Accurate and Efficient Computations with Structured Matrices

by

Plamen Stefanov Koev

Grad. (University of Sofia) 1997

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Mathematics

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor James W. Demmel, Chair
Professor Beresford N. Parlett
Professor Laurent El Ghaoui

Spring 2002

The dissertation of Plamen Stefanov Koev is approved:

Chair Date

Date

Date

University of California at Berkeley

Spring 2002

Accurate and Efficient Computations with Structured Matrices

Abstract

Accurate and Efficient Computations with Structured Matrices

by

Plamen Stefanov Koev
Doctor of Philosophy in Mathematics

University of California at Berkeley

Professor James W. Demmel, Chair

Computing solutions to real life scientific or engineering problem is most often the cheapest, fastest, safest and sometimes the only way to attack a problem. The computational bottleneck for solving the mathematical model built upon the real problem is most often the solution of a linear system, an eigenvalue or a singular value problem. In many instances the matrices that are involved in the computation are structured and it is not uncommon for the desired solution to be too ill conditioned to be computed with the traditional algorithms.

Matrix and problem structure can be exploited to obtain faster and very accurate solutions in spite of ill conditioning as the following new results in this thesis show.

- We derive new $O(n^3)$ methods for computing the Singular Value Decomposition of Polynomial Vandermonde matrices and of M-Matrices to high relative accuracy, regardless of condition number.

- We present a new algorithm for fast evaluation of the Schur function, that is exponentially faster than the previous best result and is only linear in the number of variables.

- Using the above algorithm for computing the Schur function we obtained new $O(n^2)$ Björck-Pereyra-type algorithms for solving generalized Vandermonde linear systems. Those methods allow for in many instances solutions of generalized Vandermonde linear systems to be computed to the full machine precision independent of condition

numbers. We also explain the combinatorial properties which make the existence of those algorithms possible for any given class of matrices.

- Harry Diamond's theorem states that if f is a real valued strictly convex or concave monotone function, g is its inverse, and $h(x) = [g([f([x])])]$, where $[\cdot]$ is the operator rounding any result to n significant digits, then $h(h(h([x]))) = h(h([x]))$. We generalize this theorem by proving that it remains valid when f is not strictly convex or concave when the arithmetic is performed according to the IEEE standard for binary floating point arithmetic in standard rounding mode – to the nearest floating point number. We also prove that a stronger condition holds, namely $h(h([x])) = h([x])$ for the natural logarithm and the exponential functions.

- We present a new faster algorithm for map making and power spectrum estimation of the Cosmic Microwave Background (CMB) radiation. The analysis of the CMB is one of the most powerful tools in studying the early Universe. By exploiting the matrix structure in this problem we designed new $O(n^{3/2})$ algorithms for this analysis, where n is the number of sky pixels being observed. For comparison the traditional algorithms scale as $O(n^3)$ and will be inapplicable for the analysis of the upcoming PLANCK experiments which will deliver data from the observation of tens of millions of pixels.

- We present a new algorithm for evaluating the complete symmetric polynomial in the algebra of $n \times n$ rational matrices, which is exponentially faster than the traditional algorithm and use this algorithm to prove that in the above algebra all polynomial identities of degree $2n+2$ follow from the Standard Identity of degree $2n$ for $n = 4, 5$.

Professor James Demmel

Dissertation Committee Chair

Contents

List of Figures

List of Tables

Acknowledgements

First and foremost I would like to thank my thesis advisor James Demmel for his patience and tireless efforts as he guided my intellectual development in the field of numerical linear algebra. His expertise, vision and ideas, which underlie most of the results in this thesis, have been invaluable. Countless helpful discussions over the years have resulted in the successful solution of problems ranging from numerical linear algebra to astrophysics to algebras with polynomial identities. His clarity of presentation has helped me immensely in learning how to make this thesis more readable.

A major source of inspiration in this work is attributed to many supporting colleagues. I am very grateful to William Kahan for many discussions on computer arithmetic and correct rounding of transcendental functions as well as numerical linear algebra in general. All ideas in the chapter on the generalization of Harry Diamond's theorem belong to him, including the motivation and importance of this result, which were most difficult to comprehend.

I would like to thank: Beresford Parlett for sharing with me his expertise and unbounded enthusiasm for numerical linear algebra; Gene Golub for several conversations on the conditioning of Vandermonde systems; Nicholas Higham for sharing his extensive knowledge of the numerical treatment of Vandermonde matrices, which resulted in substantial improvements in the algorithms for generalized Vandermonde matrices; Ming Gu for several fruitful conversations, most notably on an algorithm for implementing the computation of removable singularities when computing the SVDs of polynomial Vandermonde matrices; Alberto Grunbaum for helping me understand better the computational aspects of orthogonal polynomials; Vesselin Drensky for a number of ideas on the applications of computers to detect or disprove existence of polynomial identities in matrix algebras; John Strain for his excellent teaching of the course on Matrix Computations which introduced me to the world of numerical linear algebra; Julian Borrill for introducing me to the infinite world of astrophysics and his patience in repeatedly explaining astrophysics concepts until I believed I understood them; the staff at Lawrence Berkeley National Laboratory and in particular the National Energy Research Scientific Computing Center for providing computational resources, excellent working conditions and funding my research for a year; Richard Vuduc, David Bindel and Christopher Cantalupo for answering countless programming questions in the process of implementation of the algorithms.

Nina Boiadjieva has provided invaluable support over the years and I thank her for enduring not seeing much of me in the last few months of preparation of this thesis.

Finally, I would like to once again thank Professor Demmel for bringing some of the daunting challenges of numerical linear algebra to my attention, for funding my research in my final year, and for encouraging me to present the results of this thesis at a number of national and international conferences.

Chapter 1

Introduction

Computational mathematics is becoming ever more important in modern society as harder and harder scientific and engineering problems await cost-effective solutions. Computing solutions to real life problems is most often the cheapest, fastest, safest and sometimes only way to attack a problem. There are numerous examples from engineering, including crash testing automobiles and airplanes, designing mechanical and electrical devices as well as science, e.g. analyzing the Cosmic Microwave Background (CMB) radiation (which we discuss in detail later on), etc.

The process of solving a real-life engineering or applied science problem has the following three phases:

- A scientific model is built by the engineers or applied scientists.
- A simplified mathematical model is then built by the applied mathematicians in collaboration with the engineers or applied scientists. The computational bottleneck in the ultimate solution is often the solution of a linear system, an eigenvalue or a singular value problem.
- The mathematical model problem is then solved using tools built by computer scientists on a particular computing platform.

Understanding all three phases and the interaction between them is a necessity in solving any real-life problem successfully. In this thesis we present new results in all three areas and the interactions between them.

- We exploit the algebraic properties of highly ill conditioned problems as well as the properties of computer arithmetic in order to obtain accurate solutions in spite of ill conditioning. These include:
  - A new $O(n^3)$ algorithm for computing the SVD of polynomial Vandermonde matrices involving orthonormal polynomials to high relative accuracy (Chapter 2);
  - A new $O(n^3)$ algorithm for computing the SVD of weakly diagonally dominant M-matrices to high relative accuracy (Chapter 3).
- Using combinatorial tools from pure mathematics, e.g. properties of Schur functions, we derive a new Björck-Pereyra-type algorithm for solving generalized Vandermonde linear systems, that is exponentially faster than the previous best accurate algorithm (Chapter 4).
- We generalize Harry Diamond's theorem, which is a property of correctly rounded functions and may serve as a basis for specifications for correctly rounding transcendental functions, which is still an open problem (Chapter 5).
- Through collaboration with a team of astrophysicists and after gaining an insight into the physical properties of the astrophysical model for estimating the power spectrum of the Cosmic Microwave Background radiation, we produced new, faster algorithms that will be used for analyzing data from future experiments. These are described in Chapter 6.
- Using techniques commonly used in applied mathematics we solve problems in pure mathematics. We partly prove a conjecture on the generators of the T-ideal of the algebra of $n \times n$ matrices over a field of characteristic 0. Namely we prove that all polynomial identities of degree $2n + 2$ in the above algebra follow from the standard identity for $n = 4$ and $n = 5$ (Chapter 7).

The unifying theme in this thesis is exploiting the physical, combinatorial and algebraic structure in a problem in order to obtain solutions that are faster and/or more accurate. We continue with a more detailed introduction to the problems presented in this thesis.

1.1 A New $O(n^3)$ Algorithm for Computing the SVD of a Polynomial Vandermonde Matrix Involving Orthonormal Polynomials to High Relative accuracy

A polynomial Vandermonde matrix is defined as

$$V_P(x_1, x_2, \dots, x_n) = \begin{bmatrix} P_0(x_1) & P_1(x_1) & \dots & P_{n-1}(x_1) \\ P_0(x_2) & P_1(x_2) & \dots & P_{n-1}(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ P_0(x_n) & P_1(x_n) & \dots & P_{n-1}(x_n) \end{bmatrix},$$

where P is a basis of the space of polynomials of degree less than n and $\deg P_i = i$.

The polynomial Vandermonde matrices appear in interpolation problems and can be extremely ill conditioned (the traditional condition number $\kappa(V_p) = \|V_p\| \cdot \|V_p^{-1}\|$ is very large), causing the traditional algorithms to compute the smallest singular values with no relative accuracy whatsoever. Despite $\kappa(V_p)$ being enormous it turns out that the singular values are often well conditioned functions of the data $x_1, x_2, \dots, x_n$. This justifies our search for structure exploiting algorithms that will compute the smallest singular values accurately regardless of condition numbers.

Of particular interest to us are polynomial Vandermonde matrices that involve orthogonal or orthonormal polynomials. Because of a close connection between the polynomial Vandermonde and the Cauchy matrices we can easily reduce the problem of computing the SVD of polynomial Vandermonde matrices to high relative accuracy to the problem of computing the SVD of a Cauchy matrix to high relative accuracy, which was solved by Demmel in 1998.

The new results in Chapter 2 are:

- A new $O(n^3)$ algorithm for computing the SVD of polynomial Vandermonde matrices involving orthogonal polynomials to high relative accuracy, regardless of condition numbers.

- We answer an open question of Demmel [20] by eliminating the need for an extended precision table of the roots of unity for the purpose of computing the SVD of an ordinary Vandermonde matrix to high relative accuracy.

1.2 Computing the SVD of Weakly Diagonally Dominant M-Matrices to High Relative Accuracy

An $n \times n$ weakly diagonally dominant M-matrix A is a matrix for which the off diagonal elements are nonpositive $a_{ij} \leq 0, i \neq j$ and the row sums $s_i = \sum_{j=1}^{n} a_{ij} \geq 0$ are nonnegative for all i.

The following new results are presented in Chapter 3:

- We prove that the off diagonal elements and the row sums determine the SVD accurately in the sense that small relative changes in those quantities only cause small relative changes in the SVD.

- We present a new $O(n^3)$ algorithm for computing the SVD of A to high relative accuracy regardless of condition numbers. We prove that in fact the problem of computing an SVD or LDU decomposition of A is *never* ill conditioned, independent of $\kappa(A)$.

1.3 A New Algorithm for Computing the Schur Function with Small Relative Error

The Schur function $s_\lambda(x_1, x_2, \ldots, x_n)$ appears in many areas of mathematics, such as combinatorics, the theory of symmetric polynomials, representation theory of the symmetric group, etc. One of its many equivalent definitions, that is of numerical interest, is

$$s_\lambda(x_1, \ldots, x_n) = \det \left(\begin{bmatrix} x_1^{a_1} & x_1^{a_2} & \ldots & x_1^{a_n} \\ x_2^{a_1} & x_2^{a_2} & \ldots & x_2^{a_n} \\ & & \ddots & \\ x_n^{a_1} & x_n^{a_2} & \ldots & x_n^{a_n} \end{bmatrix} \right) \cdot \det \left(\begin{bmatrix} 1 & x_1 & \ldots & x_1^{n-1} \\ 1 & x_2 & \ldots & x_2^{n-1} \\ & & \ddots & \\ 1 & x_n & \ldots & x_n^{n-1} \end{bmatrix} \right)^{-1},$$

where $a_1 < a_2 < \ldots < a_n$ are integers and λ equals the partition

$$\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_n) = (a_n - n + 1, a_{n-1} - n + 2, \ldots, a_1).$$

The determinant of the ordinary Vandermonde matrix

$$\det \left(\left[x_i^{j-1} \right]_{i,j=1}^{n} \right) = \prod_{i>j} (x_i - x_j)$$

can be evaluated directly with small relative error in floating point arithmetic. Thus if we can compute the Schur function accurately and efficiently, we can evaluate the determinant of a generalized Vandermonde matrix (the matrix on the left) and its minors accurately and efficiently. If we can compute accurate minors, we can compute accurate solutions of $Ax = b$, SVD and LDU decompositions, etc. of generalized Vandermonde matrices.

The Schur function is a symmetric polynomial with nonnegative coefficients, which implies that it can be computed with small relative error if $x_i > 0$ for all i. The only question is cost. Evaluating the Schur function directly as a polynomial is about the same as the number of nonzero terms in it, which is an exponential function

$$O\left(n^{\lambda_1+\lambda_2+\ldots+\lambda_n}\right).$$

In Chapter 4 we present a new algorithm for accurate computation of the Schur function that depends only *linearly* on n and whose total cost can be polynomially bounded by:

$$2(\lambda_1+1)(\lambda_2+1)^2\ldots(\lambda_n+1)^2 p(n+5)$$

where p is the number of nonzero parts of λ. This result is of independent interest for the purpose of computing Jack polynomials [69, 34], which are generalizations of the Schur functions, and to which the same algorithm applies with minor modifications.

1.4 Björck-Pereyra-Type Methods for Generalized Vandermonde Matrices

Totally positive Vandermonde and Cauchy linear systems can be solved very accurately, regardless of condition number, using Björck-Pereyra-type methods. They are the perfect example of structure exploiting algorithms that deliver more accuracy than the traditional algorithms and run in time $O(n^2)$ compared with $O(n^3)$ for the traditional algorithms.

In Chapter 4 we present the following new results.

- We identify combinatorial properties that a matrix or a class of matrices must possess in order for Björck-Pereyra-type methods to exist for these matrices, namely the matrix must be Totally Positive and its *initial* minors must be computable efficiently with small relative error.

- Using our new algorithm for computing the Schur function we design new Björck-Pereyra-type algorithm for the solution of totally positive generalized Vandermonde systems, whose cost depends only quadratically on n.

These results extend the work of Bjorck-Pereyra [8], Higham [43] and Boros-Kailath-Olshevsky [12] and improves a result by Hugo van de Vel [71].

1.5 Stability of Rounded-Off Inverses Under Iteration. Generalization of Harry-Diamond's Theorem

Let f be a monotone and convex (or concave) function on a real interval and let g be the inverse function. We fix a positive integer N and for any expression x we denote by $[x]$ the value of the expression x rounded to N significant binary digits. We call the number $\epsilon = 2^{-N}$ machine precision and assume that the arithmetic is performed in accordance with the IEEE 754 Standard for binary floating point arithmetic [4] in standard rounding mode, i.e. each result is rounded to the nearest even floating point number. We use capital letters for the rounded-off floating point values of the lowercase variables or functions, e.g. $X = [x]$, $F(x) = [f(x)]$, etc. Let $H(x) = G(F(x))$, the composition of F and G. For any X it is shown that

$$H(H([x])) = H(H(H([x])))$$

and there exist simple strictly convex functions for which $H([x]) \neq H(H([x]))$.

This is a generalization of a theorem by Harold Diamond [26], which required strict convexity (or concavity of f), but did not impose restrictions on how the arithmetic rounds the ambiguous (or halfway) cases.

Finally we prove that $H([x]) = H(H([x]))$ for the following functions: $f(x) = x + C$; $f(x) = C \cdot x$; $f(x) = C/x$; $f(x) = e^x$ and $f(x) = \ln(x)$.

1.6 Estimating the Power Spectrum of the Cosmic Microwave Background Radiation

The cosmic microwave background (CMB) radiation provides us with a snapshot of the Universe as it was some 300,000 years after the Big Bang. As the earliest possible view of the Universe it is the best probe we have of conditions in the very early Universe. In particular, anisotropies in the CMB contain information on virtually all cosmological parameters of interest, including the geometry of the universe, the baryon density, the Hubble expansion rate, the cosmological constant, etc.

Realizing the extraordinary scientific potential of the CMB requires precise measurements of these tiny anisotropies over a significant fraction of the sky at very high resolution. Current methods for analysis of the CMB scale as

$$O\left(n^3\right),$$

where n is the number of pixels observed. Since the PLANCK satellite, to be launched in 2007, will observe tens of millions of pixels current algorithms will no longer be applicable even on supercomputers.

