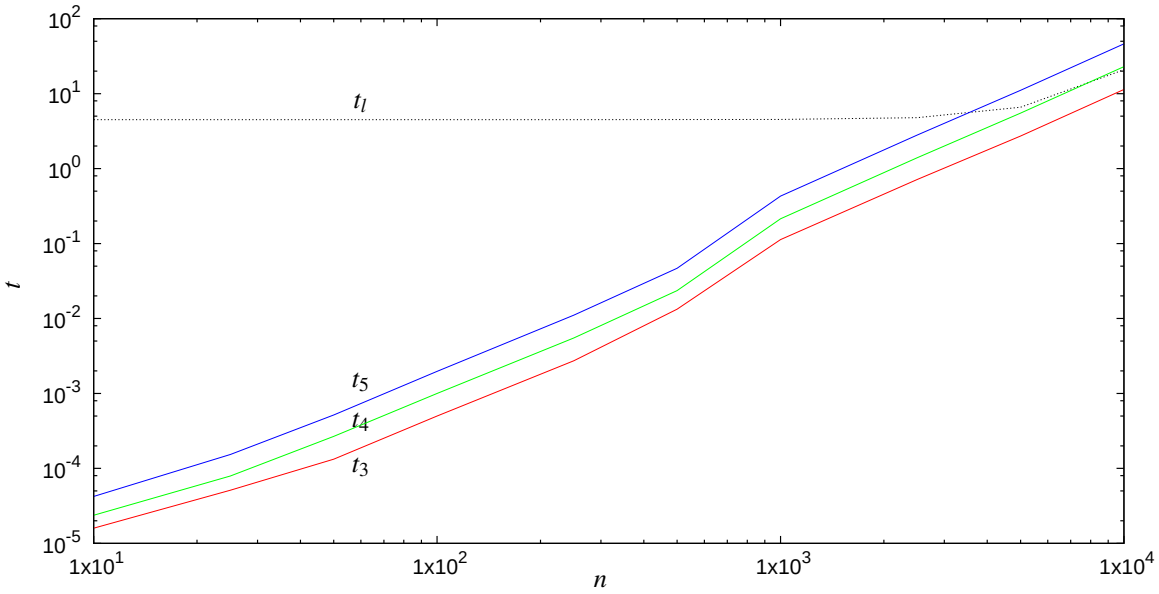


Figure 5.11: Time t_k required to calculate x_k by PSJM with 2 iterative corrections, as the size of the matrix is varied from $n = 10$ to $n = 10,000$ and the depth of recursion varies from 3 to 5, using the Gaussian matrix for evenly spaced data points, and averaged over 20 random vector b. Also the time t_l required by the lapack library to calculate x_l.

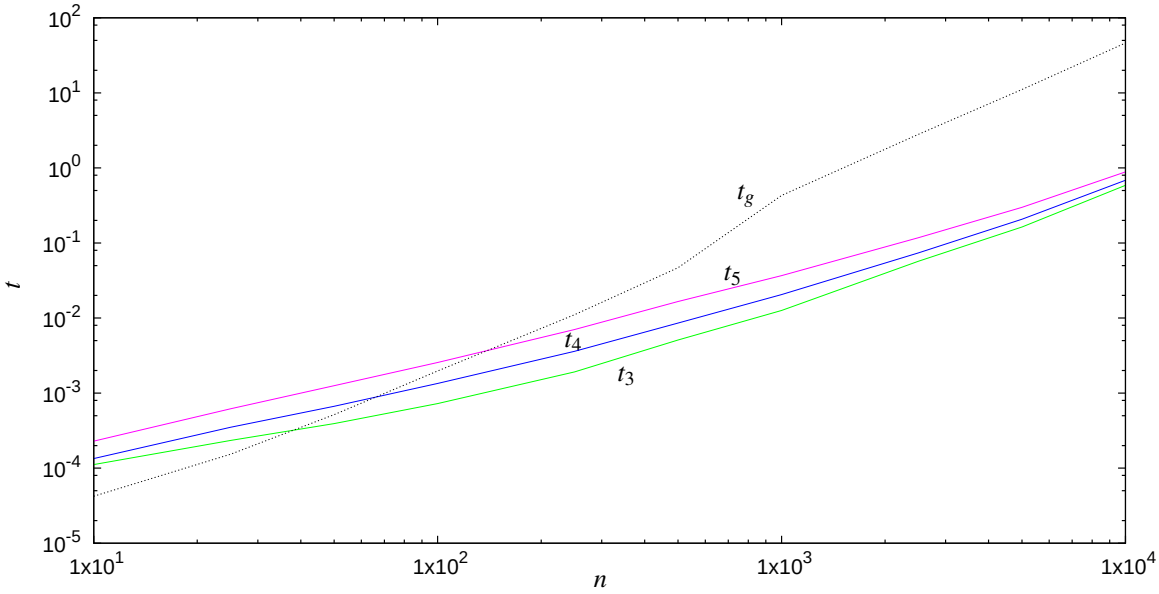


Figure 5.12: Time t_k required to calculate x_k by PSJM with 2 iterative corrections, as the size of the matrix is varied from $n = 10$ to $n = 10,000$ and the depth of recursion varies from 3 to 5, using the Toeplitz Gaussian matrix for evenly spaced data points, and averaged over 20 random vector b. Also the time t_g required to calculate x_3 for a non-Toeplitz Gaussian matrix.

error by solving for consecutively smaller vectors b, resulting in the improving quality of x_k.

Experimentally we see that the iterative error correction will allow PSJMwIEC to continue converging after the first 5 iterations, even in cases like the tridiagonally dominant matrices where the initial guess is not particularly good, and typically $1-\varepsilon < \|I - X_0 A\| < 1$ for some small ε. In these cases the iterative error correction will eventually converge to single, or even double, precision accuracy, but it may require multiple iterative corrections. Considering the Gaussian matrices, we see that varying the vector b from case to case does not particularly affect the accuracy, but that using the Toeplitz version of A and X_0 provides faster results.

Chapter 6

Applications

6.1 Overview

In this chapter we will illustrate how to use PSJM to perform the stochastic interpolation described in Chapter 2, both for the simple single variable functions described in Chapter 2, and multivariable functions, as well as interpolating entire images. We saw in Fig. 2.2 and 2.4 that adding more data points added to the accuracy of the interpolating function, allowing us to improve our interpolation by further sampling. We can also manipulate the interpolation by varying the value of the mollifier α; for small values of α the interpolation becomes almost a step function, whereas the interpolation becomes more smooth and rounded as α grows. We can see this effect in Fig. 6.1, where α varies from 0.01 to 0.20. We note that the result of stochastic interpolation is a vector of the interpolated points between the data points; as such the number of operations required will depend on both the number of data points n as well as the number of interpolated points m. Therefore, we will also consider the number of operations required for the various types of stochastic interpolation; the single variable function, the multi-variable function, and the image.

6.2 Complexity of Stochastic Interpolation

Algorithm 4 Calculating stochastic interpolation of $\{x, f(x)\}$ with m output points.

Require: The vector $f = f(x)$ of size n

Require: x consists of evenly spaced points from a to $a+b$

1: Scale x so that $x_1 = 0$ and $x_n = 1$, by shifting a and multiplying by factor b.
2: Initialize A_{nn} and X_0 using (4.21) and (4.22), where the y_k are initialized using (4.32).
3: Calculate $x_k = \textbf{PSJM\_wIEC}(A_{nn}, X_0, f, 5, 3)$ using Algorithm 2
4: $\triangleright$ $x_k \approx A_{nn}^{-1} f$, using X_0 as initial guess, f as the vector, a recursive depth of 5, and 3 iterations.
5: Initialize A_{mn} using (4.21), where the y_k are initialized using (4.32).
6: $f' = A_{mn} x_k$
7: $x'_i = b/(n-1) + a$ for $i = 1 \ldots m$
8: **return** $\{x', f'\}$

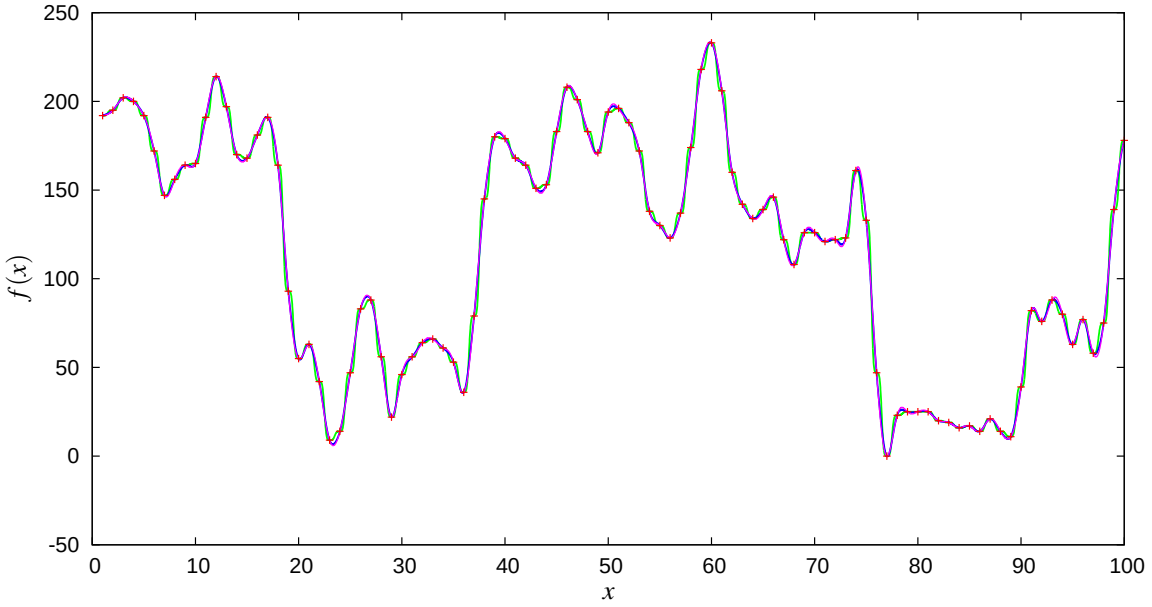


Figure 6.1: Interpolating a one-dimensional set of 100 data points $\{x, f(x)\}$ (red), which consist of a snapshot of one row of an image. The interpolation uses a Gaussian matrix with $\alpha = 0.01$ (green), 0.1 (blue), 0.20 (violet) and interpolates over 1000 output points.

The first case we consider is the single variable function, where the input consists of the evenly spaced data points $\{x, f(x)\}$, and the stochastic interpolation algorithm is implemented as in Algorithm 4. Then the number of operations required will depend on the size n of the input and the size m of the output. Fig. 6.1 is an example of such an interpolation, where $n = 100$ and $m = 1000$, giving us a ten-fold zoom. As we see from Lemma 6.2.1, this will require $\mathcal{O}(mn + m\log m)$ operations, but since $\log m < n$, the complexity will effectively be $\mathcal{O}(mn)$ as the work required is dominated by the initialization of A_{mn}; we could thus improve the algorithm in situations where we need to calculate similar problem sets by precalculating and storing A_{mn}, so that the majority of the work required lies in calculating x_k and multiplying $A_{mn}x_k$.

Lemma 6.2.1. *Given input consisting of n evenly spaced data points* $\{x, f(x)\}$*, and a desired output of m evenly spaced output points* $\{x', f'(x')\}$ *where* $x_1 = x'_1$*,* $x_n = x'_m$*, and* $f(x_i) = f'(x_i)$ *for all input* x_i*, Algorithm 4 requires* $\mathcal{O}((mn) + m\log m)$ *operations.*

Proof. We can assume that $m > n$. The scaling performed in lines 1 and 7 require $2n$ and $2m$ operations respectively. Initializing A_{nn} and X_0 requires $\mathcal{O}(n^2)$ operations, and initializing A_{mn} will require $\mathcal{O}(mn)$ operations.

Since both A_{nn} and A_{mn} are Toeplitz, and hence can be embedded in circulant matrices of

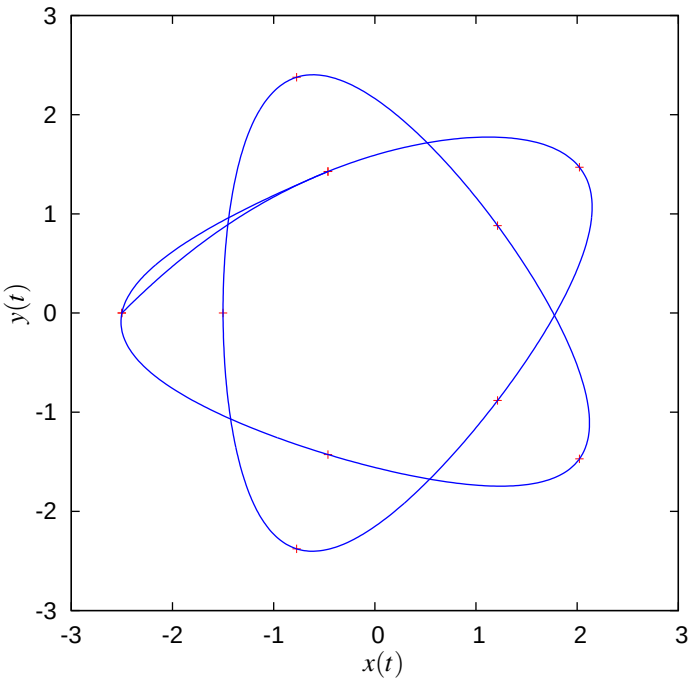


Figure 6.2: Interpolating the function $f(t) = (x(t), y(t)) = (-1/2\sin t - 2\cos(2t/3), (-1/2\cos t - 2\cos(2t/3))$, over the region $t = [0, 33/5\pi]$. The data points are marked in red; the points $(0, 2.5)$ and $(-0.147, 1.093)$ are used twice.

size $(2n-1)\times(2n-1)$ and $(m+n-1)\times(m+n-1)$ respectively and since $m+n-1 < 2m$, it is possible to perform matrix vector multiplications in $\mathcal{O}(n\log n)$ and $\mathcal{O}(m\log m)$ time so that line 6 requires $\mathcal{O}(m\log m)$ time. Since both the depth of the recursion, and the number of iterations, is bounded and constant, line 3 requires $\mathcal{O}(n\log n)$ operations by Lemma 5.2.4.

Hence, depending on whether $n > \log m$, Algorithm 4 requires either $\mathcal{O}(mn)$ operations if $n > \log m$ or $\mathcal{O}(m\log m)$ operation if $n < \log m$. □

In general, stochastic interpolation can be performed with any centrally symmetric probability density function, and we consider a second probability distribution function in Sec. 6.4. We note that if the resulting matrices A and X_0 are not Toeplitz but have some other structured form, the algorithm will require $\mathcal{O}(mn + f(m,n))$ operations, where $\mathcal{O}(f(m,n))$ is the number of operations required to perform matrix vector multiplications for that other structured form. Thus, if the resulting matrix is centrosymmetric in general, $f(m,n) = cmn$, and the overall number of operations required is $\mathcal{O}(mn)$.

6.3 Multi-variable Stochastic Interpolation

It is possible to perform stochastic interpolation over multiple dimensions. For example, a function $f(t) = (x(t), y(t))$ can be interpolated by interpolating both $x(t)$ and $y(t)$ separately, and in general $f(t) = (y_1(t), y_2(t), \ldots, y_p(t))$ can be interpolated by interpolating each of the p individual functions. As an example of a two-dimensional case, consider Fig. 6.2, where the data consists of a set of points $\{(x_i, y_i)\}$, and we interpolate $x(t)$ and $y(t)$ separately to form the final image. In this instance we have chosen to extend the domain past one full rotation: using the domain $[0, 6\pi]$ would have resulted in an interpolation whose leftmost point at $(0, -2.5)$ would be a sharp point rather than a smoothed one.

Complexity for these multiple interpolations then becomes $\mathcal{O}(mn + pm \log m)$, since we can amend Algorithm 4 to calculate a separate x_k for each of the $\{y_i\}_{i=1}^p$; thus lines 3 and 6 will each be called p times. Since the matrices A_{nn}, X_0 and A_{mn} only need to be calculated once, the cost of initialization will not increase, but the cost of calculating each x_k and each $f'(x)$ will, resulting in a cost of $\mathcal{O}(mn + p(n^2 + m \log m))$. Again, if the matrices are not Toeplitz, the cost will simplify to $\mathcal{O}(pmn)$. If p is small, calculating multiple interpolations can best be done independently. However, if $p > n$, it is possible to rewrite Algorithm 4 to improve its efficiency by solving for the unit vectors first and then assembling the final interpolations.

Algorithm 5 Calculating stochastic interpolation of multiple functions simultaneously.

Require: The vectors x and $\{f_i\}_{i=1}^p = \{f_i(x)\}_{i=1}^p$ of size n

1: Scale x so that $x_1 = 0$ and $x_n = 1$, by shifting a and multiplying by factor b.
2: Initialize A_{nn} and X_0 using (2.3) and (4.22), where the y_k are initialized using (4.32).
3: Initialize the n unit vectors $e_1, e_2, \ldots, e_n$.
4: **for** $j = 1$ to n **do**
5: $\quad x_{k,j} = \textbf{PSJM\_wIEC}(A_{nn}, X_0, e_j, 5, 3)$
6: **end for**
7: Initialize A_{mn} using (4.21), where the y_k are initialized using (4.32).
8: **for** $i = 1$ to p **do**
9: $\quad x_i = \sum_{j=1}^n f_i(x_j) x_{k,j}$
10: $\quad f_i' = A_m n x_i$
11: **end for**
12: $x' = b/(n-1) + a$ for $i = 1 \ldots m$
13: **return** $\{x', \{f_i'\}\}$

Lemma 6.3.1. *Given input consisting of p sets of n evenly spaced data points* $\{x, f(x)\}_{i=1}^p$, *and a desired output of m evenly spaced output points* $\{x', f'(x')\}_{i=1}^p$ *where* $p > n$, $x_1 = x_1'$,

$x_n = x'_m$, and $f(x_i) = f'(x_i)$ for all input x_i, Algorithm 5 requires $\mathcal{O}(mn + p(n^2 + m\log m))$ operations.

Proof. As in the proof for Lemma 6.2.1, we assume that $m > n$. Again, the initialization of x and the y_k require $\mathcal{O}(n)$ operations, while the initialization of x' will require $\mathcal{O}(m)$ operations. Similarly, initializing the matrices A_{nn} and X_0 will require $\mathcal{O}(n^2)$ operations, initializing A_{mn} requires $\mathcal{O}(mn)$ operations, and initializing the n unit vectors will require $\mathcal{O}(n^2)$ operations.

Now consider line 5, which executes n times. Each iteration of the loop requires $\mathcal{O}(n\log n)$ operations by Lemma 5.2.4, and the entire loop requires $\mathcal{O}(n^2\log n)$ operations. Consider also the loop in lines 8 to 11; this loop will iterate p times, and on each iteration will perform a sum over n vectors, requiring $\mathcal{O}(n^2)$ operations and a matrix vector operation requiring $\mathcal{O}(m\log m)$ operations. Hence the loop will require $\mathcal{O}(p(n^2 + m\log m))$ operations.

Given that we only know that $p > n$ and $m > n$, the total number of operations will be dominated either by the initialization of A_{mn}, and hence be $\mathcal{O}(mn)$, or by the calculation of the x_i and f'_i. Thus the total number of operations required is $\mathcal{O}(mn + p(n^2 + m\log m))$. □

And as, after Lemma 6.2.1, we use a non-Toeplitz matrix for A_{nn}, X_0, and A_{mn}, the complexity simplifies to $\mathcal{O}(pmn)$, since the matrix vector multiplication will require $\mathcal{O}(mn)$ operations, which is the bottleneck.

6.4 Using the Laplace Probability Distribution Function

The Gaussian probability distribution function is not the only centrally symmetric pdf we can use; another easily implemented pdf is the Laplace pdf, which is also centrally symmetric but, as we see in Fig. 6.3, much sharper. Loosely speaking, this is important because stochastic interpolation includes around each point x_i those neighbors whose pdf is greater than machine zero for the pdf centred at x_i. Thus the sharper the pdf, the fewer points are considered, and the wider the pdf, the more neighbors we take into consideration at every point. To use the Laplace probability distribution function we must modify the definition of A_{nn} from (4.21) so that

$$a_{jk} = \Psi(y_{k+1}, x_j, \alpha) - \Psi(y_k, x_j, \alpha) \tag{6.1}$$

where

$$\Psi(y, x, \alpha) = \begin{cases} \frac{1}{2}e^{(y-x)/\alpha} & \text{if } y < x \\ 1 - \frac{1}{2}e^{(x-y)/\alpha} & \text{if } y \geq x \end{cases}$$

which is the cumulative distribution function for the Laplace pdf. If $\alpha \ll 1$, the resultant

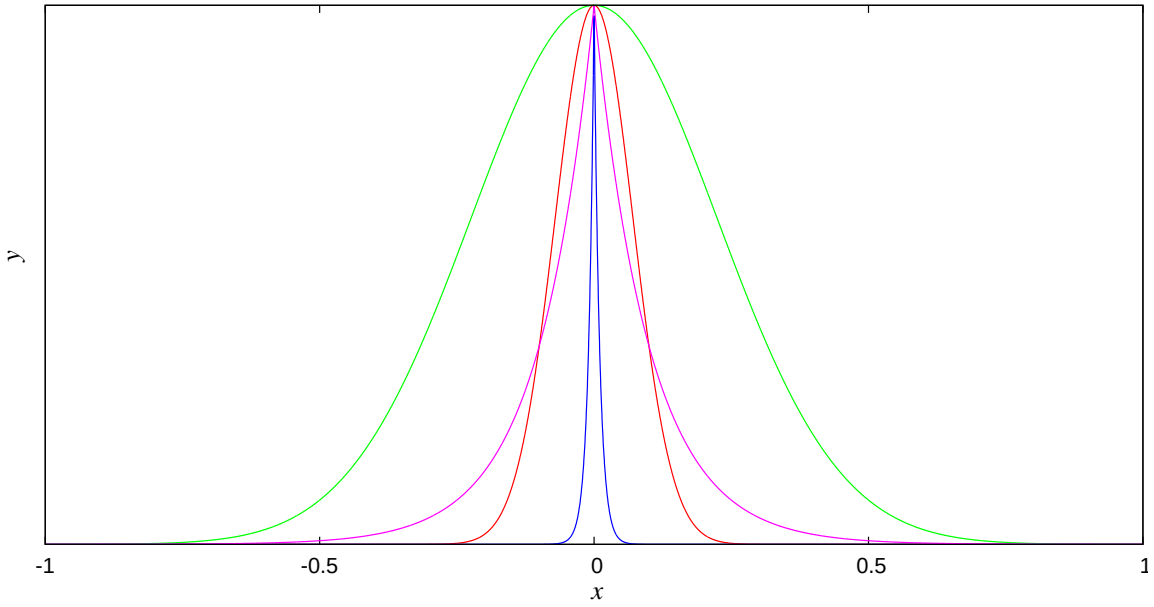


Figure 6.3: The Gaussian and Laplacian probability functions, with $\alpha_G = 0.01$ (red), $\alpha_G = 0.10$ (green), $\alpha_L = 0.01$ (blue) and $\alpha_L = 0.10$ (violet).