The matrices involved in the map making and power spectrum estimation of the CMB are structured, which allowed us to design new, parallel, structure exploiting algorithms which run in time

$$O\left(n^{3/2}\right)$$

and scale nearly linearly with the number of processors on parallel platforms.

The code was implemented in C on the IBM SP at Lawrence Berkeley National Laboratory (LBL) in collaboration with Julian Borrill and Christopher Cantalupo at the CMB Astrophysics Research Program at LBL. The code will be used to analyze the data to be delivered by the PLANCK satellite in 2009. We presented these algorithms at 2001 PLANCK Workshop on Image Processing held in Pisa, Italy, July 4-6, 2001 which brought together over 200 researchers from around the world working on the PLANCK satellite experiment to discuss design of the experiment, algorithms and analysis of the CMB data.

1.7 Computational Approach to Polynomial Identities of Matrices

The polynomial identities of the algebra $M_n(K)$ of $n \times n$ matrices over a field K of characteristic 0 are among the most intriguing topics in the theory of algebras with polynomial identities. Since 1950, when Amitsur and Levitzki proved their famous theorem that $M_n(K)$ satisfies the so called *standard identity* of degree $2n$, many results have been established. Nevertheless, the picture is far from a satisfactory form and only the structure of the polynomial identities of the 2×2 matrices is well known. There are many open problems and conjectures concerning the bases of polynomial identities of $M_n(K)$, and one of the most interesting questions is that of finding the minimal degree of the polynomial identities which do not follow from the standard identity.

Some of the facts in the theory have been obtained, or at least conjectured, as a result of extensive calculations by hand or computer. We develop further the computational methods of Drensky [30, 32] for calculations with the polynomial identities of matrices and obtain the following new results:

- We present a new divide-and-conquer method for computing the complete skew-symmetric polynomial

$$s_m(x_1, x_2, \ldots, x_m) = \sum_{\sigma \in S_m} \mathrm{sign}(\sigma) x_{\sigma(1)} x_{\sigma(2)} \cdots x_{\sigma(m)}, \tag{1.1}$$

  where S_m is the symmetric group of order m, i.e. the group of all permutations σ on m objects, and each x_i is an $n \times n$ matrix of numerical values. Our method is exponentially faster than a straightforward evaluation of the explicit polynomial, which was the method used in prior work [30].

- Using this algorithm we were able to perform extensive computations and to prove the following result:

  All polynomial identities in $M_n(K)$ of degree $2n+2$ are consequences of the standard identity of degree $2n$ for $n = 4, 5$.

Chapter 2

Accurate SVDs of Polynomial Vandermonde Matrices Involving Orthonormal Polynomials

We consider the problem of computing the SVD of $n \times n$ polynomial Vandermonde matrices $V_P = [P_{i-1}(x_j)]$ to high relative accuracy. The small singular values of these matrices can be arbitrarily smaller than the largest ones, so that traditional algorithms compute them with no relative accuracy at all.

When the P_i form a sequence of orthonormal polynomials, the polynomial Vandermonde matrix can be represented as a product of a Cauchy matrix and an orthogonal matrix Q. We compute the SVD of a Cauchy matrix to high relative accuracy using an algorithm of Demmel and postmultiply it by Q to obtain an accurate SVD of V_P.

We also show that the singular values, even the tiniest ones, are usually well-conditioned functions of the data x_i, justifying this computation.

We also explain how this theory can be extended to other polynomial Vandermonde matrices, involving polynomials that are not orthonormal or even orthogonal.

2.1 Introduction

A *Polynomial Vandermonde* matrix is a matrix of the form

$$V_P(x_1, x_2, \ldots, x_n) = \begin{bmatrix} P_0(x_1) & P_1(x_1) & \ldots & P_{n-1}(x_1) \\ P_0(x_2) & P_1(x_2) & \ldots & P_{n-1}(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ P_0(x_n) & P_1(x_n) & \ldots & P_{n-1}(x_n) \end{bmatrix},$$

where $P = \{P_0(x), P_1(x), \ldots, P_{n-1}(x)\}$ denotes a basis of the linear space of all complex polynomials of degree less than n and $\deg P_i = i$. When $P_i(x) = x^i$ we obtain the standard Vandermonde matrix.

We will consider *orthogonal polynomials*, i.e. a set of polynomials that form a basis of the space of polynomials of degree less than n and satisfy the orthogonality relation

$$\int_a^b w(x) P_m(x) P_n(x) dx = \delta_{mn} c_n,$$

where $w(x)$ is a nonnegative weighting function, $\deg P_i = i$ and

$$c_n = \int_a^b w(x) P_n^2(x) dx. \tag{2.1}$$

If $c_i = 1$ for all i then the polynomials are called *orthonormal.* A set of orthogonal polynomials can be easily normalized: $\left\{\frac{P_n(x)}{\sqrt{c_n}}\right\}_{n=0}^{\infty}$. The orthogonal and orthonormal polynomials are useful in the solution of various mathematical and physical problems and provide a natural way to solve, expand, and interpret solutions to many types of important differential equations. For more information we refer the reader to [70, 1]. The roots of P_i are simple, real, in $[a, b]$ and interlace those of P_{i+1}. All orthogonal (orthonormal) polynomials satisfy a *three-term recurrence*

$$P_{n+1}(x) = d_n(x - b_n)P_n(x) - a_n P_{n-1}(x), \qquad P_0(x) = a_0, P_1(x) = d_0 x + b_0.$$

The *discrete orthogonality property* [39] of the orthonormal polynomials is of prime importance in everything that follows: If ξ_i are the roots of P_n, then:

$$\sum_{i=1}^{n} \lambda_i P_m(\xi_i) P_k(\xi_i) = \delta_{mk}, \text{ for } k, m < n,$$

where λ_i are the Christoffel numbers.

When P is a set of orthogonal (orthonormal) polynomials, we say that V_P is a polynomial Vandermonde matrix involving orthogonal (orthonormal) polynomials.

There are many classes of important orthogonal polynomials. The properties of some of them [1, 70] are given in Table 2.1.

Table 2.1: Properties of orthogonal polynomials

Type	Interval	$w(x)$	c_j	d_j	b_j	a_j
Chebyshev, first kind	$[-1,1]$	$(1-x^2)^{-1/2}$	π, $c_0=\frac{\pi}{2}$	2, $d_0=1$	0	1
Chebyshev, 2nd kind	$[-1,1]$	$\sqrt{1-x^2}$	$\frac{\pi}{2}$	2	0	1
Legendre ($P_j(1)\equiv 1$)	$[-1,1]$	1	$\frac{2}{2j+1}$	$\frac{2j+1}{j+1}$	0	$\frac{j}{j+1}$
Laguerre	$[0,\infty)$	e^{-x}	1	$-\frac{1}{j+1}$	$2j+1$	$\frac{j}{j+1}$
Hermite	$(-\infty,\infty)$	e^{-x^2}	$\sqrt{\pi}2^j j!$	2	0	$2j$

Our goal in this chapter is to *accurately and efficiently* compute the Singular Value Decomposition (SVD) of $V_P = W\cdot\Sigma\cdot Z^T$, where W and Z are orthogonal, and $\Sigma = \mathrm{diag}(\sigma_1,\sigma_2,\ldots,\sigma_n); \sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n \geq 0$ are the singular values. By *accurately* we mean [20, 22]:

- The error $|\sigma_i - \hat{\sigma}_i|$ in the computed singular value σ_i is bounded by $O(\epsilon)\sigma_i$, where ϵ is the machine precision, i.e. the relative error is small.

- The angle $\theta(w_i,\hat{w}_i)$ between the true left singular vector w_i and the computed singular vector $\hat{w}_i$ is bounded by $O(\epsilon)/\mathrm{relgap}_i$, where $\mathrm{relgap}_i = \min_{i\neq j}|\sigma_j-\sigma_i|/\sigma_i$ is the relative gap between σ_i and the nearest other singular value. An analogous statement holds for the computed right singular vectors $\hat{z}_i$.

By *efficiently* we mean that the cost must be bounded by a polynomial function of n, preferably $O(n^3)$, *independent of the condition number* $\kappa(G)=\|G\|\cdot\|G^{-1}\|$.

We will use a standard model of floating point arithmetic, in which we assume only that the relative error of any arithmetic operation is small:

$$fl(x\odot y) = (x\odot y)(1+\delta) \tag{2.2}$$

where $\odot\in\{+,-,*,/\}$ and $|\delta|<\epsilon$ for some fixed ϵ, called *machine precision*. We also assume that no underflow or overflow occurs. This model implies that products, quotients, and sums of like-signed quantities can be computed accurately (i.e. with low relative error), but expressions involving cancellation may not be (for example, the sum of three numbers can provably not be computed accurately in this model [25]). However, if x and y are inputs (and so can be considered exact) then $fl(x \pm y) = (x \pm y)(1 + \delta)$ is computed accurately.

In what follows we may use a single variable x to indicate a vector $(x_1, x_2, \ldots, x_n)$, e.g. $V_P(x)$ will be the same as $V_P(x_1, x_2, \ldots, x_n)$. For ease of notation we may also use $x_{i:j}$ to indicate the vector $(x_i, x_{i+1}, \ldots, x_j)$. Also, when the choice of polynomials is clear from the context we may only write $V(x)$ instead of $V_P(x)$.

If $x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_n)$ then a *Cauchy matrix* $C(x, y)$ is defined as

$$C = \left[\frac{h_i f_j}{x_i - y_j} \right]_{i,j=1}^{n},$$

where h_i and f_j are given constants.

Our algorithm is based on a simple identity $V_P(x) = C(x, y)V_P(y)$ (see Proposition 2.3.1 for what the constants h_i and f_j are in this case) allowing us to change the set of points at which the polynomial Vandermonde matrix is evaluated at the cost of premultiplication by a Cauchy matrix $C(x, y)$. This identity, proven in a more general setting in [13], is nothing more than Lagrange interpolation in disguise. If y are the roots of P_n then the discrete orthogonality property implies that $V(y) = \Lambda^{1/2} \cdot Q$, where Λ is a diagonal matrix with the Christoffel numbers on the diagonal and Q is orthogonal. The matrix $C(x, y) \cdot \Lambda^{1/2}$ is still Cauchy and we can compute its SVD to high relative accuracy in $O(n^3)$ time by using the algorithm of Demmel [20]. A postmultiplication by Q is all it takes to get the SVD of $V_P(x)$. There are a number of issues to be addressed, namely

- How do we accurately compute the roots y of P_n and then form $V(y)$?
- What do we do if $x_i = y_j$ for some i and j? This turns out to be a removable singularity. In the case of a standard Vandermonde matrix ($P_i(x) = x^i$) we simplify the SVD algorithm in [20] by handling the case of $x_i = y_j$ for some i and j without the need to compute the roots of unity $y_j = e^{2\pi\sqrt{-1}j/n}$, $j = 1, 2, \ldots, n$ to higher than working precision.
- What if the P_i are orthogonal, but not orthonormal, i.e. $V_P(y) = \Lambda^{1/2} \cdot Q \cdot F^{1/2}$, where Λ is diagonal, Q is orthogonal and $F = \mathrm{diag}(c_1, c_2, \ldots, c_n)$ is also diagonal?
- Is this computation justified in the sense that the tiny singular values are in fact

well-conditioned functions of the data x_i? We answer this affirmatively in section 2.7 below.

This chapter is organized as follows:

- In Section 2.2 we briefly review the algorithm of Demmel [20] for computing the SVD of a Cauchy matrix to high relative accuracy.
- Section 2.3 describes the connection between Cauchy and polynomial Vandermonde matrices and states the main $O(n^3)$ algorithm for computing the SVD of polynomial Vandermonde matrices involving orthonormal polynomials.
- In Section 2.4 we explore the conditions under which our main algorithm applies to polynomial Vandermonde matrices that do not necessarily involve orthogonal or orthonormal polynomials and consider various examples.
- In Section 2.5 we explain how to compute roots of orthogonal polynomials and the Christoffel numbers.
- We explain how to handle removable singularities in the algorithm for computing the LDU decomposition of a Cauchy matrix in Section 2.6.
- We present our perturbation theory in section 2.7.
- Finally we present our numerical experiments in section 2.8.

2.2 Computing the SVD of a Cauchy Matrix

A *Rank revealing decomposition* (RRD) [22] is a factorization $A = XDY$ where the matrices X and Y are well-conditioned and D is diagonal. An example of an RRD is the LDU decomposition of a matrix resulting from Gaussian Elimination with complete pivoting, since in practice L and U are well-conditioned. Whether a RRD can be computed accurately depends on the structure of the matrix. For a Cauchy C matrix the LDU from Gaussian Elimination with complete pivoting can be computed with small relative error in each entry of L,D and U [20] in $O(n^3)$ time by exploiting the fact that each entry of L, D and U is a quotient of minors of C and all minors of a Cauchy matrix are computable accurately and efficiently. This is achieved by making a small change in the inner loop of Gaussian Elimination from

$$C_{ij} = C_{ij} - \frac{C_{ik}C_{kj}}{C_{kk}} \tag{2.3}$$

to

$$C_{ij} = C_{ij}\frac{(x_i - x_k)(y_k - y_j)}{(x_k - y_j)(x_i - y_k)}, \tag{2.4}$$

which preserves the relative accuracy at every step.

The SVD algorithm in [22] allows us to compute in $O(n^3)$ time the SVD of any matrix for which an RRD is available.

2.3 Main Algorithm

The following proposition is proven in [13] and as mentioned previously is just Lagrange interpolation written in matrix form.

Proposition 2.3.1 *Let* $\{x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n\}$ *be* $2n$ *pairwise distinct (complex) numbers, let* $u(x) = \prod_{i=1}^{n}(x - y_i)$*, and* P *be a basis of the space of polynomials of degree less than* n *with* $\deg P_i = i$*. The following formula is valid.*

$$V_P(x_{1:n}) = \mathrm{diag}(u(x_1), \ldots, u(x_n)) \cdot \left[\frac{1}{x_i - y_j}\right]_{i,j=1}^{n} \cdot \mathrm{diag}\left(\frac{1}{u'(y_1)}, \ldots, \frac{1}{u'(y_n)}\right) \cdot V_P(y_{1:n}).$$

We will also use the following

Proposition 2.3.2 (Discrete Orthogonality Property [39]) *If* P *is a set of orthonormal polynomials with* $\deg P_i = i$ *and* $\xi_1, \xi_2, \ldots, \xi_n$ *are the roots of* P_n*, then*

$$\sum_{i=1}^{n} \lambda_i P_m(\xi_i) P_k(\xi_i) = \delta_{mk},$$

for $k < n$ *and* $m < n$*. The* λ_i *are the Christoffel numbers.*

Corollary 2.3.1 *Under the above assumptions, the matrix* $\Lambda^{1/2} \cdot V_P(\xi)$ *is orthogonal, where* $\Lambda = \mathrm{diag}(\lambda_1, \lambda_2, \ldots, \lambda_n)$.

Algorithm 2.3.1 *The following algorithm computes the SVD of* $V_P(x)$*, where* P *is a set of orthonormal polynomials,* $\deg P_i = i$ *and* $x = (x_1, x_2, \ldots, x_n)$.

1. *Compute the roots* $y_1, y_2, \ldots, y_n$ *of* P_n.

2. *Let* $u(x)$ *and* λ_i *be defined as above. Using the algorithm of Demmel [20] compute the SVD of the Cauchy matrix* $C = W \cdot \Sigma \cdot Z^T$, *where*

$$C_{ij} = \frac{u(x_i) \cdot [u'(y_j)]^{-1} \lambda_j^{-1/2}}{x_i - y_j} \tag{2.5}$$

3. *The SVD of* $V_P(x)$ *is*

$$V_P(x_{1:n}) = W \cdot \Sigma \cdot \left(Z^T \cdot \left(\Lambda^{1/2} \cdot V(y)\right)\right), \tag{2.6}$$

   where $S^T = Z^T \cdot \left(\Lambda^{1/2} \cdot V(y)\right)$ *is the product of two orthogonal matrices.*

Note: If $x_i = y_j$ for some i and j then all expressions are defined in the limiting sense. This is also easily implementable on a computer as explained in Section 2.6.

Theorem 1 *Algorithm 2.3.1 computes the SVD of* $V(x)$ *to high relative accuracy, provided all* λ_i *are computed to high relative accuracy and* $\Lambda^{1/2} \cdot V(y)$ *is close to orthogonal and is formed with small* norm *error. The cost is* $O(n^3)$.

Note: We do not need that y_i's be computed with any given accuracy as long as the λ_i's are computed accurately and $\Lambda^{1/2} \cdot V(y)$ is close to orthogonal and with small norm error.
Proof:

Since $u(x) = \prod_{i=1}^{n}(x - y_i)$ all expressions $u(x_i)$ and $u'(y_j) = \prod_{i=1, i\neq j}^{n}(y_j - y_i)$ will be computed to high relative accuracy. Then [20] guarantees that the SVD of C will be computed to high relative accuracy at $O(n^3)$ cost. The product

$$S^T = Z^T \cdot \left(\Lambda^{1/2} \cdot V(y)\right) \tag{2.7}$$

can also be formed at $O(n^3)$ cost. Both factors are orthogonal and with small norm error, therefore the same will be true for their product. The SVD algorithm in [22] guarantees that $\theta(z_i, \hat{z}_i) \leq O(\epsilon)/\text{rel\_gap}_i$. Since the orthogonal matrix $\Lambda^{1/2} \cdot V(y)$ is formed with small norm error then the additional error in the angle of the computed s_i when forming the product (2.7) will be $O(\epsilon)$. Therefore an error bound of the form $\theta(s_i, \hat{s}_i) \leq O(\epsilon)/\text{rel\_gap}_i$ for the right singular vectors s_i will hold. $\square$

2.4 The SVD of Polynomial Vandermonde Matrices Involving Polynomials that are Not Necessarily Orthonormal or Even Orthogonal

The SVD of V_P can still be computed to high relative accuracy in many cases when the set of polynomials P is not orthonormal or not even orthogonal. Recalling that $V_P(x) = C \cdot V_P(y)$, where C is Cauchy, we can compute the SVD of any $V_P(x)$ as long as there exist a set of points y for which $V_P(y) = D' \cdot M$, where D' is diagonal and M is well-conditioned. To do so we first compute an LDU of the Cauchy matrix $C \cdot D' = LDU$. We then have an RRD of $V_P(x) = L \cdot D \cdot (U \cdot M)$, which we use to obtain the SVD of $V_P(x)$ using the algorithm from [22]. Of course D' must be computable with small relative error componentwise and M must be computable with small norm error.

For the orthonormal polynomials, as we saw, y can be chosen so M is orthogonal. What if the polynomials P are orthogonal, but not normalized? That is

$$V(y) = \Lambda^{1/2} \cdot Q \cdot F^{1/2},$$

where $F = \mathrm{diag}(c_1, c_2, \ldots, c_n)$, c_n are defined as in (2.1) and Q is orthogonal. The matrix $Q \cdot F^{1/2}$ has singular values $c_i^{1/2}$. Therefore if the c_i are of not widely varying magnitude, e.g. in the case of the Chebyshev polynomials of the first kind or the Legendre polynomials, then $Q \cdot F^{1/2}$ will be well conditioned and we can still compute the SVD of $V(x)$ to high relative accuracy in $O(n^3)$ time.

In fact, since our goal is to compute the SVD in polynomial time, we can tolerate a much wilder behavior of the c_i's. In the case of Hermite polynomials for example the matrix $B = Q \cdot F^{1/2}$ would seem very ill-conditioned ($\mathrm{cond}(B) = \sqrt{\sqrt{\pi}2^n n!}$). From the complexity point of view our algorithm will still complete in polynomial time, if we choose $\epsilon < \log(\mathrm{cond}(B))$. Choosing such an ϵ in practice means that we run our algorithm in precision that is about n times wider than the working precision, which will only add at most an n^2 factor to the complexity of the algorithm. This is because an addition in this wider precision costs $O(n)$ and a multiplication costs at most $O(n^2)$. This analysis, however, goes beyond the scope of this chapter and we will not pursue it further.

The situation when P are not orthogonal must be considered on a case-by-case basis. A perfect example in this regard is the standard Vandermonde matrix $V(x) = \left[x_i^{j-1}\right]_{i,j=1}^{n}$.

Even though the monomials are not orthogonal, we have [20] for $y_j = e^{\frac{2\pi\sqrt{-1}(j-1)}{n}}$

$$V(y) = \left[e^{\frac{2(i-1)(j-1)\pi\sqrt{-1}}{n}}\right]_{i,j=1}^{n}$$

is the matrix of the *Discrete Fourier Transform* (DFT), which is Hermitian. Thus $V(x) = C(x,y) \cdot (\mathrm{DFT})$ and if the SVD of $C(x,y)$ is $W \cdot \Sigma \cdot Z^T$ computed to high relative accuracy then the SVD of $V(x)$ is $W \cdot \Sigma \cdot \left(Z^T \cdot (\mathrm{DFT})\right)$ also computed to high relative accuracy.

2.5 Computing y, Λ and $V(y)$

In this section we assume that P is a set of orthonormal polynomials and consider the problem of an accurate computation of the roots $y_1, y_2, \ldots, y_n$ of $P_n(x)$, the Christoffel numbers $\lambda_1, \lambda_2, \ldots, \lambda_n$ and $\Lambda^{1/2} \cdot V(y)$. We argue that this computation should not be considered a part of our algorithm because it need only be done once and for all for each class of orthonormal polynomials. The problem of computing the y and Λ is equivalent to computing the Gauss-Christoffel quadrature formulas and is a problem that has been studied in detail [38, 41]. The problem may be very ill-conditioned if all that is known about the orthonormal polynomials is the respective weights $w(x)$. If the coefficients of the three-term recurrence are available, then the problem becomes a lot easier [40, 41] and the nodes y may be computed as eigenvalues of a tridiagonal matrix [38]. The Christoffel numbers are then the squares of the leading components of the corresponding eigenvectors. For Chebyshev, Legendre, Hermite and Laguerre orthonormal polynomials this problem is very well conditioned and allows for accurate computation of y and Λ using existing software.

It is also worth mentioning that in the case when $b_j = 0$ in the three-term recurrence (i.e. Chebyshev, Legendre and Hermite polynomials) then the tridiagonal eigenproblem becomes equivalent to the computation of the SVD of a bidiagonal matrix, which can be done to high relative accuracy, regardless of condition numbers [23].

In the case of Chebyshev polynomials and the monomials we have exact formulas for y_i and $V(y)$. The following formulas are valid on $[-1,1]$ for the Chebyshev polynomials of the first kind

$$T_n(x) = \cos(n \arccos x)$$

and the Chebyshev polynomials of the second kind

$$U_n(x) = \frac{\sin((n+1)\arccos x)}{\sin(\arccos x)}.$$

Their roots are $y_i = \cos\frac{(2i-1)\pi}{2n}$ and $y_i = \cos\frac{i\pi}{n+1}$ respectively. The matrices

$$V_T(y) = \left[\cos\frac{2(i-1)(2j-1)\pi}{n}\right]_{i,j=1}^{n} \quad \text{and} \quad V_U(y) = \left[\frac{\sin\frac{(i-1)j\pi}{n+1}}{\sin\frac{j\pi}{n+1}}\right]_{i,j=1}^{n}$$

are computable to high relative accuracy componentwise. The Chebyshev polynomials of the first kind are not orthonormal, but $\operatorname{cond}(V_T(y)) = \operatorname{cond}(V_U(y)) = \sqrt{2}$, which means that we can still compute the SVD of any $V_T(x)$ and $V_U(x)$ to high relative accuracy. If we have the orthonormal Chebyshev polynomial bases T' and U' instead of T and U then we can obtain the Christoffel numbers by scaling the rows of $V_{T'}(y)$ and $V_{U'}(y)$ and thus obtaining orthogonal matrices.

In the case of the standard Vandermonde matrix, when we use the basis of monomials, we can compute $y_i = e^{\frac{2\pi i}{n}} = \cos\frac{2\pi}{n} + i\sin\frac{2\pi}{n}$ accurately and we have $V(y) = \left[e^{\frac{2i(j-1)\pi}{n}}\right]_{i,j=1}^{n}$ is the matrix of the Discrete Fourier Transform, which is Hermitian and is trivially computable with small norm error.

2.6 Computer Implementation

All algorithms we use have been implemented before. We only owe an explanation on how to handle the case of $x_i = y_j$ when computing an LDU decomposition that results from Gaussian Elimination with complete pivoting applied to a Cauchy matrix. Row (column) pivoting in a Cauchy matrix preserves the Cauchy structure and is equivalent to permuting the x_i's (y_i's).

The algorithm from [20] for computing an LDU decomposition of a Cauchy matrix

$$C = \operatorname{diag}(u(x_1), \ldots, u(x_n)) \cdot \left[\frac{1}{x_i - y_j}\right]_{i,j=1}^{n} \cdot \operatorname{diag}\left(\frac{1}{u'(y_1)}, \ldots, \frac{1}{u'(y_n)}\right) \cdot D, \tag{2.8}$$

where $u(x) = \prod_{i=1}^{n}(x - y_i)$, $u'(y_i) = \prod_{j=1, j\neq i}^{n}(y_i - y_j)$ and the x_i and y_j are pairwise distinct and D is diagonal looks like this:

Algorithm 2.6.1 *Gaussian elimination on a Cauchy matrix (2.8) when* $x_i \neq y_j$

```
for i = 1 : n
   for j = 1 : n
      C_ij = u(x_i)(u'(y_j))^{-1} D_j / (x_i - y_j)
for k = 1 : n - 1
   Find the largest absolute entry in C(k : n, k : n)
   Swap rows and columns of C, and entries of x and y,
       so C_kk is largest
   if C_kk = 0 quit
   for i = k + 1 : n
      for j = k + 1 : n
         C_ij = C_ij (x_i - x_k)(y_k - y_j) / ((x_k - y_j)(x_i - y_k))
```

If $x_i = y_j$ for some i and j then every entry of the i-th row of C is zero, except $C_{ij} = D_j$. In the elimination process we can update the nonzero entries as in Algorithm 2.6.1 and the zero entries using the traditional update formula (2.3) preserving the relative accuracy in each case. To allow the possibility of $x_i = y_j$ we change Algorithm 2.6.1 as follows:

Algorithm 2.6.2 *Gaussian elimination on a Cauchy matrix (2.8), general case*

```
for i = 1 : n
   for j = 1 : n
      C_ij = (u'(y_j))^{-1} D_j
      for k = 1 : n
         if k ≠ j then C_ij = C_ij · (x_i - y_k)
for k = 1 : n - 1
   Find the largest absolute entry in C(k : n, k : n)
   Swap rows and columns of C, and entries of x and y, so C_kk is largest
   if C_kk = 0 quit
   for i = k + 1 : n
      for j = k + 1 : n
         if C_ij = 0 then
            C_ij = - C_ik C_kj / C_kk
         else
            C_ij = C_ij (x_i - x_k)(y_k - y_j) / ((x_k - y_j)(x_i - y_k))
```

Here is how this algorithm works. The matrix C is formed. If $x_i = y_j$ then this singularity is first removed by cancelling the term $x_i - y_j$ both in the numerator and the denominator.

In the update process the update

$$C_{ij} = C_{ij} - \frac{C_{ik}C_{kj}}{C_{kk}} \tag{2.9}$$

is computed accurately when $C_{ij} = 0$. In all remaining cases we update C_{ij} according to (2.4). Here is why the update formula (2.4) is still valid in the case when $C_{kk} \neq 0, C_{ij} \neq 0$ and the x_i and y_j are not necessarily pairwise distinct.

Let $S^{(k)}$ be the k-th Schur complement. We first note that since the orthogonal polynomials have simple roots, we have $u'(y_i) \neq 0$ for all i. The conditions and $C_{kk} \neq 0$ and $C_{ij} \neq 0$ imply $S^{(k)}_{kk} \neq 0, S^{(k-1)}_{ij} \neq 0$ and $\det(C(1:k,1:k)) \neq 0$. Then following [20] we get

$$\det(C(1:k,1:k)) = \left(\prod_{1\leq r<s\leq k}(x_s - x_r)(y_r - y_s)\right) \cdot \left(\prod_{r=k+1}^{n}\prod_{s=1}^{k}(x_s - y_r)\right) \cdot \prod_{s=1}^{k} D_s \neq 0. \tag{2.10}$$

We also have

$$S^{(k)}_{ij} = \frac{\det(C([1:k,i],[1:k,j]))}{\det(C(1:k,1:k))} = D_j \cdot \left(\prod_{r=k+1,r\neq j}^{n} \frac{x_i - y_r}{y_j - y_r}\right) \cdot \left(\prod_{r=1}^{k} \frac{x_i - x_r}{y_j - x_r}\right) \tag{2.11}$$

where the denominator in the last expression is not zero because of (2.10). Similarly

$$S^{(k-1)}_{ij} = D_j \cdot \left(\prod_{r=k,r\neq j}^{n} \frac{x_i - y_r}{y_j - y_r}\right) \cdot \left(\prod_{r=1}^{k-1} \frac{x_i - x_r}{y_j - x_r}\right)$$

is also well defined because of (2.10). Therefore

$$\frac{S^{(k)}_{ij}}{S^{(k-1)}_{ij}} = \frac{(x_i - x_k)(y_k - y_j)}{(x_k - y_j)(x_i - y_k)},$$

i.e. (2.4) is still valid in this case.

With Algorithm 2.6.2 we answer an open question posed at the end of section 5 of [20]. Namely, we do not need a high precision table of the roots of unity in order to compute an accurate SVD of a standard Vandermonde matrix, when some of the nodes of the Vandermonde matrix coincide with the (floating point representations of the) roots of unity. In particular, the assumptions of Theorem 1 to guarantee the accuracy of Algorithm 2.3.1 may be easily confirmed. Indeed, we can compute roots of unity to high relative accuracy and the DFT to high norm accuracy.

2.7 Perturbation Theory

We have presented an algorithm that computes an accurate SVD, independent of how sensitive the output is to small changes in the input. It is hard to complain about this accuracy, but it is still of interest to know how sensitive the output is to changes in the input. This is because the computation is justified in many applications only when the sensitivity is low enough, since the input is rarely exactly known.

In this section we show that the singular values are typically well-conditioned functions of the x_i, *no matter how tiny they are.* Specifically, we will show that a small relative change $\eta \ll 1$ in the x_i can only cause a small relative change $\epsilon = O(\eta/\text{rel\_gap}_x)$ in the singular values, where rel_gap_x is the smallest *relative gap* between any pair of x_is:

$$\text{rel\_gap}_x \equiv \min_{i \neq j} \frac{|x_i - x_j|}{|x_i|} \quad .$$

This is lower bound on the sensitivity of *any* polynomial Vandermonde SVD (or inversion or determinant or linear equation solving) problem, not just for orthogonal polynomials, because a relative change of $\eta = \text{rel\_gap}_x$ in some x_i can make it equal to some other x_j, making the matrix singular and the smallest singular value zero, an $\epsilon = 100\%$ relative change. Another way to look at this is that the rel_gap_x is the (relative) distance from $V_P(x)$ to the nearest singular matrix, or ill-posed problem [18], since the matrix is singular if and only if two x_i are equal. Then the condition number is the reciprocal of this distance.

More specifically, in Theorem 2 we will prove this claim under the additional assumption that the relative gap between any x_i and y_j (defined as rel_gap_{xy} below) is at least as large as rel_gap_x. We conjecture that this technical assumption is unnecessary, a conjecture which is supported by numerical experiments in section 2.8. We also prove the claim without any assumptions for standard Vandermonde matrices in Theorem 3. Interestingly, it is easier to compute all the singular values of a standard Vandermonde matrix to high relative accuracy than it is to compute all the entries of its inverse (it is in fact impossible in our model of arithmetic because the inverse can contain expressions of the form $x_i + x_j + x_k$ [25]).

Theorem 2 *Let P be a basis of orthonormal polynomials, $y_1, y_2, \ldots, y_n$ be the roots of*

$P_n(x)$ and $x_1, x_2, \ldots, x_n, z_1, z_2, \ldots, z_n$ be such that

$$\begin{aligned} \text{rel\_gap}_x \equiv \min_{i \neq j} \frac{|x_i - x_j|}{|x_i|} &\geq \eta \\ \text{rel\_gap}_{xy} \equiv \min_{i,j} \frac{|x_i - y_j|}{|x_i|} &\geq \eta \\ \max_i \frac{|x_i - z_i|}{|x_i|} &\equiv \theta \end{aligned} \tag{2.12}$$

where $\theta \ll \eta \leq 1$. Let $V_P(x) = U \cdot \Sigma \cdot V^T$ and $V_P(z) = U' \cdot \Sigma' \cdot (V')^T$ be the SVDs of $V_P(x)$ and $V_P(z)$ respectively, where $\Sigma = \text{diag}(\sigma_1, \sigma_2, \ldots, \sigma_n)$ and $\Sigma' = \text{diag}(\sigma'_1, \sigma'_2, \ldots, \sigma'_n)$ are the singular values.

Then the following bound is valid

$$|\sigma_i - \sigma'_i| \leq O(n\theta/\eta)\sigma_i,$$

i.e. small relative changes in the x_i's only cause small relative changes in the singular values for as long as the relative gaps between the x_i's and the relative gaps between the x_i's and the y_j's are large enough.