matrix A_{nn} will be diagonally dominant and row stochastic, and near-Toeplitz but not fully Toeplitz. Unlike the Gaussian matrix, there is no easy resetting of the endpoints y_0 and y_n to guarantee a Toeplitz matrix, so generally speaking using the Laplace matrix we will require $\mathcal{O}(n^2)$ operations for both matrix vector multiplications and as the cost of using PSJM, with and without iterative correction. Since the Laplace probability distribution function is so sharp, however, the fixed precision matrix will be diagonally dominant and will contain only a limited number of sub- and super diagonals, depending on the value of α_L. Thus, the complexity of the matrix vector multiplication can be reduced to $\mathcal{O}(tn)$, where t is the maximum number of non-zero values on a single row. Also, since the matrix is diagonally dominant, we can use as initial guess X_0 the scaling matrix whose diagonal is the reciprocal of the diagonals of A_{nn}, as in Theorem 4.2.1.

6.5 Blending Interpolations

It is possible to blend the Gaussian and Laplacian matrices, so that if A_G is the Gaussian matrix with $\alpha = \alpha_G$, and A_L the Laplacian matrix with $\alpha = \alpha_L$, then $A_B = (gA_G + lA_l)$, assuming that $g + l = 1$. This blended interpolation allows for three different parameters given the same set of data points which affect the interpolation: α_g will affect the Gaussian interpolation, α_l the Laplacian interpolation, and g and $1 - g$ will affect the weighting of

the two matrices. Therefore, for the same value of α, the Gaussian will include more neighbors, weighted more heavily, than the Laplacian will consider, and the blending allows us to emphasize closest neighbors when interpolating. We see the results of blending and modifying the α values in Figures 6.4 through 6.9, which illustrate using a simple single variable how the varying α_G and α_L affect the interpolation as both matrices are used equally.

First we consider the interpolation of the step function $f(x) = \lfloor x \rfloor$; in Fig. 6.4 we use a 50/50 blend of the Gaussian and Laplacian matrix, with equal values for $\alpha = 0.2$. Similar values of α reveal similar curves, although the Laplace curve is measurably flatter than the Gaussian curve. Regardless of blending, however, the interpolating functions do not closely follow the curve. Changing the Laplacian matrix so that $\alpha_L = 0.01$ results in Fig. 6.5 results in an interpolation that looks like a step function, but is somewhat offset, since the steep increases occur at the midpoint between data points. This latter deficit can be overcome by using more datapoints, as in Fig. 6.6. Thus the Laplace matrix allows us to capture the steep increases with fewer data points, whereas the Gaussian matrix results in a smooth curving function.

In terms of the number of operations required though, whereas the Gaussian matrix is Toeplitz and requires $\mathcal{O}(n \log n)$ operations for each matrix vector multiplication, and the Laplace matrix is sparse and requires only $\mathcal{O}(tn)$ operations where t is the maximum number of non-zero values in a single row, the blended matrix is neither Toeplitz nor sparse, and hence requires $\mathcal{O}(n^2)$ operations.

Obviously, interpolating a step function can be done best using the Laplace probability distribution function, but not every function is a step function. Next we consider the interpolation of the Runge function, which is a very smoothly curved function. In Fig. 6.7 both the Gaussian and Laplacian curves suffer from a lack of data points; adding even two more data points results in Fig. 6.8, which is a much better interpolation. We note that any set of data points which does not include $(0, f(0))$ results in an interpolation that does not reach the extremum at $(0, f(0)) = (0, 1)$, but it is here in Fig. 6.9 that we see that the Gaussian matrix captures more of the extremum, illustrating that the Gaussian matrix succeeds better at approximating extremum that are not included in the data set.

We conclude that the Laplacian and Gaussian matrix each have their use, but recognize that both the step and the Runge function are carefully contrived test cases. The data in Fig. 6.1 is a snapshot of one line of a grayscale image. where each data point represents one pixel, and $f(x)$ is the shading of that pixel, and the interpolation in that figure uses only a Gaussian matrix. In Fig. 6.10 which uses the same source data, we can see the effect of adding the Laplacian matrix and blending the two functions using exaggerated values for the mollifiers to emphasize the effect, so that here we use $\alpha_G = 0.50$ and $\alpha_L = 0.01$. We

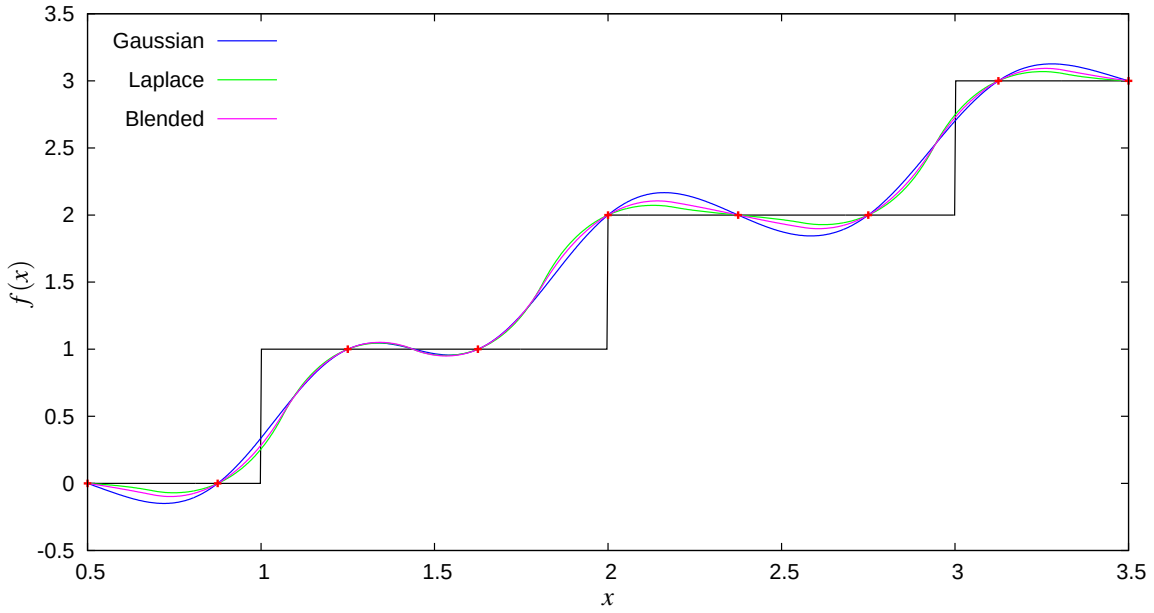


Figure 6.4: Interpolation of $f(x) = \lfloor x \rfloor$ based on a blended Gaussian/Laplacian, utilizing 50% of the Gaussian matrix and 50% of the Laplacian matrix with $\alpha = 0.2$ for the Gaussian matrix, and $\alpha = 0.2$ for the Laplacian, and using 9 datapoints.

can see that the Laplacian matrix is helpful in flattening out the curves of the Gaussian in the flat regions, and sharpening the points.

6.6 Interpolating Images

Having considered single and multiple variable functions, we now turn our attention to images. A grayscale $n \times m$ pixel image is stored as an $n \times m$ array of integers, each holding a value between 0 and 255, where 0 indicates black and 255 indicates white, and a colour image stored in RGB consists of 3 different $n \times m$ arrays, each representing one of the colour channels and containing values between 0 and 255. We can consider each row and column of these matrices as a function, and interpolate between them to create enlarged images. For instance, interpolating each row of the image would result in an image that was stretched horizontally, as the number of horizontal points increased. Similarly, interpolating each row would stretch the image vertically as the number of rows increased. Trying to stretch both the rows and columns of the same image simultaneously would result in an image with many pixels missing. As such, it is important to stretch in one direction first, and then use this modified image as the source to stretch in the other direction, resulting ultimately in an expanded image.

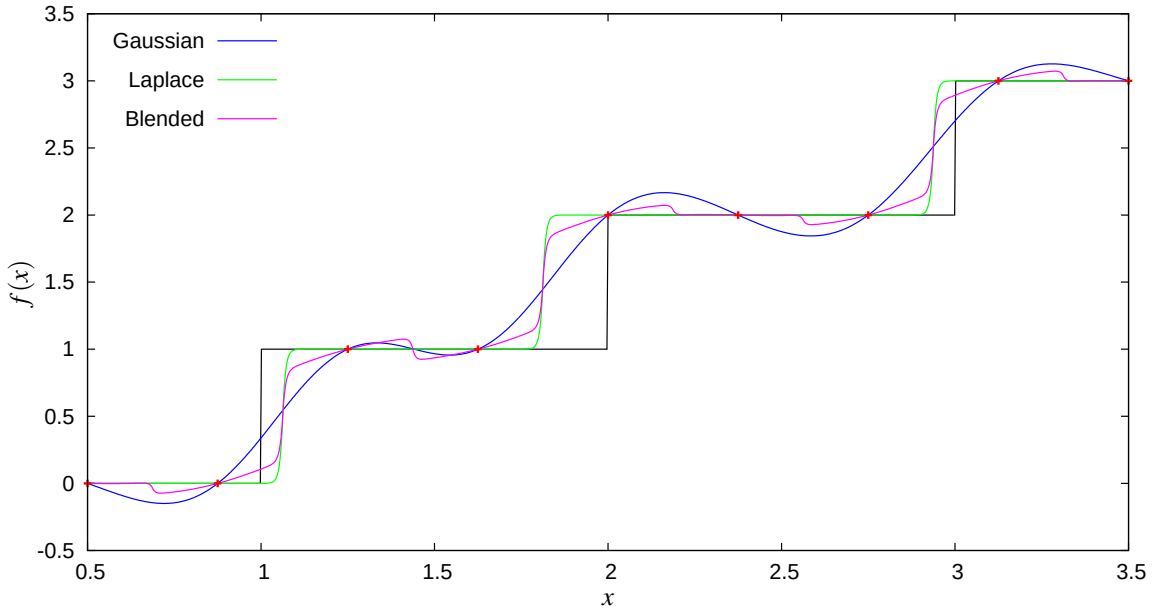


Figure 6.5: Interpolation of $f(x) = \lfloor x \rfloor$ based on a blended Gaussian/Laplacian, utilizing 50% of the Gaussian matrix and 50% of the Laplacian matrix with $\alpha = 0.2$ for the Gaussian matrix, and $\alpha = 0.01$ for the Laplacian, and using 9 datapoints.

Consider the different ways in which we can modify this expansion. First, naturally, we can vary the factor to which we expand the image, whether it be doubling in size, tripling in size, or growing even larger. The advantage of using interpolation is that we can add more than one interstitial pixel at a time. Secondly, we can vary the values of the mollifiers, and the relative weights of the Gaussian and Laplacian matrices, depending on the source picture. To create the expanded image, we use Algorithm 6, which is itself an extension of Algorithm 5, and interpolates the image by first expanding each row, resulting in an image with the same number of rows but many more columns, followed by expanding each column. Thus an $n \times m$ image becomes an $zn \times zm$ image, where z is the expansion, or zoom, factor. We also note that, given the structure of an RGB based colour image, we can expand a colour image by expanding each of the three arrays representing the red, green, and blue channels separately. We will see an example of colour expansion in Fig. 6.18. Hence, expanding the colour image will require only three times as many operations as expanding the grayscale image, and to analyse the complexity we need only consider the grayscale image. In analysing the number of operations required for Algorithm 6, we will consider the three parameters z, n and m, where z is the expansion factor, and n and m are the dimensions of the original picture. We note, however, that z is usually a fairly small integer.

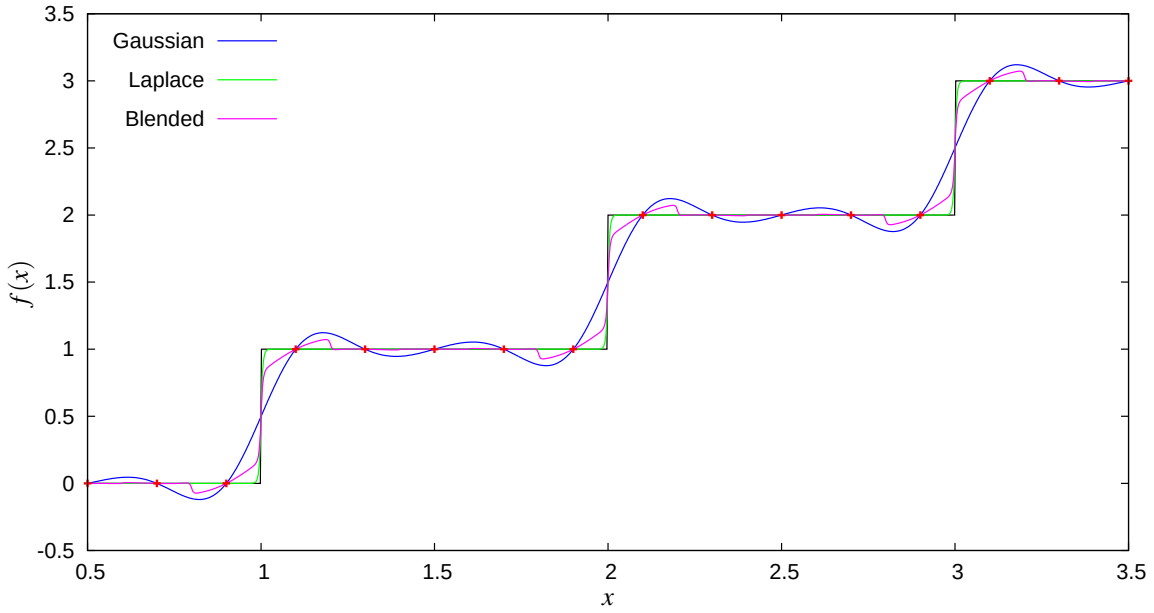


Figure 6.6: Interpolation of $f(x) = \lfloor x \rfloor$ based on a blended Gaussian/Laplacian, utilizing 50% of the Gaussian matrix and 50% of the Laplacian matrix with $\alpha = 0.2$ for the Gaussian matrix, and $\alpha = 0.01$ for the Laplacian, and using 16 datapoints.

Lemma 6.6.1. *Given an $n \times m$ grayscale image and an integer zoom factor z, Algorithm 6 will calculate the expanded $zn \times zm$ image using $\mathcal{O}(zm^3 + znm^2)$ operations.*

Proof. We assume without loss of generality that $n \leq m$. We also assume that the various matrices are not necessarily Toeplitz; if A_{pq} and X_0 are Toeplitz we can speed up the matrix vector multiplications, but the loops requiring the matrix vector multiplications will still be dominated by the vector addition, and adding together p vectors of length q will require $\mathcal{O}(pq)$ operations. Also, initializing the various A_{pq} and X_0 will require $\mathcal{O}(pq)$ operations, so that initializing the unit vectors and the matrices will require $\mathcal{O}(zm^2)$ operations, since the largest marix created will be $A_{zm,m}$.

Let us first consider the number of operations required to expand the number of rows from n to zn, which is completed in steps 3 through 13. Examining it closely, we observe that this is an iteration of Algorithm 5 but that, since we are using a blended matrix which is non-Toeplitz, the cost of each iteration of step 6 will be $\mathcal{O}(n^2)$, resulting in a cost for that loop of $\mathcal{O}(n^3)$. The loop from step 9 to step 12 will iterate m times, and each iteration will require the summation of n vectors of length n, and the multiplication of an $zn \times n$ matrix and an n element vector. The former will require $\mathcal{O}(n^2)$ operations, while the latter will require $\mathcal{O}(zn^2)$ operations, and thus the entire loop will require $\mathcal{O}(zn^2m)$ operations, and at

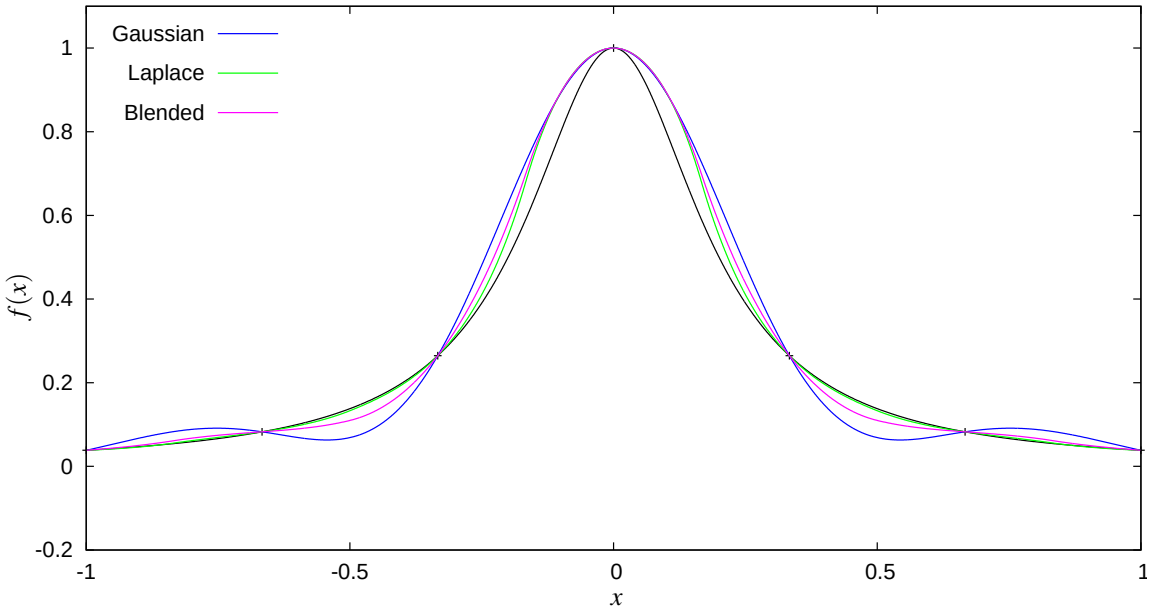


Figure 6.7: Interpolation of the Runge function $f(x) = 1/(1+25x^2)$ based on a blended Gaussian/Laplacian, utilizing 50% of the Gaussian matrix and 50% of the Laplacian matrix with $\alpha = 0.2$ for both matrices, using 7 data points.

step 13 the image will now be an $zn \times m$ pixel image.

It remains only to expand number of columns in turn, in steps 14 through 23. The cost of each iteration of step 17 will be $\mathcal{O}(m^2)$, resulting in a cost for that loop of $\mathcal{O}(m^3)$. The loop from step 20 to 23 will iterate zn times, and each iteration will require the summation of m vectors of length m, and the multiplication of an $zm \times m$ matrix and an m element vector. The former will require $\mathcal{O}(m^2)$ operations and the latter will require $\mathcal{O}(zn(zm^2)) = \mathcal{O}(z^2nm^2)$.

We can conclude that the overall number of operations is bounded by $\mathcal{O}(zm^3 + z^2nm^2)$; if $zn < m$ then this simplifies to $\mathcal{O}(zm^3)$, and otherwise it simplifies to $\mathcal{O}(z^2nm^2)$. □

To illustrate this algorithm we zoom in on a small 50×50 pixel patch of Fig. 6.11, and using a zoom factor of 4, which results in a 200×200 pixel image. The results of Algorithm 6 depend on the values used for the mollifiers α_G and α_L, and it is also possible to modify the $\alpha_{G'}$ used for steps 11 and 22. Although the matrix in these steps is blended, modifying the value of $\alpha_{L'}$ had no visible effect. Figures 6.13 through 6.16 show the effects of modifying the various mollifiers; the blended weighting is exaggerated so that 99% of the result is due to the relevant mollifier, and in each figure the original image is included. To make the comparison easier, the original image is scaled so that each original pixel is now a 4×4 block. We can see from Fig. 6.13, where the value of α_L is varied, that using a very

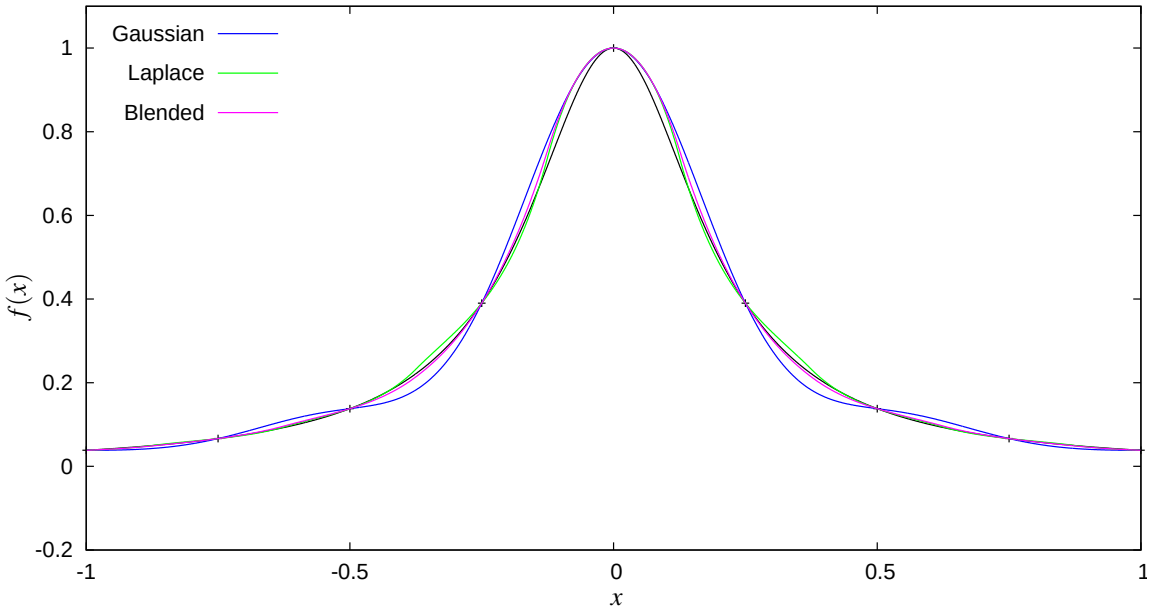


Figure 6.8: Interpolation of the Runge function $f(x) = 1/(1+25x^2)$ based on a blended Gaussian/Laplacian, utilizing 50% of the Gaussian matrix and 50% of the Laplacian matrix with $\alpha = 0.2$ for both matrices, using 9 data points.