Proof: We will use our algorithm to prove the perturbation theory. Let C and C' be the Cauchy matrices (2.5) corresponding to $V_P(x)$ and $V_P(z)$ respectively, and $C = LDU$ and $C' = L'D'U'$ be their LDU decompositions. The matrices C and $V_P(x)$ have the same singular values and so do the matrices C' and $V_P(z)$.

The inequalities (2.12) along with the identity (2.11) imply (after dropping higher order terms) that

$$|l_{ij} - l'_{ij}| \leq 2n(\theta/\eta) \cdot |l_{ij}|, \quad |d_{ii} - d'_{ii}| \leq 2n(\theta/\eta) \cdot |d_{ii}| \quad \text{and} \quad |u_{ij} - u'_{ij}| \leq 2n(\theta/\eta) \cdot |u_{ij}|.$$

Since the matrices L and U are assumed to be always well conditioned, we can use Theorem 2.1. from [22] to conclude (after dropping higher order terms) that

$$|\sigma_i - \sigma'_i| \leq O(2n\theta/\eta)\sigma_i.$$

□

We also conjecture that the above theorem is still valid independent of the relative gaps between the x_i's and the y_j's and successfully test this conjecture in the numerical experiments below.

Theorem 3 *Let $V(x)$ and $V(z)$ be standard Vandermonde matrices (i.e. $P_i(x) = x^i$). Define* $\mathrm{rel\_gap}_x$, *θ, σ_i and σ_i' as in Theorem 2. Then if $\theta \ll \mathrm{rel\_gap}_x \leq \frac{\pi}{2n^2}$, we have*

$$|\sigma_i - \sigma_i'| \leq O(n\theta/\mathrm{rel\_gap}_x)\sigma_i \quad .$$

Proof: The proof is very similar to the proof of the last theorem. Let $y_j = e^{2\sqrt{-1}\pi(j-1)/n}$. Note that if we multiply each x_i (and z_i) by ω, where $|\omega| = 1$, then $\mathrm{rel\_gap}_x$, θ, σ_i and σ_i' do not change, but $\mathrm{rel\_gap}_{xy}$ may. In particular, we can choose ω to make $\mathrm{rel\_gap}_{xy}$ at least about π/n^2, a worst case that occurs when the x_i are evenly spaced on the unit circle with angular separation $2\pi/n^2$. The rest of the proof follows as before. □

2.8 Numerical Experiments

To test our SVD algorithm we chose a 20×20 Chebyshev-Vandermonde matrix A (with the orthonormalized Chebyshev polynomials of the first kind) and uniformly distributed random nodes in [0.0.2] – within the interval of orthogonality. The resulting matrix had singular values, computed using our algorithm as implemented in Matlab, ranging over 35 orders of magnitude. We also computed the singular values using 60 digit arithmetic in Mathematica and got the same result to 14 digits. For comparison we computed the singular values also using a traditional SVD algorithm which computed the singular values with *absolute* error $\sigma_1\epsilon$.

We also tested the sensitivity of the SVD to small perturbation of the initial data. We ran two tests.

In the first test we took our test data below, for which the minimum relative gap between the x_i's ($\mathrm{rel\_gap}_x$) is $8.05 \cdot 10^{-5}$ and the minimum relative gap between the x_i's and the y_j's ($\mathrm{rel\_gap}_{xy}$) is $1.20 \cdot 10^{-2}$. We introduced random perturbation in the 10th digit of each of the x_i's, which resulted in a 5th digit perturbation in the relative gaps between the $x_i's$. We then computed the SVD of the thus perturbed matrices using Algorithm 2.3.1 and in Mathematica using 60 digit arithmetic. The singular values computed using each of these methods agreed with the singular values of the unperturbed matrix to 5 digits.

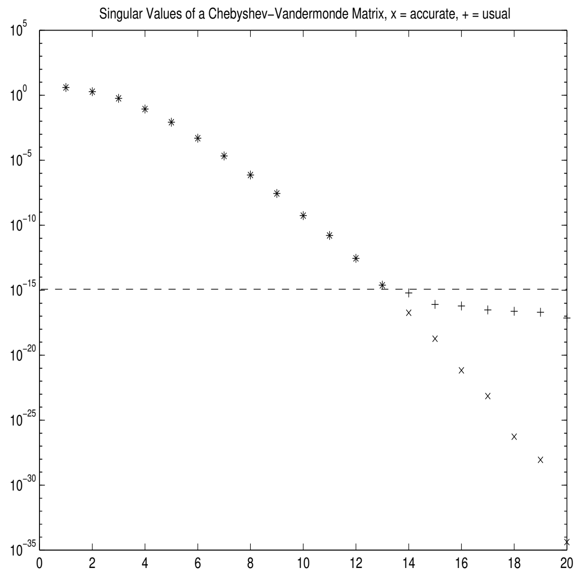
In our second test we tested the sensitivity of the algorithm to relative changes in the x_i's when the relative gaps between the x_i's and the y_j's are small. We replaced the odd nodes $x_1, x_3, \ldots, x_{19}$ with $y_1, y_3, \ldots, y_{19}$, ensuring that the relative gaps between the x_i and the y_j are zero. The minimum relative gap among the x_i's is $3.61 \cdot 10^{-3}$. As in our previous test we introduced a random perturbation in the 10th digit in each x_i and computed the SVD of the Chebyshev-Vandermonde matrix corresponding to the perturbed nodes. The singular values of the perturbed matrix agreed with the singular values of the unperturbed matrix to 7 digits.

Both numerical tests confirm our conjecture that the SVD is only sensitive to changes in the relative gaps between the x_i's, but is not sensitive to changes in the relative gaps between the x_i's and the y_j's.

Table 2.2: The nodes and singular values of a 20×20 Chebyshev-Vandermonde matrix

Random nodes in [0, 0.2]	Mathematica	Algorithm 2.3.1	Traditional
1.754545429110508e-1	3.9431633617865161e+00	3.9431633617865183e+00	3.9431633617865129e+00
1.785123568004681e-1	1.8637975498148207e+00	1.8637975498148214e+00	1.8637975498148200e+00
1.822530760791615e-1	5.8378797599607701e-01	5.8378797599607724e-01	5.8378797599607712e-01
1.337265877473653e-1	8.7323445025621033e-02	8.7323445025621074e-02	8.7323445025621046e-02
1.318492786266806e-1	8.4416423668173899e-03	8.4416423668173951e-03	8.4416423668173465e-03
1.399501459680857e-2	4.9924169675294070e-04	4.9924169675294081e-04	4.9924169675289918e-04
1.192756955101435e-1	2.1594947468609303e-05	2.1594947468609320e-05	2.1594947468623757e-05
1.466911661412304e-1	7.2784564714692648e-07	7.2784564714692700e-07	7.2784564716638695e-07
1.037535735742760e-1	2.7518442142941135e-08	2.7518442142941122e-08	2.7518442191715476e-08
1.791600477478306e-1	5.4735925469678211e-10	5.4735925469678263e-10	5.4735930523441333e-10
1.791744650424257e-1	1.6277393913826226e-11	1.6277393913826245e-11	1.6277437245500341e-11
9.045924651005480e-2	2.8485946493449978e-13	2.8485946493449983e-13	2.8487032864800328e-13
7.941574738520146e-2	2.4158345438031927e-15	2.4158345438031939e-15	2.4187910790488627e-15
1.200488382611424e-1	1.8338131347839962e-17	1.8338131347839956e-17	6.2473401943443176e-16
1.932439387160720e-1	1.8567109268982685e-19	1.8567109268982687e-19	8.1944522168311346e-17
1.736232440468190e-1	6.8715776044237070e-22	6.8715776044237004e-22	6.1039995944973018e-17
1.468702150155248e-1	7.1569407338259544e-24	7.1569407338259603e-24	3.0843251283083951e-17
7.419133543499865e-2	5.4140304287851849e-27	5.4140304287851849e-27	2.3979357147069272e-17
1.669780081142836e-1	8.9977317668911131e-29	8.9977317668911198e-29	2.0047699810203855e-17
1.020762301650144e-1	4.2539407269598119e-35	4.2539407269598130e-35	7.3091886573513421e-18

Figure 2.1: Singular values of a 20×20 Chebyshev-Vandermonde matrix



Chapter 3

Accurate SVDs of Weakly Diagonally Dominant M-Matrices

If we know the off diagonal elements and the row (column) sums of a row (column) weakly diagonally dominant M-matrix A with small relative error, then we can compute the LDU decomposition of A that results from Gaussian elimination with complete pivoting with small relative error in each entry of L, D and U. Then using an algorithm of Demmel [22] we can compute the SVD of A to high relative accuracy.

3.1 Introduction

We consider a weakly diagonally dominant M-matrix A for which we know the off diagonal elements $a_{ij} \leq 0$, $i \neq j$ and the row sums

$$s_i = \sum_{j=1}^{n} a_{ij} \geq 0$$

accurately (i.e. with small relative error).

Our goal in this chapter is to *accurately and efficiently* compute the Singular Value Decomposition (SVD) of $A = U \cdot \Sigma \cdot V^T$, where U and V are orthogonal, and $\Sigma = \mathrm{diag}(\sigma_1, \sigma_2, \ldots, \sigma_n); \sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n \geq 0$ are the singular values. By *accurately* we mean [20, 22]:

- The error $|\sigma_i - \hat{\sigma}_i|$ in the computed singular value σ_i is bounded by $O(\epsilon)\sigma_i$, where ϵ is the machine precision, i.e. the relative error is small.
- The angle $\theta(u_i, \hat{u}_i)$ between the true left singular vector u_i and the computed singular vector $\hat{u}_i$ is bounded by $O(\epsilon)/\mathrm{rel\_gap}_i$, where $\mathrm{rel\_gap}_i = \min_{i \neq j} |\sigma_j - \sigma_i|/\sigma_i$ is the relative gap between σ_i and the nearest other singular value. An analogous statement holds for the computed right singular vectors $\hat{v}_i$.

By *efficiently* we mean that the cost must be bounded by a polynomial function of n, preferably $O(n^3)$, *independent of the condition number* $\kappa(G) = ||G|| \cdot ||G^{-1}||$.

We will use a standard model of floating point arithmetic, in which we assume only that the relative error of any arithmetic operation is small:

$$fl(x \odot y) = (x \odot y)(1 + \delta) \tag{3.1}$$

where $\odot \in \{+, -, *, /\}$ and $|\delta| < \epsilon$ for some fixed ϵ, called *machine precision.* We also assume that no underflow or overflow occurs. This model implies that products, quotients, and sums of like-signed quantities can be computed accurately (i.e. with low relative error), but expressions involving cancellation may not be (for example, the sum of three numbers can provably not be computed accurately in this model [25]). However, if x and y are inputs (and so can be considered exact) then $fl(x \pm y) = (x \pm y)(1 + \delta)$ is computed accurately.

The off diagonal elements and the row sums appear to be the right parameters to choose in order to describe a weakly diagonally dominant M-matrix [2] in the sense that small relative changes in those quantities only cause small relative changes in each entry of matrix, each entry of the inverse, the smallest eigenvalue and as we will see now – the SVD.

3.2 Algorithm for Computing LDU and the SVD

The algorithm of Demmel [22] allows us to compute an accurate SVD for every matrix for which we can compute an accurate LDU decomposition that results from Gaussian elimination with complete pivoting. By accurate here we mean that each entry of L, D and U must be computed with small relative error. In what follows we will show how to compute such an LDU decomposition of A.

Our algorithm is a small variation on the GTH algorithm as discussed in [58] and referenced in [2]. The GTH algorithm applies in the special case where $A = I - S$ and S is a stochastic matrix (nonnegative with unit row sums). Thus A has all zero row sums in this case.

First we will review the classical fact that the Schur complement of A after one step of Gaussian elimination will also be a weakly diagonally dominant M-matrix [19, 42]. Furthermore, we can compute the off diagonal elements and row sums of the Schur complement to high relative accuracy. Since A is weakly diagonally dominant, the largest pivot will be on the main diagonal and so complete pivoting will preserve the property of being a weakly diagonally dominant M-matrix.

Let $A^{(2)}$ be the matrix A after one step of Gaussian elimination with complete pivoting. The diagonal elements of A can be computed in a forward stable manner as a sum of nonnegative quantities

$$a_{ii} = s_i - \sum_{j \neq i} a_{ij},$$

and so can the off diagonal elements, which will remain nonpositive

$$a'_{ij} = a_{ij} - \frac{a_{i1} a_{1j}}{a_{11}}. \tag{3.2}$$

For the row sums we similarly have

$$s'_i = s_i - \frac{a_{i1}}{a_{11}} s_1 \tag{3.3}$$

(a multiple of the first row is subtracted from the i-th row), which is also computed accurately because $s_i > 0$, $s_1 > 0$, $a_{i1} < 0$ and $a_{11} > 0$, and (3.3) represents the addition of two nonnegative quantities, each of which is known accurately. Thus the row sums will start out and remain nonnegative.

This analysis implies that every quantity in the LDU decomposition of A with complete pivoting is computed to high relative accuracy:

Algorithm 3.2.1 *Given the off diagonal elements $a_{ij}, i \neq j$ and row sums $s_i = \sum_{j=1}^{n} a_{ij}$ the following algorithm computes the LDU decomposition and then the SVD of a weakly diagonally dominant M-matrix A to high relative accuracy:*

$P = I_n$ `... the permutation matrix, initially the identity matrix`

`for` $k = 1 : n-1$

  `for` $i = k : n$

    $a_{ii} = s_i - \sum_{j \neq i} a_{ij}$

  `Pivot the largest` a_{ii} `to` (k,k) `position, updating` A, P `and` s `accordingly`

  `for` $i = k+1 : n$

    `if` $a_{kk} \neq 0$

      $a_{ik} = a_{ik}/a_{kk}$

      $s_i = s_i - a_{ik} s_k$

      `for` $j = k+1 : n$

        `if` $i \neq j$

          $a_{ij} = a_{ij} - a_{ik} a_{kj}$

$a_{nn} = s_n$

L `and` DU `are stored in the lower/upper triangular part of` A `respectively`

`Call the SVD algorithm from` [22] `passing` L, D, U

□

If the matrix is exactly singular (for example in the case dealt with by the original GTH algorithm [58]) then one or more row sums s_i will initially be zero and remain exactly zero until pivoting pushes them to the end of the matrix, at which point the remaining Schur complement is exactly zero. So zero singular values may be computed exactly.

3.3 Cost

Computing the LDU decomposition of A costs n^3. The SVD algorithm of Demmel from [22] costs $O(n^3)$, thus the total cost of computing an accurate SVD of A is $O(n^3)$.

3.4 Error Analysis

Because of (3.2) and (3.3) the relative error in the row sums will not exceed the relative error in the off diagonal element at every step of Gaussian elimination. Then the error analysis in [58] goes over verbatim. The only difference between our algorithm and the GTH algorithm in [58] is that in our algorithm the diagonal elements that need to be computed at the k-th step of Gaussian elimination are the sum of $n-k+1$ quantities - the off diagonal elements and the row sums, whereas in [58] the row sums are zero and the diagonal elements are the sums of $n-k$ quantities. The author of [58] however treats the computation of the diagonal elements as being a result of $n-k+1$ arithmetic operations. Thus we can conclude that each entry of L, D and U has relative error no larger than $3n^3\epsilon$.

3.5 Perturbation Theory

It is easy to see from the above error analysis that small relative changes in the initial data (off diagonals and row sums) will cause only small relative changes in the entries of L and U. These in turn cause only small relative changes in the singular values [22]. Therefore the SVD is a well-conditioned problem.

3.6 Numerical Experiments

To test our algorithm, we randomly generated a number of very ill-conditioned matrices as follows

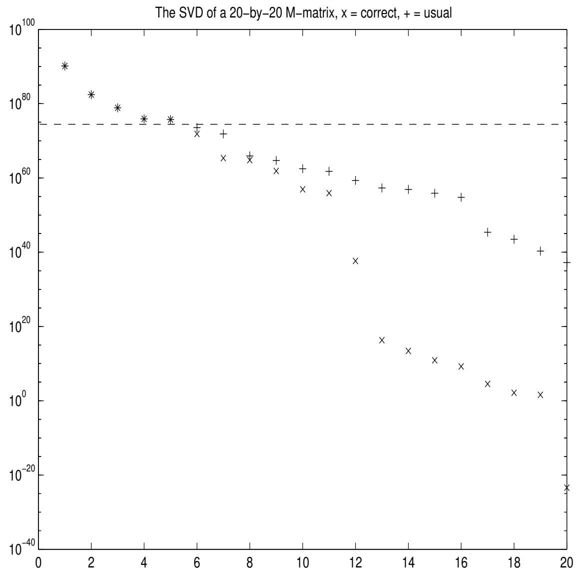
1. We chose a 20×20 matrix A with uniform random off diagonal entries between -1 and 0;

2. We chose 20 random row sums s_i of the form $r \cdot 10^k$, where r is a uniform random number in $[0, 1]$ and k is a uniform random integer in $[-40, -20]$;

3. We multiplied the i-th row of A and s_i by another random number of the form $r \cdot 10^j$, where r is again uniform random in $[0, 1]$, and j is a uniform random integer in $[-100, 100]$

The reason for step 3. is that without it, we will have one tiny singular value and the rest large, which is not much of a test. The resulting matrices had singular values, computed using our algorithm as implemented in Matlab [55], ranging over about 110 orders of magnitude. We present a typical example in Fugure 3.1.