small $\alpha_L = 0.05$ results in a heavily pixelated image, but that for $\alpha_L = 0.20$ the edges are reasonably distinct, an effect that would be enhanced if we utilized more of the Gaussian matrix. Varying α_G, as we do in Fig. 6.14, we see that using a very small α_G values leads to excessive blurriness, where edges are almost entirely lost. On the other hand, when $\alpha_G \neq \alpha_{G'}$, and especially when the difference is pronounced, the result is an interesting moire effect. Fig. 6.15 shows the effect of modifying $\alpha_{G'}$; there is again a moire effect when $\alpha_{G'} \ll \alpha_G$, but when $\alpha_{G'} \gg \alpha_G$, the result is a blurred, smoothed image. In contrast, Fig. 6.16 illustrates the effect of using a far less reasonable value for α_G, resulting in a strong checker board effect.

6.6.1 Timing

The interpolations in this chapter were implemented in Matlab, using the code Appendix A. Random square snapshots of the image in Fig. 6.11 were used for the timing tests, ranging in size from 50×50 to 500×500 pixels, and expanding them with expansion factors of 2, 3, 4, and 8. At each size and expansion factor 10 random snapshots were used, and the time recorded and averaged. The results, as we can see in Fig. 6.12, exceed our expectations, as the best fit power regression $f(x) = cn^t$ results in values for t of $1.86 < t < 1.9$ for each

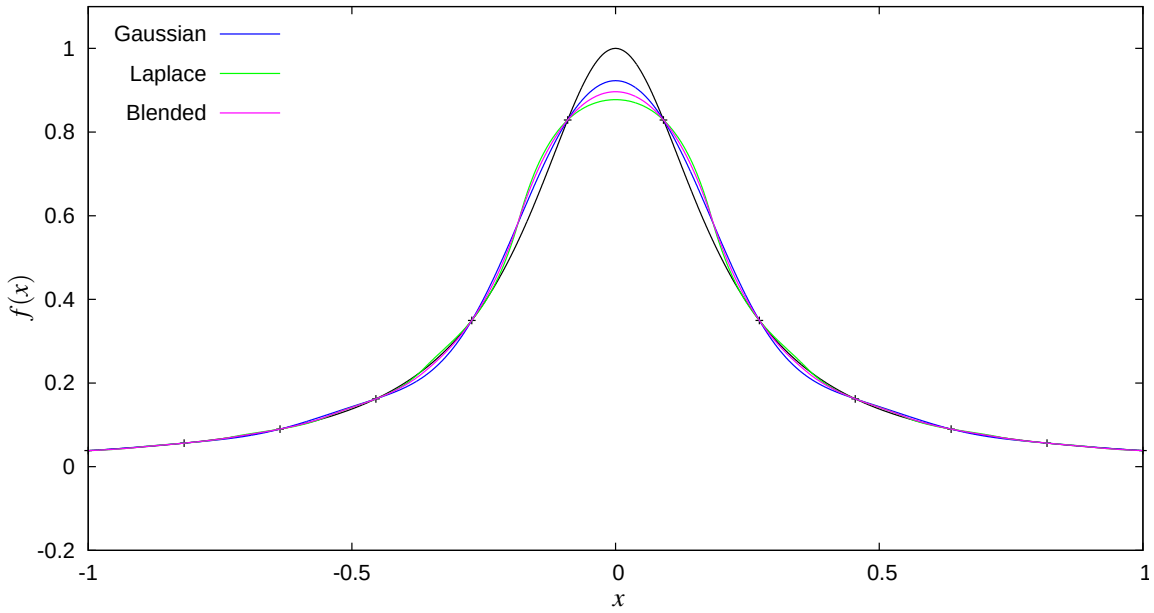


Figure 6.9: Interpolation of the Runge function $f(x) = 1/(1+25x^2)$ based on a blended Gaussian/Laplacian, utilizing 50% of the Gaussian matrix and 50% of the Laplacian matrix with $\alpha = 0.2$ for both matrices, using 12 data points.

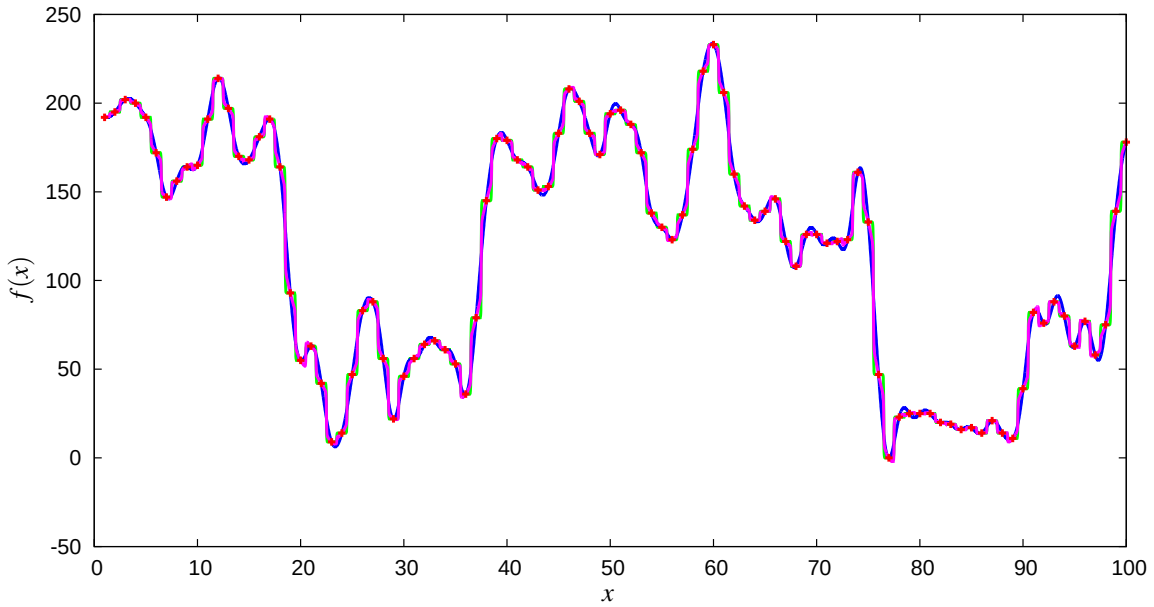


Figure 6.10: Interpolating a one-dimensional set of 100 data points $\{x, ff(x)\}$ (red), which consist of a snapshot of one row of an image. The interpolation uses a Gaussian matrix with $\alpha = 0.50$ (blue), a Laplacian matrix with $\alpha = 0.01$, (green) and a blended Gaussian/Laplacian matrix using 50% of each (violet), all interpolated over 1000 output points.

Algorithm 6 Expanding an image using blended stochastic interpolation.

Require: Image Ξ and integer zoom factor z

Require: The convolution and deconvolution factors α_G, α_L, and $\alpha_{G'}$ and the percentage p_G representing the degree to which the Gaussian matrix is used. ($p_L = 1 - p_G$)

Let r, c be the dimensions of Ξ.
▷ Zoom the image Ξ in 2 steps: first increase the number of rows, and then the number of columns
Initialize $A_{r,r}$ and X_0 using a blend of Gaussian and Laplacian matrices.
Initialize the r unit vectors $e_1, e_2, \ldots, e_r$.
for $j = 1$ to r **do**
    $x_{k,j} = \textbf{PSJM\_wIEC}(A_{r,r}, X_0, e_j, 5, 3)$
end for
Initialize $A_{zr,r}$ using a blend of Gaussian and Laplacian matrices.
for $i = 1$ to c **do**
    $x_i = \sum_{j=1}^{r} \Xi(i,j)(x_j) x_{k,j}$
    $\Xi'(:,i) = A_{zr,r} x_i$
end for ▷ Ξ' now has zr rows, but still c columns.
$\Xi = \Xi'$
Initialize $A_{c,c}$ and X_0 using a blend of Gaussian and Laplacian matrices.
Initialize the r unit vectors $e_1, e_2, \ldots, e_r$.
for $j = 1$ to c **do**
    $x_{k,j} = \textbf{PSJM\_wIEC}(A_{c,c}, X_0, e_j, 5, 3)$
end for
Initialize $A_{zc,c}$ using a blend of Gaussian and Laplacian matrices.
for $i = 1$ to zr **do**
    $x_i = \sum_{j=1}^{c} \Xi(i,j)(x_j) x_{k,j}$
    $\Xi'(:,i) = A_{zc,c} x_i$
end for ▷ Ξ' is now an $zr \times zc$ image.
return Zoomed Image Ξ'

of the four expansions. These results are explained by the fact that Matlab's matrix vector operations are heavily optimized, and takes advantage of multiple cpus, when available, to parallelize the calculations. Since there are 8 cpus available, this provides a significant speed improvement. The C code, in Chapter 5, is not parallelized to take advantage of the multiple CPUs, and hence requires time more proportional as proxy to the number of expected operations.

6.6.2 Zooming In On An Eye - Expanding a Colour Image

Lastly we consider an image which contains many edges and definitions, in the form of a picture of an eye. The original image is 69×100 pixels in size. We first converted it to grayscale, using gimp, and expanded the image fourfold, yielding Fig. 6.17. The

Figure 6.11: Source image for the image interpolation; a grayscale image that is 1347×1422 pixels in size.