Figure 3.1: The singular values of a 20×20 M-matrix



We also computed the singular values using 200 digit arithmetic in Mathematica [72], and every time got the same result to 14 digits. Note that we generated the random numbers in Matlab output them as decimal numbers to a file, read them into Mathematica, converted them to 200 decimal digit big floats, and then formed the explicit matrix of which we computed the SVD. In other words, we more than likely made some errors in the 16th place just in forming the matrix in Mathematica. But the tiniest singular values still agree with the Matlab results to 14 places, confirming our claims that the problem is well-conditioned, even for computing the tiniest singular values.

In contrast the traditional algorithm only computed the singular values greater than $\sigma_1\epsilon$ with some relative accuracy (the value of $\sigma_1\epsilon$ is indicated by a horizontal dotted line in the figure). We could see from our tests (and is clear from the example above) the traditional algorithm tries to respect grading [22, §4] in the sense that the the singular values that are less than $\sigma_1\epsilon$ are not clustered around $\sigma_1\epsilon$, but are orders of magnitude smaller.

To verify the perturbation theory we perturbed the off diagonal elements and the row sums of our test matrices with random noise in the 10th digit. Then computed the SVD of the thus perturbed matrices using Algorithm 3.2.1 and in Mathematica using 200 digit arithmetic. The singular values computed using each of these methods agreed with the singular values of the unperturbed matrices to 9 digits in each case confirming the fact that small relative perturbations in the initial data only cause small relative perturbations in the SVD.

Chapter 4

Accurate and Efficient Matrix Computations with Totally Positive Generalized Vandermonde Matrices using Schur Functions

We present a new algorithm for computing the Schur function $s_\lambda(x_1, x_2, \ldots, x_n)$ with small relative error when $x_i > 0$ in time that is only linear in n. With the help of this algorithm we derive a new Björck-Pereyra-type algorithm for the accurate solution of totally positive generalized Vandermonde linear systems whose cost only increases quadratically with n. As a consequence we can efficiently compute an accurate determinant, inverse and LDU decomposition of a totally positive generalized Vandermonde matrix.

We also prove that Björck-Pereyra-type methods can be used to solve any totally positive linear system whose initial minors can be computed accurately and efficiently with guaranteed small componentwise relative backward error and, depending on the right hand side, with small componentwise relative forward error.

4.1 Introduction

Outline

Our overall goal is to devise accurate and efficient algorithms for linear algebraic computations with structured matrices. We define these terms more carefully, and briefly describe our contributions.

A *structured matrix* is one whose n^2 entries depend algebraically on many fewer parameters. A typical example is a Vandermonde matrix V, where $V_{ij} = x_i^{j-1}$, which depends on n parameters $x_1, \ldots, x_n$. We may also have more structure, such as Total Positivity (each minor must be nonnegative), which for a Vandermonde is equivalent to $0 < x_1 < x_2 < \cdots < x_n$. The structured matrices we consider in detail here are Cauchy, Vandermonde, confluent Vandermonde, polynomial Vandermonde, and "generalized Vandermonde", where the exponents of the x_i are not just 0 through $n-1$, but any increasing set of nonnegative integers. We also consider the cases of totally positive matrices of each type.

The *linear algebraic computations* we consider include computing the determinant, inverse, various triangular factorizations, the SVD, and linear equation solving.

An *accurate* computation is one where each component in the answer (e.g. each entry of the inverse, or each singular value) is computed with high relative accuracy, or with many correct digits. By *efficiently* we mean that the time should be bounded by a polynomial function of the matrix dimension n, preferably $O(n^3)$, *independent* of the usual condition number $\kappa(A) \equiv \|A\| \cdot \|A^{-1}\|$. We take roundoff into account via the traditional model $fl(a \odot b) = (a \odot b)(1 + \delta)$ where $\odot$ is an arithmetic operation and $|\delta|$ is a tiny number (bounded by machine epsilon), and assume no over/underflow occurs.

It is our twin goal of accuracy and efficiency that distinguishes this work from much prior work in structured matrix computations, which endeavored to get fast algorithms (better than $O(n^3)$) with some reasonable accuracy, typically with error proportional to $\kappa(A)$.

One inspiration for our work is the Björck-Pereyra algorithm for solving totally positive Vandermonde linear systems. When used to compute the inverse, this algorithm has all the properties we want. One important thread of our work is asking to what wider class of structured matrices the ideas behind the Björck-Pereyra algorithm can be generalized.

Our first contribution is to show that the Björck-Pereyra idea can be generalized to any totally positive matrix for which there is an efficient and accurate way to evaluate *initial minors*, i.e. minors consisting of contiguous rows and columns, and containing either the first row or the first column. If this is possible, then determinants, inverses, certain triangular factorizations, certain linear system solving problems and sometimes the SVD can be computed accurately and efficiently as desired. This is discussed in section 4.3.

In particular, we apply this idea to totally positive generalized Vandermonde matrices. Generalized Vandermonde matrices $G_{ij} = x_i^{a_j}$ depend on the sequence of integer exponents $0 \le a_1 < a_2 < \cdots < a_n$ that appear. The (initial) minors of G may be expressed in terms of *Schur polynomials*, well-studied polynomial functions of $x_1, x_2, \ldots, x_n$ whose coefficients and exponents are complicated combinatorial functions of the sequence $(a_1, a_2, \ldots, a_n)$. Following the combinatorial literature, we will use the so-called *partition* $(\lambda_1, \lambda_2, \ldots, \lambda_n)$ where $\lambda_i = a_{n+1-i} - (n-i)$ instead of $(a_1, a_2, \ldots, a_n)$ to label a generalized Vandermonde matrix and its Schur function.

Our second and most important contribution is the first polynomial time algorithm for the accurate computation of these Schur functions. By this we mean that the running time is a polynomial-bounded function of $n, \lambda_1, \ldots, \lambda_n$. All previous accurate algorithms had run in time exponential in these parameters, or ran in polynomial-time but did not guarantee accuracy. This is discussed in section 4.2, and applied to matrix computations in section 4.4. This result is of interest independent of matrix computations, because it provides the first known polynomial-time algorithm (accurate or not) for evaluating other combinatorial objects, like Jack polynomials [69, 34].

Our third contribution is to show that $\kappa(A)$ is the *wrong* condition number for many of the problems we consider, and that the true condition number is much smaller. This justifies seeking the accurate algorithms that we do, because the answer is in fact accurately determined by the data, even when the data is uncertain. The true condition number is often the best possible one, namely the reciprocal of the distance to the nearest ill-posed problem, where we measure distance by the largest relative perturbation in any of the parameters defining the structured matrix (eg the x_i defining the Vandermonde). This is discussed throughout the text as each problem arises.

Our fourth contribution is the proof that some accurate computations are *impossible*. Such a statement depends strongly on our model of computation, namely the traditional model $fl(a \odot b) = (a \odot b)(1+\delta)$ mentioned above. If we instead thought of floating point numbers as represented by bit strings that we could manipulate arbitrarily, then nothing would be "impossible", though it might be very expensive. Our results are based on our prior result [25] that computing $x + y + z$ accurately for arbitrary x, y, z is impossible in the traditional model.

Our fifth contribution is the first accurate and efficient algorithm for the SVD of polynomial Vandermonde matrices, where we assume the polynomials P_j defining $V_{ij} = P_j(x_i)$ form a sequence of orthonormal polynomials. These results are presented elsewhere [24], but we summarize them below for completeness.

As outlined above, we have a large number of algorithms to discuss, one for each pair (type of structured matrix, matrix computation). To organize this information, we present Table 4.1 below summarizing the known algorithms for each such pair. Each table entry gives the complexity of the fastest known accurate algorithm, which may be polynomial, exponential, be proven not to exist, or be unknown. If the results are already known in the literature, a reference is given, and if they are new (presented in this chapter or our other papers) they are shown in boldface.

The rest of the Introduction defines the problems we want to solve more carefully, and outlines the rest of this chapter in more detail.

Types of Structured Matrices

We consider the following matrices. A *Vandermonde matrix* is a matrix of the form

$$V(x_1, x_2, \ldots, x_n) = \begin{bmatrix} 1 & x_1 & \ldots & x_1^{n-1} \\ 1 & x_2 & \ldots & x_2^{n-1} \\ & & \ddots & \\ 1 & x_n & \ldots & x_n^{n-1} \end{bmatrix}.$$

A polynomial Vandermonde matrix is defined by $V_P = [P_{j-1}(x_i)]_{i,j=1}^n$, where P_i is a polynomial of degree i. *Three-term (polynomial) Vandermonde matrices* are ones for which the polynomials P satisfy $P_0(x) = a_0, P_1(x) = d_0(x - b_0)$ and the three-term recurrence

$$P_{n+1}(x) = d_n(x - b_n)P_n(x) - c_n P_{n-1}(x), \tag{4.1}$$

where $d_i \neq 0$ for all i. When the polynomials P_i are orthogonal the matrix V_P is called a *polynomial Vandermonde matrix involving orthogonal polynomials*. A *confluent polynomial Vandermonde matrix* [44, p.429] is defined by $V_Q = [Q_0(x_1), Q_1(x_2), \ldots, Q_{n-1}(x_n)]$, where the nodes are ordered so that equal nodes are contiguous, i.e.

$$x_i = x_j,\ (i < j) \quad \Rightarrow \quad x_i = x_{i+1} = \ldots = x_j$$

and the vectors $Q_j(x)$ are defined recursively by

$$Q_j(x) = \begin{cases} [P_0(x), P_1(x), \ldots, P_{n-1}(x)]^T, & \text{if } j = 0 \text{ or } x_j \neq x_{j-1} \\ \frac{d}{dx} Q_{j-1}(x) & \text{otherwise.} \end{cases}$$

A *Cauchy matrix* is defined as

$$C = \left[\frac{f_i h_j}{x_i - y_j} \right]_{i,j=1}^n,$$

where f, h, x and y are vectors and $x_i \neq y_j$ for $i, j = 1, 2, \ldots, n$. We include Cauchy matrices, because of their close relationship with polynomial Vandermonde matrices for the purpose of finding the SVD of the latter [24, 20].

The main topic of this chapter are the *generalized Vandermonde matrices* defined as

$$G = \begin{bmatrix} x_1^{a_1} & x_1^{a_2} & \ldots & x_1^{a_n} \\ x_2^{a_1} & x_2^{a_2} & \ldots & x_2^{a_n} \\ & & \ddots & \\ x_n^{a_1} & x_n^{a_2} & \ldots & x_n^{a_n} \end{bmatrix},$$

where $0 \leq a_1 < a_2 < \ldots < a_n$ are integers. Of course, the generalized Vandermonde matrix becomes Vandermonde when $a_i = i - 1$.

We will use the notation of *partitions* to describe generalized Vandermonde matrices: The nonincreasing sequence of nonnegative integers

$$\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_n) = (a_n - (n-1), a_{n-1} - (n-2), \ldots, a_1)$$

is called a *partition* of the sum $|\lambda| = \sum_i \lambda_i$, and determines a class of generalized Vandermonde matrices sharing a common set of exponents $a_1, a_2, \ldots, a_n$. Conversely, any sequence λ of n nonincreasing nonnegative integers determines the exponents a_i of a class of generalized Vandermonde matrices. The standard Vandermonde matrix, with $a_i = i - 1,\ , i = 1, 2, \ldots, n$, has partition $\lambda = (0, \ldots, 0)$.

Model of Arithmetic

We will use a standard model of floating point arithmetic, in which we assume only that the relative error of any arithmetic operation is small:

$$fl(x \odot y) = (x \odot y)(1 + \delta) \tag{4.2}$$

where $\odot \in \{+,-,*,/\}$ and $|\delta| \leq \epsilon$ for some fixed ϵ, called *machine precision*. We also assume that no underflow or overflow occurs. This model implies that products, quotients, and sums of like-signed quantities can be computed accurately (i.e. with low relative error), but expressions involving cancellation may not be (for example, the sum of three numbers can provably not be computed accurately in this model [25] thus this is *not* a bit model, in which this computation is certainly possible). However, if x and y are inputs (and so can be considered exact) then $fl(x \pm y) = (x \pm y)(1+\delta)$ is computed accurately.

Total Positivity

For reasons that will be apparent shortly, the above floating point model leads us to consider the special class of *totally positive (TP)* matrices. A matrix is TP if all its minors are nonnegative, and *Strictly Totally Positive (STP)* if all of its minors are positive.

It is known that a (generalized) Vandermonde matrix G is STP if and only if $x_n > x_{n-1} > \cdots > x_1 > 0$. If the x_i are positive and only distinct rather than sorted, then simply permuting G's rows will make it STP. It is well known that

$$\det(V) = \prod_{n \geq i > j \geq 1} (x_i - x_j). \tag{4.3}$$

The minors of the Vandermonde matrix are generalized Vandermonde determinants, as are the minors of the generalized Vandermonde matrix. For the determinant of G we have

$$\det(G) = \det(V) \cdot s_\lambda(x_1, x_2, \ldots, x_n), \tag{4.4}$$

where $s_\lambda(x_1, x_2, \ldots, x_n)$ is a symmetric polynomial, called a *Schur function*, with positive integer coefficients that only depend on the partition λ of G. We will take a detailed look at the Schur function in Section 4.2.

Formulas (4.3) and (4.4) immediately show that G is TP when $x_n > x_{n-1} > \cdots > x_1 > 0$. They also have the following important consequences.

1. Formula (4.3) for $\det(V)$ can be evaluated accurately for any floating point data x_i, since it only involves differences of input data and products.

2. When $x_n > x_{n-1} > \cdots > x_1 > 0$ so G is TP, the positivity of the coefficients of the polynomial $s_\lambda(x_1, x_2, \ldots, x_n)$ implies that formula (4.4) for $\det(G)$ can be evaluated accurately in floating point model (4.2), since s_λ involves only products and sums

of positive quantities. In other words we have a way to accurately evaluate any minor of a TP generalized Vandermonde matrix, independent of condition number. This in turn gives us an accurate way to compute inverses and perform other matrix computations [25]. The only question is cost, which is one of the main subjects of this chapter .

Matrix computations we want to perform

We want to compute the following quantities and decompositions:

- $\det(A)$
- A^{-1} accurately componentwise
- Any minor of A
- The LDU decomposition of A, resulting from Gaussian Elimination with partial, complete or no pivoting with small relative error in each entry of L, D and U.
- The bidiagonal decomposition of A^{-1} as obtained by the process of Neville Elimination with small relative error in each entry of each factor. *Neville Elimination*, similarly to Gaussian Elimination, reduces a matrix to an upper triangular form, except that in the process of elimination, zeros are obtained by subtracting a multiple of the previous row from the current row instead of using a fixed row for elimination. The result is a bidiagonal decomposition of the inverse of the matrix [37].
- An accurate solution $\hat{x}$ to $Ax = b$ with small componentwise relative backward and/or forward error, which are defined as follows. The smallest η such that there exist $|\delta A| \leq \eta|A|$ and $|\delta b| \leq \eta|b|$ satisfying $(A + \delta A)\hat{x} = b + \delta b$ is called *componentwise relative backward error.* The smallest η such that $|x - \hat{x}| \leq \eta|x|$ is called *componentwise relative forward error.* The absolute values in both cases are meant componentwise.
- The Singular Value Decomposition (SVD) of $A = W \cdot \Sigma \cdot Z^T$, where W and Z are orthogonal, $\Sigma = \mathrm{diag}(\sigma_1, \sigma_2, \ldots, \sigma_n)$ and $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n \geq 0$ are the singular values. See [22, 20] for what relative accuracy means when computing the SVD; it includes computing each singular value with small relative error.

Goals

Our goal is to *accurately and efficiently* perform matrix computations on a matrix A from one of the above matrix classes. By *accurately* we mean that the relative error in the computed result should be small. In particular this means that zeros must be computed exactly. By *efficiently* we mean that the cost must be bounded by a polynomial function of n, preferably $O(n^3)$, *independent of the condition number* $\kappa(G) = \|G\| \cdot \|G^{-1}\|$.

Means for Achieving the Goals

We can compute an accurate determinant, inverse and LDU decomposition of a matrix A if we can compute accurate minors of A as is discussed later in this Introduction.

Solving linear systems accurately is another matter and the relationship with minors is a lot less obvious. TP Cauchy or Vandermonde linear systems can be solved very accurately using the Björck-Pereyra methods [8] or generalizations thereof [12]. These methods have the properties:

- they produce a bidiagonal decomposition of the inverse of the matrix
- each bidiagonal factor has a checkerboard sign pattern, i.e. $\text{sign}(B_{ij}) = (-1)^{i+j}$ or 0.
- each entry of every bidiagonal factor is computed with small relative error

Those properties are attractive for the following reasons:

- The inverse can be formed by forming the product of all bidiagonal factors and the checkerboard sign pattern ensures than no subtractive cancellation occurs in the process, i.e. the inverse is computed accurately componentwise.
- Linear systems with right hand sides that have alternating sign pattern can also be solved accurately, again, because no subtractive cancellation occurs when forming the solution.

Thus we are motivated to ask:

Q1) Which matrices allow for bidiagonal decomposition of their inverses?

Q2) When do the bidiagonal factors have checkerboard sign pattern?

Q3) Of the above matrices, when can this decomposition be computed accurately?

We answer those questions as follows:

A1) All matrices with nonsingular leading minors have bidiagonal decomposition of their inverses. This bidiagonal decomposition is a direct result of the process of Neville elimination.

A2) The bidiagonal factors have checkerboard sign pattern if and only if the matrices are TP.

A3) Each entry of each bidiagonal factor equals the negated product of two quotients of *initial minors* of the matrix. (The minors are initial if they are contiguous and include the first row or the first column). Thus the question of computing a bidiagonal decomposition of a TP matrix reduces to the question of accurate computation of initial minors. For Cauchy and Vandermonde matrices the initial minors are very easy to compute accurately.

All this theory is covered in Section 4.3.

Therefore, in order to perform accurate computations with generalized Vandermonde matrices we need to be able to compute accurate initial minors. This in turn reduces to the question of accurate computation of the Schur function which we discuss in Section 4.2. We present a new algorithm for the accurate computation of the Schur function which is exponentially faster than previous such algorithms. Based on this algorithm we design Björck-Pereyra-type algorithms for TP generalized Vandermonde systems in section 4.4.

Perturbation Theory

The reader may wonder why we want to compute accurate minors (and then inverses, singular values, etc.) of very ill-conditioned matrices, since small inherent uncertainties in the data x_k might utterly change the answer. In fact formula (4.4) tells us that the minors of a TP G can be arbitrarily less sensitive to small relative changes in the x_k than the usual condition number $\kappa(G)$ tells us; here is why. The positivity of the coefficients of $s_\lambda(x_1, \ldots, x_n)$ implies that small relative changes in any $x_k > 0$ can only cause small relative changes in s_λ (possibly magnified by the degree of s_λ); in other words s_λ is nearly perfectly conditioned. So the only real sensitivity can arise from the factor $\prod_{n\geq i>j\geq 1}(x_i - x_j)$. It is easy to see that a relative change of $\eta \ll 1$ in x_k can cause a relative change in

$\prod_{n\geq i>j\geq 1}(x_i - x_j)$ of at most about

$$\eta \cdot \sum_{i\neq k} \frac{|x_i| + |x_k|}{|x_i - x_k|} \quad . \tag{4.5}$$

In other words, when there is a reasonably large *relative gap* $\text{rel\_gap}_{ik} = \frac{|x_i - x_k|}{|x_i|+|x_k|}$ between every pair x_i and x_k, then every minor of G (and consequently the inverse, singular values, etc.) are well-conditioned functions of the data, and deserve to be computed accurately. The only alternatives we know of to the algorithms in this chapter can involve arbitrary precision arithmetic, and so arbitrarily higher cost.

Since a Vandermonde matrix is singular only when $x_i = x_j$ for some i and j, the minimum relative gap rel_gap_{ik} is the "relative distance" to the nearest singular Vandermonde matrix and because of (4.5) the reciprocal of that quantity can be thought of as the "structured" condition number of the Vandermonde matrix. See also [18].

State of Knowledge - Table 4.1

Table 4.1 summarizes our current state of knowledge, including the contributions of this chapter . There is a row for each matrix type and a column for each matrix operation. A table entry describes the complexity of the best known algorithm for performing the column matrix operation on the row matrix type accurately, independent of the condition number. In other words, we consider only algorithms that work in floating point model (4.2), and do not use precision independent of condition number.

For all algorithms except the SVD, the worst case error bound of the algorithms are of the form $O(\text{poly}(n)\epsilon)$, where $\text{poly}(n)$ is a polynomial in n and ϵ. Thus to guarantee a relative error less than some $0 < \zeta < 1$ requires that $\epsilon \approx \frac{\zeta}{\text{poly}(n)}$ or $b = \log_2 \frac{1}{\epsilon} \approx \log_2 \frac{1}{\zeta} + O(\log_2 n)$ bits of precision. If we fixed ζ and assumed the cost of arithmetic with b-bit numbers were $O(b^2)$ (the best known cost is $O(b \log b \log\log b)$ [65]) then the cost of the algorithm would be multiplied by $O(\log_2^2 n)$ a factor we omit from Table 4.1. For the SVD the error bound includes the factor $\kappa(L)\kappa(U)$, where L and U are unit triangular factors, resulting from GECP. These are typically quite small, but could conceivably be as large as 2^n, though we know of no such example. This would require $b = \log_2 \frac{1}{\zeta} + O(n)$ bits of precision, and raise the cost of the algorithm by $O(n^2)$. But this highly unlikely event would still not make the difference between an algorithm running in polynomial time or not.

The entry EXP means that the best algorithms we know of in the model (4.2) run in time that may grow exponentially in n [25]; in these cases one is in practice better off resorting to a conventional $O(n^3)$ algorithm running in high precision arithmetic, where the precision needed to compute an accurate answer depends on the condition number. The entry "?" means that we know of no accurate algorithm for this problem that does not rely on high precision arithmetic (or some simulation thereof [60]) where the number of digits grows proportionally to the logarithm of the condition number $\kappa(G)$. The entry "No" means that no accurate algorithm can exist in this floating point model.

Discussion of Table 4.1 by columns

The Determinant

We can compute the determinant accurately if we have formulas (Cauchy, Vandermonde, TP Generalized Vandermonde) that can be evaluated accurately or if Gaussian or Neville elimination can be performed without losing accuracy.

The Inverse

The inverse of a matrix can be computed accurately but not necessarily efficiently by using Cramer's rule, if accurate minors can be computed (e.g. Cauchy, TP Vandermonde). For TP matrices an accurate inverse can be computed by first using a Björck-Pereyra-type method to compute an accurate bidiagonal decomposition of the inverse (see the column marked NENP) and then computing the product of the factors, which can be formed accurately because of the checkerboard sign pattern of the factors and the inverse. See section 4.5. For polynomial Vandermonde matrices involving orthogonal polynomials that satisfy additional conditions a similar decomposition is available: the factors, although not bidiagonal, still have checkerboard sign patterns and their product can be formed accurately to produce an accurate inverse [43, 44].

Gaussian Elimination

The acronyms GENP, GEPP and GECP stand for Gaussian elimination with no pivoting, partial pivoting, and complete pivoting, respectively. The approach in computing the LDU factors accurately is to exploit the fact that each entry of L, D and U is a quotient of minors (see section 4.7).

SVD

An accurate LDU decomposition from GECP implies we can compute an accurate SVD [22]. (Polynomial) Vandermonde matrices can be converted to Cauchy matrices for which the SVD is easy to compute [20, 24].

Neville Elimination

NENP stands for the diagonal decomposition of the inverse obtained by Neville Elimination with no pivoting. It is equivalent to the existence of Björck-Pereyra-type methods and is only of interest for TP matrices.

Solving $Ax = b$

In the last two columns we include references to algorithms for accurate solution of linear systems with small componentwise relative backward error [19]

$$\max_i \frac{|A\hat{x} - b|_i}{(|A||\hat{x}| + |b|)_i}$$

and small forward error of the form

$$|x - \hat{x}| \leq O(\epsilon)|A^{-1}||b| \tag{4.6}$$

respectively. The inequality (4.6) is equivalent to small componentwise relative forward error when A is TP and b has alternating sign pattern [43]. A system can be solved with small forward error if an accurate inverse is available and can be solved with small backward error if A is TP and an accurate bidiagonal decomposition of A^{-1} exists [43].

Discussion of Table 4.1 by rows

Cauchy

We start out with Cauchy matrices, which are TP if $x_1 > x_2 > \ldots > x_n > y_1 > y_2 > \ldots > y_n$. All results for Cauchy matrices are based on the fact that all minors are easy to compute accurately, which implies algorithms for accurate computation of the inverse (§4.5), LDU (section 4.7) and the SVD [20]. Cauchy linear systems can be solved very accurately using Björck-Pereyra-type methods [12].

Vandermonde

Vandermonde systems have accurate formula for their determinants (4.3). Their SVD can be computed accurately by first converting them to Cauchy matrices by postmultiplication by the orthogonal matrix of the Discrete Fourier Transform [20, 24]. There are a number of quantities that are impossible to compute in this particular model of arithmetic because their computation would require the accurate computation of a sum of three numbers, which is impossible in this model [25]. For example for a 4×4 Vandermonde matrix V we have $(V^{-1})_{31} = \frac{x_2+x_3+x_4}{(x_2-x_1)(x_3-x_1)(x_4-x_1)}$ and since the denominator can always be computed accurately, the question of computing $(V^{-1})_{31}$ accurately is equivalent to the question of adding three numbers accurately, which we know is impossible, unless $x_i > 0$, which is the case when V is TP. All other "No" entries in the table are there because a sum of three numbers may have to be computed in the process. In the TP case Schur functions can be used to compute the inverse, and LDU decompositions through the computation of minors. TP Vandermonde systems can be solved using the Björck-Pereyra methods [8]. Cramer's rule is, we believe, the fastest way to compute an accurate inverse of a Vandermonde matrix, beating the algorithm for computing the inverse using the Björck-Pereyra methods by a factor of 4.8. See section 4.6 for details.

Three-term Vandermonde matrices involving orthogonal polynomials

Their determinants are easy to compute thanks to the explicit decomposition [44, p.435]. Similarly to the ordinary Vandermonde matrices these matrices can be converted into Cauchy matrices for the purpose of computing the SVD [24].

The three-term Vandermonde matrices (4.1) for which $0 < x_1 < x_2 < \ldots < x_n$ and $d_i > 0, c_i \geq 0$ for all i, and $b_i - x_k \leq 0$ for all $i+k \leq n-1$ are the "Three-term Vandermonde matrices involving orthogonal polynomials with other conditions" mentioned in the table. The paper [43] includes algorithms and analysis on how to do matrix computations with these matrices.

Confluent Vandermonde

The methods in [43] also apply to confluent Vandermonde matrices, which explains the entries for confluent Vandermonde matrices in the table.

Generalized Vandermonde

The generalized Vandermonde matrices are the main topic of this chapter . Unless the matrix is TP there is nothing we can do accurately. In the TP case the results in the table, along with the respective algorithms are discussed throughout this chapter . The main tool is computing accurate minors and in the process the accurate computation of the Schur function.

New Results

The new results that we report in this chapter are:

- New algorithm for computing the Schur function $s_\lambda(x_1, x_2, \ldots, x_n)$ accurately, where $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_p)$. The cost of our algorithm can be bounded by

$$O\left(n(\lambda_1+1)(\lambda_2+1)^2 \ldots (\lambda_p+1)^2 p\right). \tag{4.7}$$

  When p is of order n and $\lambda_p > 0$ this can still be an exponential function of n. But the actual cost in this case can still be exponentially less than (4.7) , as discussed in section 4.2. Our algorithm is also exponentially faster than our previous best algorithm for computing the Schur function [25], whose cost was bounded by

$$O\left(n(\lambda_1+1)^{\log_2 n}(\lambda_2+1)^{\log_2 n} \ldots (\lambda_p+1)^{\log_2 n}\right)$$

  and is also exponentially faster than the traditional algorithm for computing the Schur function [51] which costs as much as

$$O\left(n^{\lambda_1+\lambda_2+\ldots+\lambda_p}\right).$$

- We present a detailed analysis on the conditions under which Björck-Pereyra-type algorithms can exist for the accurate solution of TP linear systems. Namely, if we can compute accurate initial minors, we can compute accurate bidiagonal decompositions of the inverses of the corresponding linear systems which allows for the accurate computation of the solutions of the linear systems.

  We combine this theory along with our new algorithm for computing the Schur function to obtain Björck-Pereyra-type algorithms for solving TP generalized Vandermonde matrices.

- We present an algorithm for accurate inversion of Vandermonde matrices which is faster than the previous best algorithm, based on the Björck-Pereyra methods, by a factor of 4.8.

- We show how to compute an accurate LDU decomposition resulting from GENP or GEPP of (generalized) Vandermonde matrices in time that only depends on n^2 instead of n^3.

- In the floating point model (4.2) there are quantities that cannot be computed accurately. All these apply to quantities whose computation would involve the sum of at least three numbers not all of which will have the same sign, which was proven to be impossible in [25].

Organization of this chapter

- In Section 4.2 we cover all definitions and facts about partitions, Young diagrams, Young tableaux and present our main result, the algorithm for computing the Schur function accurately and efficiently.

- In Section 4.3 we give a simpler proof of the known fact that each entry of the bidiagonal decomposition of the inverse of a TP matrix is a (negated) product of two quotients of minors.

- In Section 4.4 we use the algorithm for computing the Schur function to obtain Björck-Pereyra-type algorithms for generalized Vandermonde matrices.

- In Section 4.5 we review the use of the Cramer's rule for inverting Cauchy and TP (generalized) Vandermonde matrices.

- In Section 4.6 we derive an algorithm for inverting a Vandermonde matrix accurately in $\frac{5}{12}n^3$ time, which is about 4.8 times faster than the previous best accurate inversion algorithm based on the Björck-Pereyra methods.

- In Section 4.7 we describe how to compute an accurate LDU decomposition for TP generalized Vandermonde matrices.

- We present the results of our numerical experiments in Section 4.8

Table 4.1: State of knowledge

Type of Matrix	$\det(A)$	A^{-1}	Any minor	GE NP	GE PP	GE CP	SVD	NE NP	$Ax=b$ F[4]	$Ax=b$ B[4]
Cauchy	n^2 §4.5	n^2 §4.5	n^2 §4.5	n^3 [20]	n^3 [20]	n^3 [20]	n^3 [20]	n^2 [12]	n^2 [12]	?
TP Cauchy	n^2 §4.5	n^2 §4.5	n^2 §4.5	n^3 [20]	n^3 [20]	n^3 [20]	n^2 [20]	n^2 [12]	n^2 [12]	n^2 [12]
Vandermonde	n^2 §4.1	**No**	**No**	**No**	**No**	**No**	$\mathbf{n^3}$ [20, 24]	n^2 [8]	**No**	?
TP Vandermonde	n^2 §4.1	$\mathbf{n^3}$ **§4.5,[43]**	**exp** **§4.2**	n^2 **§4.7**	n^2 **§4.7**	**exp** **§4.7**	$\mathbf{n^3}$ [20, 24]	n^2 [8]	n^2 [43]	n^2 [43]
Confluent Vandermonde	n^2 [43]	**No**	**No**	**No**	**No**	**No**	?	n^2 [43]	**No**	?
TP Confluent Vandermonde	n^2 [43]	n^3 [43]	?	n^3 §4.7	?	?	?	n^2 [43]	n^2 [43]	n^2 [43]
Three-term Vandermonde Orth. Poly.	n^2 [43]	?	?	?	?	?	$\mathbf{n^3}$ **[24]**	?	?	?
3-term Vand. Orth. Poly. + other cond.[7]	n^2 [43]	n^3 [43]	?	?	?	?	$\mathbf{n^3}$ **[24]**	?	n^2 [43]	?
Generalized Vandermonde	**No**	**No**	**No**	**No**	**No**	**No**	**No**	**No**	**No**	?
TP generalized Vandermonde[6]	$\mathbf{\Lambda n + n^2}$ **§4.2**	$\mathbf{\Lambda n^2 + n^3}$ **§4.4,4.5**	**exp**[5]	$\mathbf{\Lambda n^2}$ **§4.7**	$\mathbf{\Lambda n^2}$ **§4.7**	**exp** **§4.7**	**exp** **§4.7**	$\mathbf{\Lambda n^2}$ **§4.4**	$\mathbf{\Lambda n^2}$ **§4.4**	$\mathbf{\Lambda n^2}$ **§4.4**

1) Each entry is meant in a big-O sense, i.e. n^2 means $O(n^2)$.

2) Results that are new or are improvements of known results are in **bold**.