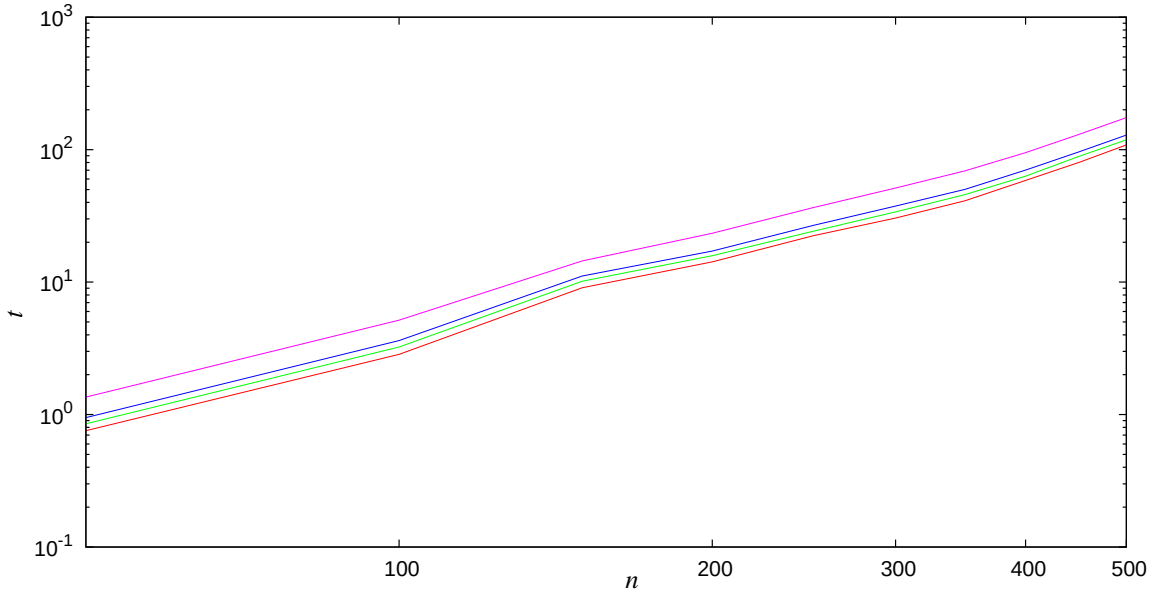


Figure 6.12: Average time required to interpolate an $n \times n$ image, over 10 iterations per size, of random snapshots from the image in Fig. 6.11. Images are expanded up to 2 (red), 3 (green), 4 (blue), or 8 (violet) times their original size.

scaled original image, for comparison, is drawn with each pixel expanded to a 4×4 block, illustrating the source data. We can see that rounded edges, such as the border of the iris

a) Scaled Original Image

b) $\alpha_L = 0.05$

c) $\alpha_L = 0.20$

d) $\alpha_L = 0.40$

Figure 6.13: Zooming in on a 50×50 region of Fig. 6.11 to create a 200×200 pixel image, using blended interpolation utilizing 1% of the Gaussian and 99% of the Laplacian matrices. The convolution factor $\alpha_G = 0.20$ and deconvolution factor $\alpha_{G'} = 0.20$ are constant, but the convolution factor α_L varies.

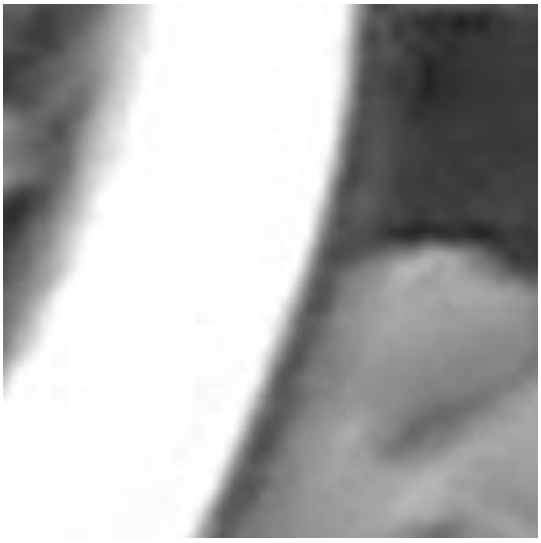
and the edges of the eye lids, expand nicely. However, the sharp lines of the eyelashes and eyebrow hair do not sharpen up as well; the source data is too pixelated. After expanding the grayscale version, we expanded the original RGB colour image by expanding each of

a) Scaled Original Image

b) $\alpha_G = 0.05$

c) $\alpha_G = 0.20$

d) $\alpha_G = 0.50$



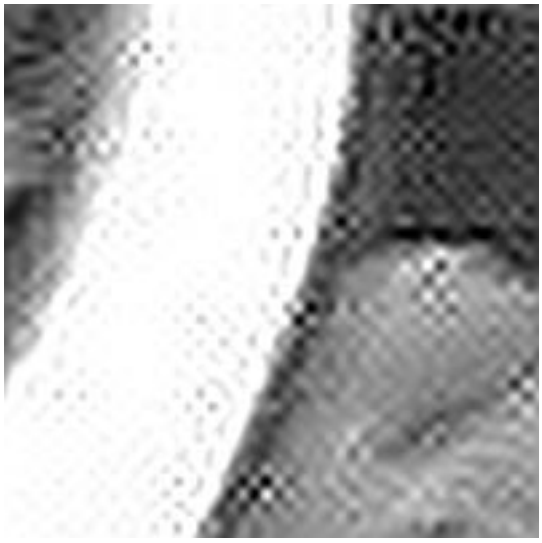
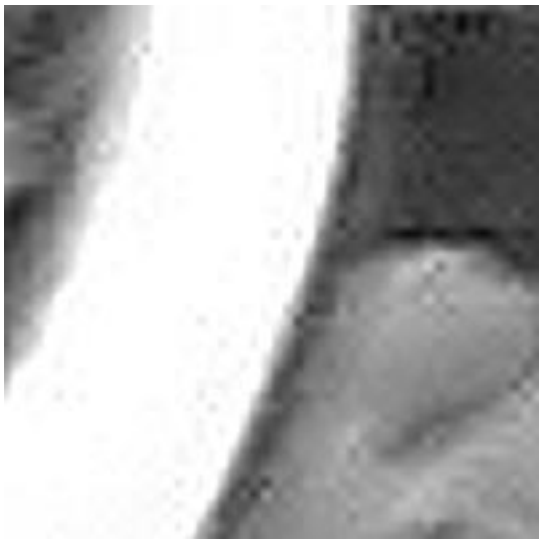


Figure 6.14: Zooming in on a 50×50 region of Fig. 6.11 to create a 200×200 pixel image, using blended interpolation utilizing 99% of the Gaussian and 1% of the Laplacian matrices. The convolution factor $\alpha_L = 0.10$ and deconvolution factor $\alpha_{G'} = 0.20$ are constant, but the convolution factor α_G varies.

the red, green, and blue image sub-arrays separately, as can be seen in Fig. 6.18. Again rounded edges expand nicely, but there are the same issues with the hairs, although they are easier to see because with colour the contrast between hair and skin is greater.

a) Scaled Original Image

b) $\alpha_{G'} = 0.05$



c) $\alpha_{G'} = 0.20$

d) $\alpha_{G'} = 0.50$

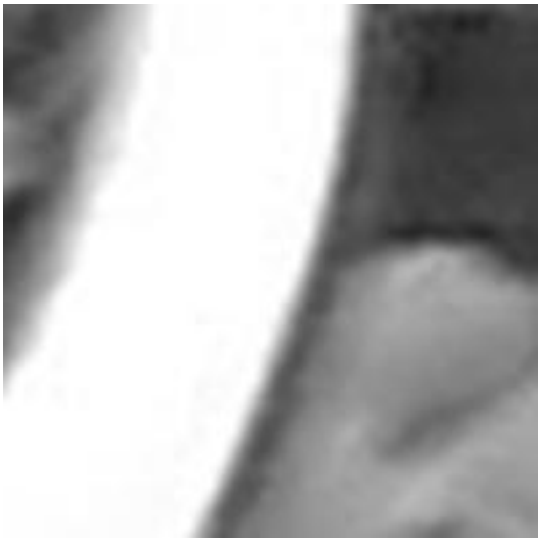


Figure 6.15: Zooming in on a 50×50 region of Fig. 6.11 to create a 200×200 pixel image, using blended interpolation utilizing 99% of both Gaussian and 1% of the Laplacian matrices. The convolution factors $\alpha_G = 0.20$ and $\alpha_L = 0.10$ are constant, but the deconvolution factor $\alpha_{G'}$ varies.

6.6.3 Summary

PSJM is applied to the problem of stochastic interpolation, with emphasis on the interpolation, or expansion, of images. Stochastic interpolation relies on the use of a probability

a) Scaled Original Image

b) $\alpha_{G'} = 0.05$

c) $\alpha_{G'} = 0.20$

d) $\alpha_{G'} = 0.50$

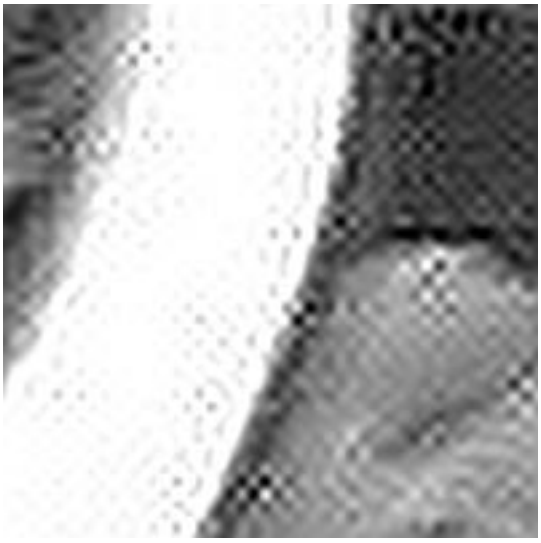


Figure 6.16: Zooming in on a 50×50 region of Fig. 6.11 to create a 200×200 pixel image, using blended interpolation utilizing 99% of both Gaussian and 1% of the Laplacian matrices. The convolution factors $\alpha_G = 0.50$ and $\alpha_L = 0.10$ are constant, but the deconvolution factor $\alpha_{G'}$ varies.

distribution function (pdf) to weight the neighbors of each data point, centering the pdf at that data point and weighting each neighbor with a probability based on their distance from the data point. While SI is originally introduced using the Gaussian pdf, the use of

the Laplace pdf is also examined. Like the Gaussian pdf, the Laplace pdf is a centrally symmetric pdf, but unlike the Gaussian, the Laplace pdf produces far sharper interpolants so that in essence fewer neighbors are considered and weighted more heavily. Both pdfs produce row stochastic matrices, but given the same mollifier value, the representation of these will result in a far greater number of zeroes due to underflows to zero for the matrix generated using the Laplace pdf than the one generated using the Gaussian pdf.

The advantage of using both pdfs in the same interpolation lies in the ability to blend these matrices, and to modify the mollifier of each matrix separately, expanding the number of possible adjustable parameters. However, the resulting blended matrix is no longer Toeplitz, and so our solver PSJM will require $\mathcal{O}(n^2)$ rather than $\mathcal{O}(n \log n)$ operations. Since the resulting matrix is, for good choices of α, a diagonally dominant matrix, we can take advantage of Theorem 4.2.1 to generate a good initial guess X_0.