3) Below each entry is a reference to where the result (old or new) appears, which may be a section in this chapter (e.g. "§3" means Section 3) or a reference or both.

4) The last two columns stand for the cost of solving $Ax = b$. The column marked "F" means the solution must satisfy a *Forward* error bound of the form $|x - \hat{x}| \leq O(\epsilon)|A^{-1}||b|$, implying $|x - \hat{x}| \leq O(\epsilon)|x|$, when x has alternating sign pattern and A is TP. The column marked "B" means the solution must have small componentwise relative *Backward* error, i.e. $|A - \hat{A}| \leq O(\epsilon)|A|$.

5) The partition of a minor can be $O(n)$ larger than the partition corresponding to the generalized Vandermonde matrix. Even though the cost is exponential, with the new algorithm for the computing the Schur function the cost is exponentially lower than the previous best algorithm.

6) The constant Λ depends on the partition $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_p)$, $p \leq n$ that corresponds to the generalized Vandermonde matrix, but not on n. A bound is $\Lambda \leq 2(\lambda_1 + 1)(\lambda_2 + 1)^2 \ldots (\lambda_p + 1)^2 p$, which may be exponentially larger than the actual cost. See Section 4.2 for details.

7) In addition to (4.1) we must also have $0 < x_1 < x_2 < \ldots < x_n$ and $d_i > 0, c_i \geq 0$ for all i, and $b_i - x_k \leq 0$ for all $i + k \leq n - 1$.

- Section 4.9 draws conclusions and states open problems.
- Finally, we present a Matlab implementations of the algorithms in this chapter in Section 4.10.

Notation

In what follows we will denote vectors $(x_i, x_{i+1}, \ldots, x_j)$ as $(x_{i:j})$ or $(x(i : j))$. When the set of indices is not contiguous we will include the set of indices in brackets, i.e. $(x_i, x_{i+1}, \ldots, x_j, x_n)$ will be denoted by $(x_{[i:j,n]})$ or $(x([i : j, n]))$. Similarly we will denote a minor of matrix that includes rows i through j and columns k through l, where $i \le j; k \le l$ as $A_{i:j,k:l}$ or $A(i : j, k : l)$. We will use brackets for the range of row or column indices that are not contiguous.

4.2 Young Diagrams, Young Tableaux and the Schur Function

In this section we introduce Young diagrams and tableaux and present a recursive algorithm for the computation of the Schur function. All facts and theory leading up to (4.12) are proven in a more general setting in [54]. To make this presentation easier to understand we repeat the main facts here and give simple proofs. We emphasize that our main goal is to compute the Schur function with low relative error. To achieve that goal we are only interested in a subtraction-free algorithm, meaning that we cannot use (4.4) or the Jacobi-Trudi identity [54, (3.4), p.41]. Thus we develop Algorithm 4.2.1 whose cost given in Theorem 5.

Overview of Definitions and Known Properties of Partitions, Diagrams and the Schur Function

For a partition $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_p)$ of $|\lambda| = \lambda_1 + \lambda_2 + \ldots + \lambda_p$, where $\lambda_1 \ge \ldots \ge \lambda_p \ge 0$ we call $|\lambda|$ the *weight* of the partition and the non zero λ_i *parts* of the partition. We identify two partitions if they differ by a string of zeros at the end. For example $\lambda = (4, 2, 1) = (4, 2, 1, 0) = (4, 2, 1, 0, 0)$. It is also convenient to indicate the number of times each integer occurs in the partition and to write for example $(3, 2^2, 1^4)$ instead of $(3, 2, 2, 1, 1, 1, 1)$.

The *Young diagram* of a partition $\lambda = (\lambda_1, \ldots, \lambda_p)$, may be formally defined as the set of knots or points (i, j), where i, j are integers such that $1 \leq j \leq \lambda_i$, $i = 1, \ldots, p$. We denote the diagram of a partition λ by D_λ. Graphically we draw the diagrams replacing the knots by square boxes, adopting the convention, as with matrices, that the first coordinate i (the row index) increases as one goes downwards, and the second coordinate j (the column index) increases as one goes from left to right. The first boxes from the left of each row are one above another and the i-th row contains λ_i boxes. For example, for $\lambda = (4, 2, 1)$, the corresponding Young diagram is:

$$D_\lambda = D_{(4,2,1)} = \ydiagram{4,2,1}$$

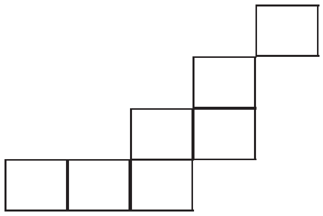
We also introduce ordering among the partitions and Young diagrams. For the partitions $\lambda = (\lambda_1, \lambda_2, \ldots)$ and $\mu = (\mu_1, \mu_2, \ldots)$ we say that $\lambda \leq \mu$ if $\lambda_i \leq \mu_i$ for all $i = 1, 2, \ldots$. We also say that $\lambda < \mu$ if $\lambda \leq \mu$ and $\lambda \neq \mu$. In the language of diagrams $\lambda \leq \mu$ means that diagram of λ is contained in the diagram of μ.

For a partition $\lambda = (\lambda_1, \ldots, \lambda_p)$ of $|\lambda|$, we define a λ-*tableau* T_λ *of content* $\alpha = (\alpha_1, \ldots, \alpha_s)$, where $\alpha_1 + \ldots + \alpha_s = |\lambda|$, as the Young diagram of λ, whose boxes are filled in with the numbers $1, 2, \ldots, s$ and the number 1 occurs α_1 times, the number 2 occurs α_2 times, ..., the number s occurs α_s times. The tableau is *semistandard* if its entries do not decrease from left to right in the rows and increase from top to bottom in the columns. The tableau T_λ is *standard* if it is semistandard and of content $(\underbrace{1, 1, \ldots, 1}_{|\lambda|})$, i.e. every integer $1, 2, \ldots, |\lambda|$ occurs in it exactly once. For example, the first two tableaux below are, respectively, semistandard of content $(2, 3, 1, 1)$ and standard, and the third is not semistandard.

$$\begin{ytableau} 1 & 1 & 2 & 4 \\ 2 & 2 \\ 3 \end{ytableau} \qquad \begin{ytableau} 1 & 3 & 6 & 7 \\ 2 & 4 \\ 5 \end{ytableau} \qquad \begin{ytableau} 1 & 2 & 3 & 4 \\ 6 & 5 \\ 7 \end{ytableau}$$

If λ and μ are partitions and $\mu \leq \lambda$ then the set-theoretic difference $\theta = \lambda - \mu$, which we will denote by λ/μ is called a *skew partition*. Skew partitions correspond to *skew diagrams* in which we align the diagrams of λ and μ at the upper left box and erase the boxes belonging to μ from the diagram of λ, obtaining a skew diagram for the skew-partition $\theta = \lambda/\mu$. We denote the skew diagram corresponding to $\theta = \lambda/\mu$ by $D_{\lambda/\mu}$ or just D_θ. For example if $\lambda = (5, 4^2, 3)$ and $\mu = (4, 3, 2)$ the skew diagram of λ/μ is:

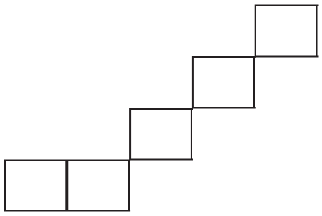
$$D_{\lambda/\mu} = D_{(5,4^2,3)/(4,3,2)} =$$



Of course, for any partition λ we have $\lambda = \lambda/(0)$ where (0) is the zero partition.

The skew diagram of a skew partition θ is a *horizontal m-strip* if $|\theta| = m$ and it has at most one square in each column. For example:

$$D_{(5,4,3,2)/(4,3,2)} =$$



is a horizontal strip. A necessary and sufficient condition for λ/μ to be a horizontal strip is that the sequences λ and μ be interlaced:

$$\lambda_1 \geq \mu_1 \geq \lambda_2 \geq \mu_2 \geq \ldots . \tag{4.8}$$

Definition 4.2.1 *A* Schur function *corresponding to a partition λ is defined as:*

$$s_\lambda(x_1, x_2, \ldots, x_m) = \frac{\det\left(\left[x_i^{j-1+\lambda_{m-j}}\right]_{i,j=1}^m\right)}{\det\left(\left[x_i^{j-1}\right]_{i,j=1}^m\right)} \tag{4.9}$$

We have the determinant of a generalized Vandermonde matrix corresponding to a partition λ in the numerator and the determinant of an ordinary Vandermonde matrix in the denominator.

Theorem 4 *The Schur function corresponding to a partition λ satisfies:*

$$s_\lambda(x_1, x_2, \ldots, x_m) = \sum W_m^{(n_1,n_2,\ldots,n_m)} x_1^{n_1} x_2^{n_2} \ldots x_m^{n_m}, \tag{4.10}$$

where $W_m^{(n_1,n_2,\ldots,n_m)}$ *is the number of semistandard* λ*-tableaux of content* $(n_1, n_2, \ldots, n_m)$ *and the summation is over all* m*-tuples* $(n_1, n_2, \ldots, n_m)$ *for which* $W_m^{(n_1,n_2,\ldots,n_m)}$ *is nonzero.*

For a proof see the proof of (5.12) in [54, p.73].

Example 4.2.1 *Let* $m = 3$ *and* $\lambda = (2, 1)$*. The semistandard* $(2, 1)$ *tableaux of content* (n_1, n_2, n_3) *for* $n_1 + n_2 + n_3 = m = 3$*,* $n_1 \geq n_2 \geq n_3$*, are*

$(n_1, n_2, n_3) = (1, 1, 1)$ $\qquad$ $(n_1, n_2, n_3) = (2, 1, 0)$

$$\begin{array}{|c|c|}\hline 1 & 2 \\ \hline 3 & \multicolumn{1}{c}{} \\ \cline{1-1}\end{array} \qquad \begin{array}{|c|c|}\hline 1 & 3 \\ \hline 2 & \multicolumn{1}{c}{} \\ \cline{1-1}\end{array} \qquad\qquad \begin{array}{|c|c|}\hline 1 & 1 \\ \hline 2 & \multicolumn{1}{c}{} \\ \cline{1-1}\end{array}$$

Since $s_{(2,1)}(x_1, x_2, x_3)$ *is a symmetric polynomial, we obtain*

$$s_{(2,1)}(x_1, x_2, x_3) = 2x_1x_2x_3 + x_1^2x_2 + x_1x_2^2 + x_1^2x_3 + x_1x_3^2 + x_2^2x_3 + x_2x_3^2.$$

Alternatively using (4.4) we have

$$\begin{aligned} s_{(2,1)}(x_1, x_2, x_3) &= \det\left(\begin{bmatrix} 1 & x_1^2 & x_1^4 \\ 1 & x_2^2 & x_2^4 \\ 1 & x_3^2 & x_3^4 \end{bmatrix}\right) \cdot \det\left(\begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \end{bmatrix}\right)^{-1} \\ &= \frac{(x_1^2 - x_2^2)(x_1^2 - x_3^2)(x_2^2 - x_3^2)}{(x_1 - x_2)(x_1 - x_3)(x_2 - x_3)} = (x_1 + x_2)(x_1 + x_3)(x_2 + x_3), \end{aligned}$$

which gives the same answer.

Example 4.2.2 *Easy examples of Schur functions include the* complete symmetric polynomial *of degree* n

$$s_{(n)}(x_1, x_2, \ldots, x_m) = \sum_{n_1+\ldots+n_m=n,\ n_i \geq 0} x_1^{n_1} \ldots x_m^{n_m}$$

and the elementary symmetric polynomial *of degree* $n \leq m$

$$s_{(1^n)}(x_1, \ldots, x_m) = \sum_{1 \leq i_1 < \ldots < i_n \leq m} x_{i_1} \ldots x_{i_n}. \tag{4.11}$$

The Schur function can be computed in a number of ways. The expression (4.4) is one, which only costs $O(n^3)$, but subtractive cancellation ruins the relative accuracy, and this is in fact where we started in the first place. The only reason why we need to compute the Schur function accurately is because we cannot compute $\det(G)$ accurately using traditional algorithms. Another $O(n^3)$ way to compute the Schur function is to use the

Jacobi-Trudi identity [54], but it, too, involves evaluating determinants, thus subtractive cancellation and loss of accuracy. We can compute the Schur function with guaranteed accuracy by using (4.10) (we only multiply and add positive numbers), but at exponential cost $O\left(n^{|\lambda|}\right)$ [25, 47].

Therefore we take a different approach. We will compute the Schur function as a polynomial, thus guaranteeing accuracy, but we will not form it explicitly to avoid the exponential cost. The key to the efficient computation of the Schur function is to rewrite (4.10) in terms of Schur functions corresponding to smaller partitions, which can then be computed recursively.

The expression (4.10) can be formed by first forming all semistandard λ-tableaux filled in all possible ways with the numbers $1, 2, \ldots, n$, form a monomial for each tableau and take their sum.

Let T be a semistandard Young Tableaux filled with the numbers $1, 2, \ldots, n$. The numbers in the columns of T need to strictly increase from top to bottom, therefore the number n can only appear in a "bottom" box of T, i.e. a box that has no adjacent boxes underneath – for example one of the boxes with X's in them in the following example.

$$\begin{array}{|c|c|c|c|}
\hline
 & & \mathrm{X} & \mathrm{X} \\
\hline
 & \mathrm{X} \\
\cline{1-2}
\mathrm{X} \\
\cline{1-1}
\end{array}$$

The set of boxes filled with the number n forms a horizontal strip. If we remove this set of boxes from λ we obtain a new tableau $\mu \leq \lambda$ filled with the numbers $1, 2, \ldots, n-1$. On the other hand for every tableau $\mu \leq \lambda$, filled with the numbers $1, 2, \ldots, n-1$, where λ/μ is a horizontal strip, we can fill the boxes of λ/μ with the number n obtaining a λ-tableaux filled with the numbers $1, 2, \ldots, n$. Now using (4.10) we obtain

$$s_\lambda(x_1, x_2, \ldots, x_n) = \sum_\mu s_\mu(x_1, x_2, \ldots, x_{n-1}) x_n^{|\lambda/\mu|}, \tag{4.12}$$

where the summation is over all $\mu \leq \lambda$ such that λ/μ is a horizontal strip. For an alternative proof of (4.12) see [54, p.71-72]. Analogously, if $\lambda_2 < \lambda_1$ and $\mu' = (\lambda_1 - 1, \lambda_2, \lambda_3, \ldots)$ then

$$s_{\mu'}(x_1, \ldots, x_n) = \sum_\mu s_\mu(x_1, x_2, \ldots, x_{n-1}) x_n^{|\mu'/\mu|}, \tag{4.13}$$

where the summation is over all $\mu \leq \mu'$ such that μ'/μ is a horizontal strip.

Derivation of the Main Algorithm

All results up to here can be found in [54]. We now proceed with the derivation of the new results. Substituting (4.13) into (4.12) we obtain

$$s_\lambda(x_1, x_2, \ldots, x_n) = \begin{cases} \sum_\mu s_\mu(x_1, x_2, \ldots, x_{n-1}) x_n^{|\lambda/\mu|}, & \text{if } \lambda_2 = \lambda_1 \\ s_{\mu'}(x_1, x_2, \ldots, x_n) x_n + \sum_\mu s_\mu(x_1, x_2, \ldots, x_{n-1}) x_n^{|\lambda/\mu|}, & \text{otherwise} \end{cases} \tag{4.14}$$

where in both cases the summation is over all partitions $\mu \leq \lambda$ such that $\mu_1 = \lambda_1$ and λ/μ is a horizontal strip. The number of terms in (4.14) is clearly not larger than the number of terms in (4.12). As we will see later this will become the reason for the overall cost of computing the Schur function to depend only linearly on λ_1 instead of quadratically. The recursion in (4.14) will terminate in the following cases:

1. $s_{(0)}(x_1, \ldots, x_i) = 1$
2. $s_{(1)}(x_i) = x_i$
3. $s_\mu(x_1, \ldots, x_i) = 1$ for $i = 0$ by definition.

We will also use this obvious optimization, although for simplicity we will not account for it in the complexity analysis of the algorithm.

$$s_{(\mu_1, \ldots, \mu_i)}(x_1, x_2, \ldots, x_i) = (x_1 \ldots x_i)^{\mu_i} \cdot s_{(\mu_1 - \mu_i, \ldots, \mu_i - \mu_i)}(x_1, x_2, \ldots, x_i). \tag{4.15}$$

In the rest of this chapter we will explain how to implement formula (4.14) recursively and account for the cost of doing so.

We present our main algorithm in two steps. First we will informally explain how to implement (4.14). We will then discuss the computational issues and the reasons we had to make our main algorithm seemingly complicated, although it is nothing else than a recursive computation according to the identity (4.14) and storing in memory and reusing if necessary all results that have been computed at the lower levels of recursion.