Scaled Original Image

Interpolated Image

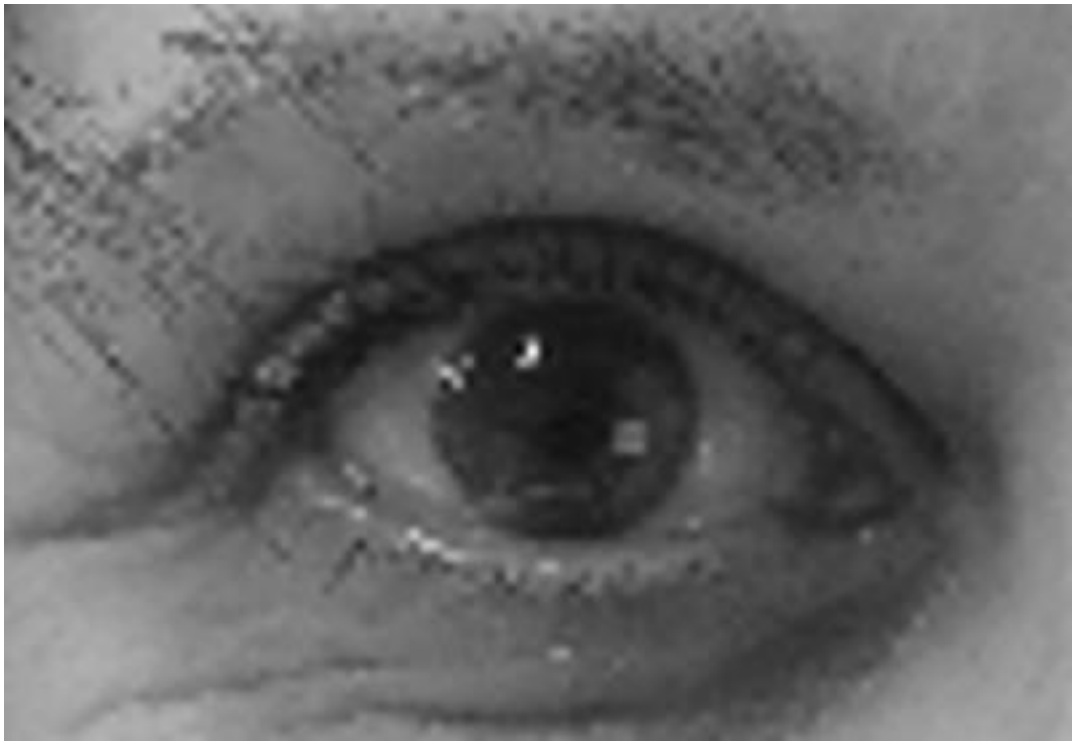


Figure 6.17: The original 69×100 pixel image of an eye is interpolated, creating an 276×400 pixel image. This interpolation uses 75% of the Gaussian matrix, 25% of the Laplacian matrix, and factors $\alpha_G = \alpha_{G'} = 0.2$ and $\alpha_L = 0.10$.

Scaled Original Image

Interpolated Image

Figure 6.18: The original 69×100 pixel image of an eye is interpolated, creating an 276×400 pixel image. This interpolation uses 75% of the Gaussian matrix, 25% of the Laplacian matrix, and factors $\alpha_G = \alpha_{G'} = 0.2$ and $\alpha_L = 0.10$, and interpolated each of the three RGB channels separately.

Chapter 7
Conclusion and Future Work

7.1 Conclusion

In this dissertation we were motivated by the desire to interpolate large images to study the problems of calculating A^{-1}, given a structured, dense matrix A, and solving for x, given $Ax = b$, using fixed precision arithmetic. Specifically, our goal was to produce a more efficient solver for stochastic interpolation problems.

We studied the Shulz-Jones-Mayer (SJM) algorithm, which converges to A^{-1} given a suitable initial guess X_0, and developed suitable choices for X_0 for a variety of classes. We modified the algorithm and improved its efficiency to create the Polynomial Shulz-Jones-Mayer (PSJM) algorithm, and applied iterative error correction to overcome the errors induced by using fixed precision arithmetic. Lastly, we applied the algorithm in implementing Stochastic Interpolation (SI), demonstrating the ability to interpolate single variable and multivariable functions and further demonstrating that SI techniques can be used to expand images in both gray-scale and RGB color.

7.1.1 The SJM algorithm

The first method, SJM, is very simple to state and converges quadratically, but is sadly not terribly efficient because it is hampered by the need to repeatedly multiply matrices which is generally an expensive operation, requiring $\mathcal{O}(n^3)$ operations for an $n \times n$ matrix. We study SJM in terms of the rate of convergence and the developing of a litmus test for possible initial guesses. Moreover, we apply that litmus test to prove that our proposed initial guesses for a variety of matrix classes will lead to convergence assuming perfect precision. Specifically we study diagonally dominant matrices, including matrices with that will be diagonally dominant if the columns or rows are permuted, tridiagonally dominant matrices where the three main diagonals dominate the matrix, and the Gaussian matrices used by stochastic interpolation.

7.1.2 The PSJM algorithm

Often solving $Ax = b$ by first calculating A^{-1} and then calculating $x = A^{-1}b$ is inefficient and prone to extra computational errors, requiring more work than actually necessary.

The improved method, PSJM, takes advantage of the iterative structure of SJM to create a polynomial whose variable is (X_0A). By careful ordering of the evaluation of the polynomial, it is possible to avoid the matrix matrix multiplications that hamstring SJM and replace them with matrix vector multiplications. The latter are much more efficient, requiring only $\mathcal{O}(n^2)$ operations in the general case, or $\mathcal{O}(n\log n)$ or even $\mathcal{O}(n)$ in certain cases.

Since PSJM calculates using only the initial matrices A and X_0 and the initial vector b to calculate x_k, where SJM calculates and then uses intermediate X_i, it is possible to bound the error when using PSJM. Careful consideration shows that the error depends on n, the size of the matrix, the norms of both A and X_0, and the norm of b. Of these parameters, the vector b is amenable to manipulation which allows iterative error correction by successively applying PSJM to problems with shrinking b vectors and combining the results. The iterative error correction is particularly useful; where SJM can continue iterating using fixed point precision, PSJM is typically limited to a small number of iterations because the coefficients of the polynomial outstrip the ability of the fixed precision arithmetic to store the coefficients without rounding. Hence the iterative error correction allows us to reset the iterations and continue converging, resulting in an algorithm that produces double precision answers given double precision input. When considering the number of operations $\mathcal{O}(f(n))$ required by PSJM, with and without iterative error correction, $f(n)$ is essentially the cost of performing matrix vector multiplications, albeit with large constants. Thus this technique shines for large problems and matrices for which fast matrix vector multiplication methods are possible.

Other solvers for Toeplitz matrix problems require at least $\mathcal{O}(n\log^2 n)$ operations, when the structure of the Toeplitz matrix is restricted, or $\mathcal{O}(n^2)$ operations for the general case. In the case of PSJM, we have shown that the method converges quadratically, and that, for those sets of Toeplitz matrices where a good initial guess is known, only $\mathcal{O}(n\log n)$ operations are required. As such, PSJM is has a substantially lower operation count than the known Toeplitz solvers and as such, has contributed significantly if incrementally to the toolbox of solving Toeplitz matrix-based problems.

7.2 Future Work

One way to expand our solver PSJM is to expand the set of matrix classes for which valid initial guesses are known. The results for the tridiagonally dominant matrices suggest that we should consider other classes of matrices that can be decomposed so that $A = B + C$, where B has a known inverse and $\left\|B^{-1}C\right\| < 1$. In those cases, B^{-1} will be a valid initial guess.

When performing stochastic interpolation on data sets where n is large, the resulting Gaussian or Laplacian matrices become banded; only the main diagonal and a relatively small number of sub- and super diagonals are non-zero when using fixed precision arithmetic, due to the inability to represent numbers smaller than the fixed precision allows. In these cases, the analysis and the code should be modified to take advantage of the fact that the matrix vector multiplications require only $\mathcal{O}(tn)$ operations, where t is the maximum number of non-zero values on any row. This coincidentally explains the results in Chapter 6 where Matlab's timing results imply faster computation than the analysis of the required number of computations suggests, as Matlab optimizes these operations based on the structure and values stored in matrices and vectors. One consideration for future work would be to bound t in terms of the mollifier α and the spacing of the data points, since the size of t will vary inversely with the maximum distance between data points.

Appendix A

Matlab Code

A.1 Initializing Matrices

This first collection of functions creates both the matrix A_{mn} used for stochastic interpolation, and the initial guess X_0 used when calculating $x_k = X_k b$. GaussMatrix produces the row-stochastic centrosymmetric matrix A_{mn}, based on (4.21). The value of $\pm\infty$ is approximated as $\pm 10^{120}$.

```
function [Amn, X0]=GaussMatrix(n,m,alpha)
 %   Creates   an   mxn   matrix   based   on   Gaussian   interpolation
 %   as   well   as   initial   guess   X0
 %   alpha   is   used   as   a   mollifying   agent
s1=0:1/m:1;
s2=-1/(2*n):1/n:(2*n+1)/(2*n);
s2(1)=-10^120;
s2(n+2)=10^120;
Amn=zeros(m+1,n+1);
X0=zeros(m+1,n+1);
for i=1:m+1
    for j=1:n+1
        v1 = erf((s2(j+1)-s1(i))/(2*sqrt(alpha)/n));
        v2 = erf((s2(j)-s1(i))/(2*sqrt(alpha)/n));
        Amn(i,j) = (1/2)*(v1-v2);
        X0(i,j) = (1/2) * (1/v1-1/v2);
    end;
end;
```

The function GaussToepMatrix produces a Toeplitz matrix, created by setting $y_0 = -1/2n$ and $y_n = (2n+1)/2n$, rather than $\pm\infty$.

```
function [Amn, X0]=GaussToepMatrix(n,m,alpha)
 %   Creates   an   mxn   matrix   based   on   Gaussian   interpolation
 %   as   well   as   initial   guess   X0
 %   alpha   is   used   as   a   mollifying   agent
s1=0:1/m:1;
s2=-1/(2*n):1/n:(2*n+1)/(2*n);
Amn=zeros(m+1,n+1);
X0=zeros(m+1,n+1);
for i=1:m+1
    for j=1:n+1
        v1 = erf((s2(j+1)-s1(i))/(2*sqrt(alpha)/n));
        v2 = erf((s2(j)-s1(i))/(2*sqrt(alpha)/n));
        Amn(i,j) = (1/2)*(v1-v2);
        X0(i,j) = (1/2) * (1/v1-1/v2);
    end;
end;
```

The function laplace_cdf calculates the Laplace cumulative density function, and is used by LaplaceMatrix in generating the Laplacian A_{mn}, which is row stochastic and diagonally dominant if α is sufficiently small. Hence the initial guess X_0 is initialized so that all off-diagonal entries are 0 and $(X_0)_{ii} = 1/(A_{mn})_{ii}$.

```
function [cx] = laplace_cdf(x,y,b);
 %   x   is   the   center   of   the   distribution
 %   y   is   the   value   whose   probability   we ' re   calculating
 %   b   is   related   to   the   variance   ( variance   is   2 b ^2)
cx=0;
if (y < x)
    cx = (1/2) * exp((y-x)/b);
else
    cx= 1 - (1/2) * exp(-1*(y-x)/b);
end;

function [Amn, X0]=LaplaceMatrix(n,m,alpha)
 %   Creates   an   mxn   matrix   based   on   Laplace   interpolation
 %   as   well   as   initial   guess   X0
 %   alpha   is   used   as   a   mollifying   agent
s1=0:1/m:1;    %   Evenly   spaced   data   points
s2=-1/(2*n):1/n:(2*n+1)/(2*n);  %   midpoints   between   data   points
   s2(1)=-10^120;
   s2(n+2)=10^120;
 % s2 =-1/ n :1/( n ):1;
Amn=zeros(m+1,n+1);
X0=eye(m+1,n+1);

for i=1:m+1
    for j=1:n+1
        v1 =2* laplace_cdf(s1(i),s2(j+1),2*(alpha/n));
        v2 =2* laplace_cdf(s1(i),s2(j),2*(alpha/n));

        Amn(i,j) = 1/2*(v1-v2);
 %           X0 ( i , j )   =   1/ Amn ( i , j );
    end;
end;
X0=(1/(Amn(1,1)))*X0;
 % if   ( n == m )   X0 = inv ( Amn );   end ;
```

The blended matrix is created by creating a Gaussian matrix A_g and Laplacian matrix A_L, each with their respective initial guesses X_G and X_L. Then the blended matrix and initial guess are a weighted sum of the matrices and initial guesses, respectively.

```
function [A,X] = BlendedMatrix(n,m,...
                               alphag, alphal, ...
                               fracg, fracl);
 %  Creates  the  row  stochastic  matrix  A  and  initial  guess  X
 %  A  is  mxn  matrix ,  calculated  as
 %     A =  ( fracg * Ag + fracl * Al )/( fracg + fracl )
[Ag,Xg]=GaussMatrix(n,m,alphag);
[Al,Xl]=Laplace(n,m,alphal);
A=(fracg*Ag+fracl*Al)/(fracg+fracl);
X=(fracg*Xg+fracl*Xl)/(fracg+fracl);
```

A.2 Calculating PSJM

These three functions do the actual work of implementing the PSJM algorithm. The first calculates the coefficients required for a recursion at depth k, returning the coefficients as a vector $coeff$, where $coeff_i = \alpha_{k,i-1}$.

```
function [coeff] = calc_PSJM_coeff(depth)
 %  Calculates  the  coefficients  needed  by  PSJM  for  k  recursions
depth=depth+1;  %  fx  for  zero  vs  1  indexing
co=zeros(2^(depth),depth);
co(1,1)=1;
for j=2:depth
    for i=1:(2^(j))
        co(i,j)=2*co(i,j-1);
    end;
    for i=1:(2^(j))
        for k=1:2^(j)
            cc=co(i,j-1)*co(k,j-1);
            p=k+i;
            if cc~=0
                co(p,j)=co(p,j)-cc;
            end;
        end;
    end;
end;
coeff=co(1:2^(depth-1),depth);
```

This next function uses the initial matrix A_{nn}, initial guess X_0, and the vector b to calculate $x_k = X_k b = \approx A^{-1}b$. It calls the function `calc_PSJM_coeff` to calculate the coefficients; this is one place where the code can eventually be sped up by storing those coefficients explicitly rather than recalculating.

```
function [xk] = calc_PSJM(A,X,b,k)
 %  Calculates  PSJM  with  matrix  A ,  vector  b ,
 %     initial  guess  X ,  and  k  recursive  steps .
coeff=calc_PSJM_coeff(k);
x2=X*b;
```

```
xk=coeff(1)*x2;
for i=2:2^k
    x1=A*x2;
    x2=X*x1;
    xk=xk+coeff(i)*x2;
end;
```

This last function performs the iterative error correction described in Algorithm 2, by repeatedly calling `calc_PSJM` r times.

```
function [xk]=calc_PSJM_wIEC(A,X,b,k,r)
 %  Performs  PSJM  with  iterative  error  correction
b1=b;
xk=calc_PSJM(A,X,b,k);
for i=1:r
    b1=b-A*xk;
    xkk=calc_PSJM(A,X,b1,k);
    xk=xk+xkk;
end;
```

A.3 Interpolating an Image

The function `BlendInterpol` interpolates the image `OrigIm`, expanding it using a blended Gaussian and Laplacian stochastic interpolation. Note that the image is assumed to be a single two-dimensional array; if the image is in color – and stored as a three dimensional array or as three two-dimensional arrays – this function will be called three times, once for each color.

```
function [FinalIm] = BlendInterpol(OrigIm, grow,...
                                   alphag, alphal,...
                                   fracg, fracl,...
                                   alphagout);
 %  OrigIm  stores  the  original  image
 %  grow  stores  factor  by  which  we  are  increasing
 %  resolution .  Is  assumed  to  be  a  positive  integer .

OrigIm=double(OrigIm);
[rows, cols]=size(OrigIm);

 %  Expand  image  by  increasing  rows  by  a  factor  of  grow .
[Ann,X]=BlendedMatrix(rows-1,rows-1,...
                      alphag, alphal, fracg, fracl);
Emat=eye(rows,rows);
Xmat=zeros(rows,rows);
for i=1:rows
    Xmat(:,i)=calc_PSJM_wIEC(Ann,X,Emat(:,i),5,5);
end;
Amn=BlendedMatrix(rows-1,grow*rows-1,...
                  alphagout, alphal, fracg, fracl);
RowIm=zeros(grow*rows,cols);
for i=1:cols
   b=zeros(rows,1);
   for j=1:rows
```

```
        b=b+OrigIm(j,i)*Xmat(:,j);
    end;
    RowIm(:,i)=Amn*b;
end;

 %  Expand  image  again ,  this  time  columnwise
[Ann,X]=BlendedMatrix(cols-1,cols-1,...
                      alphag, alphal, fracg, fracl);
Emat=eye(cols,cols);
Xmat=zeros(cols,cols);
for i=1:cols
     Xmat(:,i)=calc_PSJM_wIEC(Ann,X,Emat(:,i),5,5);
end;

Amn=BlendedMatrix(cols-1,grow*cols-1,...
                  alphagout, alphal, fracg, fracl);
FinalIm=zeros(grow*rows,grow*cols);
for i=1:(grow*rows)
    b=zeros(cols,1);
    for j=1:cols
       b=b+RowIm(i,j)*Xmat(:,j);
    end;
    FinalIm(i,:)=transpose(Amn*b);
end;

FinalIm=uint8(FinalIm);
OrigIm=uint8(OrigIm);
```

BIBLIOGRAPHY

[1] A. L. Andrew. Solution of equations involving centrosymmetric matrices. *Technometrics*, 15(2):pp. 405–407, 1973.

[2] Jacob Benesty, M. Mohan Sondhi, and Yiteng (Arden) Huang. *Springer Handbook of Speech Processing*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[3] Albrecht Böttcher and Seigei M. Grudsky. *Spectral Properties of Banded Toeplitz Matrices*. SIAM, 2005.

[4] Richard L. Burden and J. Douglas Faires. *Numerical Analysis*. Thomson; Brooks/Cole, 8th edition, 2005.

[5] S. Chandrasekaran, M. Gu, X. Sun, J. Xia, and J. Zhu. A superfast algorithm for Toeplitz systems of linear equations. *SIAM J. Matrix Anal. Appl.*, 29(4):1247–1266, 2007.

[6] G. Codevico, G. Heinig, and M. Van Barel. A superfast solver for real symmetric Toeplitz systems using real trigonometric transformations. *Numerical Linear Algebra with Applications*, 12(8):699–713, 2005.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[8] R. Crandall and J. Klivinton. Fast matrix algebra on apple g4. Technical report, Apple Computer inc., 2004. `http://images.apple.com/acg/pdf/g4_matrix072804.pdf`.

[9] Richard E. Crandall. *Topics in advanced scientific computation*. Springer, 1996.

[10] F. Dietrich. *Robust Signal Processing for Wireless Communications*. Foundations in Signal Processing, Communications and Networking. Springer, 2010.

[11] Heike Fassbender and Khakim D. Ikramov. Computing matrix-vector products with centrosymmetric and centrohermitian matrices. *Linear Algebra and its Applications*, 364(0):235 – 241, 2003.

[12] Matteo Frigo and Steven G. Johnson. Fftw. `http://www.fftw.org/`. A C subroutine library for computing the discrete Fourier transform (DFT) in one or more dimensions, of arbitrary input size, and of both real and complex data.

[13] Gene H. Golub and Charles F. van Van Loan. Matrix computations. In *Johns Hopkins Studies in Mathematical Sciences*. The Johns Hopkins University Press, 3rd edition, October 1996.

[14] Robert Gray. *Toeplitz and Circulant Matrices; A Review*, volume 2 of *Foundations and Trends in Communications and Information Theory*. Now Publishers Inc, 2005.

[15] Gnu scientific library. `http://www.gnu.org/software/gsl/`. A numerical library for C and C++ programmers, released under the GNU General Public License.

[16] Georg Heinig and Karla Rost. Fast algorithms for Toeplitz and Hankel matrices. *Linear Algebra and its Applications*, February 2011.

[17] Joseph Kolibal and Daniel Howard. The Novel Stochastic Bernstein Method of Functional Approximation. In *Adaptive Hardware and Systems*, pages 97–100, 2006.

[18] Joseph Kolibal and Daniel Howard. Stochastic Interpolation: A Probabilistic View. In *Bio-inspired, Learning, and Intelligent Systems for Security*, pages 129–135, 2008.

[19] LAPACK - Linear Algebra PACKage. `http://www.netlib.org/lapack/`. A numerical library written in Fortran 77, designed to solve common linear algebra problems on higher-performance machines.

[20] G. G. Lorentz. *Bernstein polynomials*. Chelsea Publishing Co., New York, 2nd edition, 1986.

[21] Andrew Melman. Symmetric centrosymmetric matrix-vector multiplication. *Linear Algebra and its Applications*, 320(1-3):193 – 198, 2000.

[22] S. G. Mikhlin. *Error Analysis in Numerical Processes*. John Wiley & Sons, 1991. translation of: Fehler in numerischen Prozessen.

[23] Michael K. Ng and Jianyu Pan. Approximate inverse circulant-plus-diagonal preconditioners for Toeplitz-plus-Diagonal matrices. *SIAM J. Sci. Comput.*, 32(3):1442–1464, May 2010.

[24] Michael K. Ng, Hai-Wei Sun, and Xiao-Qing Jin. Recursive-based PCG methods for Toeplitz systems with nonnegative generating functions. *SIAM J. Sci. Comput.*, 24(5):1507–1529, May 2002.

[25] Victor Pan. Fast and efficient parallel inversion of Toeplitz and block Toeplitz matrices. In H. Dym, S. Goldberg, M.A. Kaashoek, and P. Lancaster, editors, *The Gohberg Anniversary Collection*, volume 40/41 of *Operator Theory: Advances and Applications*, pages 359–389. Birkhäuser Basel, 1989.

[26] J. Stoer, R. Bulirsch, R. Bartels, W. Gautschi, and C. Witzgall. *Introduction to Numerical Analysis*. Texts in Applied Mathematics. Springer, 2002.

[27] William F. Trench. An algorithm for the inversion of finite Toeplitz matrices. *J. Soc. Indust. Appl. Math.*, 12:515–522, 1964.

[28] R. A. Usmani. Inversion of Jacobi's tridiagonal matrix. *Comput. Math. Appl.*, 27(8):59–66, 1994.

[29] C. von Runge. Über empirische funktionene und die interpolation zwischen äquidistanten ordinaten. *Zeitschrift für Mathematik und Physik*, 46:224–243, 1901.

[30] You-Wei Wen, Wai-Ki Ching, and Michael Ng. Approximate inverse-free preconditioners for Toeplitz matrices. *Applied Mathematics and Computation*, 217(16):6856 – 6867, 2011.

[31] Shalhav Zohar. Toeplitz matrix inversion: The algoritm of W. F. Trench. *J. Assoc. Comput. Mach.*, 16:592–601, 1969.

[32] Shalhav Zohar. The solution of a Toeplitz set of linear equations. *J. Assoc. Comput. Mach.*, 21:272–276, 1974.