Before we implement (4.14) we discuss how to implement (4.12). We need to form all partitions μ such that λ/μ is a horizontal strip. Because of (4.8) μ_1 can take all values $\lambda_2, \lambda_2 + 1, \ldots, \lambda_1$; μ_2 can take all values $\lambda_3, \lambda_3 + 1, \ldots, \lambda_2$; and so on. Thus (4.12) can be rewritten as

$$s_\lambda(x_1, x_2, \ldots, x_n) = \sum_{\mu_1=\lambda_2}^{\lambda_1} \sum_{\mu_2=\lambda_3}^{\lambda_2} \cdots \sum_{\mu_{p-1}=\lambda_p}^{\lambda_{p-1}} \sum_{\mu_p=0}^{\lambda_p} s_\mu(x_1, x_2, \ldots, x_{n-1}) x_n^{|\lambda/\mu|}. \tag{4.16}$$

Now (4.14) becomes

$$s_\lambda = \begin{cases} \sum_{\mu_2=\lambda_3}^{\lambda_2} \cdots \sum_{\mu_{p-1}=\lambda_p}^{\lambda_{p-1}} \sum_{\mu_p=0}^{\lambda_p} s_\mu(x_{1:n-1}) x_n^{|\lambda/\mu|}, & \text{if } \lambda_2 = \lambda_1 \\ s_{\mu'}(x_{1:n}) x_n + \sum_{\mu_2=\lambda_3}^{\lambda_2} \cdots \sum_{\mu_{p-1}=\lambda_p}^{\lambda_{p-1}} \sum_{\mu_p=0}^{\lambda_p} s_\mu(x_{1:n-1}) x_n^{|\lambda/\mu|}, & \text{otherwise} \end{cases} \tag{4.17}$$

Here is pseudocode for implementing (4.17)

```
10  Allocate space s((0)..λ, 1..n) and initialize with −1;
                    (s_μ(x_1, ..., x_i) will be stored in s(μ, i))
20  function Schur(λ, n)
30  if s(λ, n) ≥ 0 return it
40  else
50     if λ_2 = λ_1 then s(λ, n) = 0
60     else  s(λ, n) = s((λ_1 − 1, λ_2, ...), n) · x_n
70     for μ_2 = λ_3 to λ_2
80     ...
90        for μ_p = 0 to λ_p
100           s(λ, n) = s(λ, n) + Schur(μ, n − 1) · x_n^(|λ|−|μ|)
110 store s(λ, n)
```

We could compute the Schur function without storing and reusing $s(\lambda, n)$ (lines 30 and 110 above), but the recursive computation would then reproduce the explicit Schur function polynomial (4.10) and the cost would be exponential. By storing and reusing the computed Schur functions corresponding to smaller partitions we are able to compute the Schur function at a cost that we will see later to be only linear in the number of variables n, which is optimal.

There are several implementation issues that need to be addressed for the above algorithm to work in practice. Clearly we cannot have nested loops whose number depends on a parameter, we must use recursion. Passing each partition $\mu = (\mu_1, \mu_2, \ldots, \mu_p)$ as a p-vector will increase the cost of each recursive call by a factor of p. Also, we will need a substantial amount of stack space to store all partitions that have been passed recursively. Thus, ideally we would not have to pass entire partitions, but index each one, so we can pass partition indexes instead of entire partitions. Since the partitions $(0), \ldots, \lambda$ are not in an easy correspondence with their consecutive position on this list, an index would allow us to easily store or look up the value of a Schur function in memory without any additional arithmetic. We would also like that the parts of the partition be easily recoverable from the index and that a change in one part of the partition correspond straightforwardly to an easy change in the index.

All of those issues were successfully addressed, and we now present our main algorithm.

Algorithm 4.2.1 (Schur function) *The following algorithm computes the Schur function*

$$s_\lambda(x_1, x_2, \ldots, x_n)$$

given $x = (x_1, x_2, \ldots, x_n)$ *and partition* $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_p)$*, where* $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_p > 0$*, using only additions and multiplications. All functions are called by value. The variables* x_i*, the parts of the partition* λ_i*, their number* p *and the* M_i *are* global variables*. All other variables are local.*

10 $\quad M_p = 1$
20 $\quad$ `for` $i = p-1 : -1 : 1$ `do` $M_i = M_{i+1}(\lambda_{i+1}+1)$
30 $\quad N_\lambda =$ `Nmu`(λ)
50 $\quad$ `allocate sh`$(1 : N_\lambda, 1 : n)$.
60 $\quad$ `initialize(1,1,0)`
70 $\quad$ `return sch`$(1, n, 1, N_\lambda)$.

100 `function sch`(j, m, k, N_ν)
110 `if part`$(N_\nu, 1) = 0$ `or` $m < j$ `then return` 1
120 `if part`$(N_\nu, m-j+2) > 0$ `then return` 0
130 `if` $m = 1$ `then return` $x_j^{N_\nu}$
140 `if sh`$(N_\nu, j, m) > -1$ `then return sh`(N_ν, j, m)
150 `if part`$(N_\nu, m) > 0$ `then` $s = (x_j x_{j+1} \ldots x_m)^{\texttt{part}(N_\nu, m)} \cdot$`sch`$(j, m, 1, N_\nu - \sum_{i=j}^m M_i)$
160 `else`
170 $\quad s =$`sch`$(j, m-1, 1, N_\nu)$
180 $\quad$ `if part`$(N_\nu, 1) >$ `part`$(N_\nu, 2)$ `then`
190 $\qquad s = s + x_m \cdot$`sch`$(j, m, 1, N_\nu - M_1)$
200 $\qquad i = 2$
210 $\quad$ `while part`$(N_\nu, i) > 0$
220 $\qquad$ `if part`$(N_\nu, i) >$ `part`$(N_\nu, i+1)$ `then`

230 if part$(N_\nu, i) > 1$ then $s = s + x_m \cdot$ sch$(j, m, i, N_\nu - M_i)$
240 else $s = s + x_m\cdot$ sch$(j, m-1, 1, N_\nu - M_i)$
250 $i = i + 1$
260 if $k = 1$ then sh$(N_\nu, j, m) = s$
270 return s

300 function initialize(t, k, N_ν)
310 if $k \leq p$
320 if $k > 1$ then $m =$min$(\lambda_k,$ part$(N_\nu, k-1))$
330 else $m = \lambda_k$
340 for $i = 1 : m$
350 $N_\nu = N_\nu + M_k$
360 sh$(N_\nu, 1 : t, 1 : n) = -1$
370 initialize$(t, k+1, N_\nu)$

400 function part(N_ν, i) *... computes part i of partition ν given N_ν*
410 if $i = 1$ return $\left\lfloor \frac{N_\nu}{M_1} \right\rfloor$
420 else return $\left\lfloor \frac{\mathrm{mod}(N_\nu, M_{i-1})}{M_i} \right\rfloor$

500 function Nmu(μ)
500 $N = 0$
510 for $i = 1 : p$ do $N = N + \mu_i \cdot M_i$
520 return N

Matlab code for this algorithm is given in Section 4.10.

We now explain what the algorithm does and account for its cost.

In lines 10-20 we generate an array $M_{1:p}$ such that $M_i = \prod_{j=i+1}^{p}(\lambda_j + 1)$ for $i = 1, 2, \ldots, p$ at the cost of $2p$ floating point operations. We define a one-to-one correspondence between any partition $\mu = (\mu_1, \ldots, \mu_p); \mu \leq \lambda$ and a nonnegative integer $N_\mu \leq \prod_{i=1}^{p}(\lambda_i + 1)$ defined as follows $N_\mu = \sum_{i=1}^{p} \mu_i \cdot M_i$. This encoding of the partitions is particularly attractive for the following reasons.

- We can pass partitions to functions as integers instead of arrays

- If μ and ν are two partitions that differ in only one part (say) $\mu_j = \nu_j$ for all $j = 1, 2, \ldots, p$ except $\nu_i = \mu_i - t$ then $N_\nu = N_\mu - t \cdot M_i$. In what follows we will be computing the Schur function recursively by expressing it in terms of Schur functions with partitions that differ from the current partition in at most one part. Thus we can compute the N_μ for the new partition in $O(1)$ time. Additionally this encoding allows for easy indexing of Schur functions that have been computed and have to be stored.

- We can easily recover part i of a partition μ from N_μ in $O(1)$ time:

$$\mu_1 = \left\lfloor \frac{N_\mu}{M_1} \right\rfloor \text{ and } \mu_i = \left\lfloor \frac{\operatorname{mod}(N_\mu, M_{i-1})}{M_i} \right\rfloor \text{ for } i > 1, \tag{4.18}$$

  where $\operatorname{mod}(q, r)$ is the remainder of q modulo r. This is implemented by the function `part` (lines 400-420).

In line 30 we call the function `Nmu`, which computes $N_\lambda = \sum_{i=1}^{p} \lambda_i \cdot M_i$ and costs $2p$.

In line 50 we allocate `sh` which requires $(\lambda_1 + 1) \ldots (\lambda_p + 1)n$ space. For any partition $\mu \leq \lambda$ the value of $s_\mu(x_j, x_{j+1}, \ldots, x_m)$ will be stored in $\mathtt{sh}(N_\mu, j, m)$. Since we will only be computing Schur functions of the form $s_\mu(x_1, x_2, \ldots, x_m)$ *the value of j will equal* 1 *throughout.* The only reason we use a parameter j instead of 1 is that in Section 4.4 where we describe how to solve generalized Vandermonde linear systems, we would need to compute $s_\mu(x_j, x_{j+1}, \ldots, x_m)$ for all $1 \leq j \leq m \leq n$. Line 50 costs $O(1)$.

In line 60 the call to `initialize` sets $\mathtt{sh}(N_\mu, 1 : j, 1 : n) = -1$ for all $\mu \leq \lambda$. All other entries of `sh` will remain undefined and unaccessed. Since the Schur function is always nonnegative, a value of -1 would be a flag that it has not be computed yet. All partitions $\nu \leq \lambda$ are generated recursively in lexicographic order as follows. The first part of ν takes consecutively the values $0, 1, \ldots, \lambda_1$. Given $\nu_1, \ldots, \nu_{k-1}$ the k-th part of ν, $k > 1$ takes consecutively the values $0, 1, \ldots, \min(\lambda_k, \nu_{k-1})$ The value if ν_{k-1} is recovered from N_ν as $\mathtt{part}(k-1, N_n u)$. It costs four floating point operations (and three comparisons and three assignment statements whose cost we ignore) for each $\nu \leq \lambda$. So, the total cost for `initialize` is then $jn \sum_{\nu \leq \lambda} 4$. Again $j = 1$.

The function $\mathtt{sch}(j, m, k, N_\nu)$ computes $s_\nu(x_j, x_{j+1}, .., x_m)$ recursively, k is a parameter keeping track of the recursion and indicating that all subsequent partitions ν' that are generated further into the recursion must satisfy $\nu'_i = \nu_i$ for $i = 1, 2, \ldots, k$. We are ultimately interested in computing $s_\lambda(x_1, x_2, \ldots, x_n)$, which is accomplished by the call to `sch`$(1, n, 1, N_\lambda)$.

In lines 110-140 we handle the cases that terminate the recursion. Line 110 handles the cases when we have a zero partition $\lambda = (0)$ or zero number of variables $m = 0$. Line 120 handles the case of a Schur function with partition that has more non-zero parts than it has variables and is thus equal to zero. Line 130 handles the obvious fact that for Schur function with a partition that only has one part and one variable we have $\nu = (\nu_1)$, $N_\nu = \nu_1$ and $s_\nu(x_j) = x_j^{N_\nu}$. Line 140 checks if the Schur function we need has already been computed and stored and if so returns its value. The cost of lines 110-140 is $O(1)$.

Line 150 accounts for (4.15). The partition that is referred to on this line is

$$(\nu_1 - \nu_m, \nu_2 - \nu_m, \ldots, \nu_m - \nu_m).$$

Then the computation of the Schur functions proceeds according to (4.14).

We now prove the following

Lemma 4.2.1 *The number H_λ of all partitions μ such that $\mu \leq \lambda$, $\mu_1 = \lambda_1$ and $\theta = \lambda/\mu$ is a horizontal strip is*

$$H_\lambda = (\lambda_2 - \lambda_3 + 1)(\lambda_3 - \lambda_4 + 1)\ldots(\lambda_{p-1} - \lambda_p + 1)(\lambda_p + 1).$$

Proof: Let $\theta = (\theta_1, \theta_2, \ldots, \theta_p) = \lambda/\mu = (\lambda_1 - \mu_1, \lambda_2 - \mu_2, \ldots, \lambda_p - \mu_p)$. The condition $\mu_1 = \lambda_1$ implies $\theta_1 = 0$; θ_2 can take any value among $0, 1, \ldots, \lambda_2 - \lambda_3$, i.e. $\lambda_2 - \lambda_3 + 1$ values. Independent of the choice of θ_2, θ_3 can take any value among $0, 1, \ldots, \lambda_3 - \lambda_4$, i.e. $\lambda_3 - \lambda_4 + 1$ values. For θ_p we have $\lambda_p + 1$ possible values $0, 1, \ldots, \lambda_p$. Therefore

$$H_\lambda = (\lambda_2 - \lambda_3 + 1)(\lambda_3 - \lambda_4 + 1)\ldots(\lambda_{p-1} - \lambda_p + 1)(\lambda_p + 1).$$

□

Lines 170-250 implement (4.14) by recursively generating all $\mu \leq \lambda$ such that λ/μ is a horizontal strip and $\lambda_1 = \mu_1$. The recursion generates all μ such that $\mu_1 = \lambda_1$ and $\lambda_{i+1} \leq \mu_i \leq \lambda_i$. The cost of doing so is $2H_\lambda$ plus p for the `while` loop in line 210 which runs from 1 to p.

The partitions that are referred to on lines 190, 230 and 240 of Algorithm 4.2.1 are $(\nu_1 - 1, \nu_2, \ldots, \nu_p)$, $(\nu_1, \ldots, \nu_{i-1}, \nu_i - 1, \nu_{i+1}, \ldots, \nu_p)$ and $(\nu_1, \nu_2, \ldots, \nu_{i-1})$ respectively.

In line 260 the computed value of the Schur function is stored for later use.

Theorem 5 *The cost of computing $s_\lambda(x_1,\ldots,x_n)$ (and $s_\mu(x_1,x_2,\ldots,x_k)$ for all $\mu \le \lambda$ and $k \le n$) using Algorithm 4.2.1 is*

$$F_{\lambda,n} \equiv 4p + \sum_{\mu\le\lambda} 4 + \sum_{i=1}^{n} \sum_{\mu\le\lambda,\mu_{i+1}=0} \left(2\prod_{j=2}^{p}(\mu_i-\mu_{i+1}+1)+p\right). \tag{4.19}$$

The storage space requirement is

$$(\lambda_1+1)(\lambda_2+1)\ldots(\lambda_p+1)n.$$

Proof: Since $s_\mu(x_1,\ldots,x_i)$ for all $\mu \le \lambda$ and $i \le n$ will have to be computed in the course of the recursion, and $s_\mu(x_1,\ldots,x_i)=0$ costs nothing if $\mu_{i+1}>0$, the total cost of computing $s_\lambda(x_1,x_2,\ldots,x_n)$ is

$$4p + \sum_{\mu\le\lambda} 4 + \sum_{i=1}^{n} \sum_{\mu\le\lambda,\mu_{i+1}=0} (2H_\mu + p).$$

□

Example 4.2.3 *We illustrate the algorithm by a small example:*

$s_{(2,1)}(x_1,x_2,x_3) = s_{(1,1)}(x_1,x_2,x_3)x_3 + s_{(2,1)}(x_1,x_2) + x_3 s_{(1,1)}(x_1,x_2) + x_3 s_{(2)}(x_1,x_2)$

$s_{(1,1)}(x_1,x_2,x_3) = s_{(1)}(x_1,x_2,x_3)x_3 + s_{(1,1)}(x_1,x_2)$

$s_{(1)}(x_1,x_2,x_3) = s_{(1)}(x_1,x_2) + s_{(0)}(x_1,x_2)x_3$

$s_{(1)}(x_1,x_2) = s_{(1)}(x_1) + s_{(0)}(x_1)x_2$

$s_{(1)}(x_1) = x_1$

$s_{(0)}(x_1) = 1$

$s_{(0)}(x_1,x_2) = 1$

$s_{(1,1)}(x_1,x_2) = x_1x_2\mathbf{s_{(0)}(x_1,x_2)}$

$s_{(2,1)}(x_1,x_2) = x_1x_2\mathbf{s_{(1)}(x_1,x_2)}$

$\mathbf{s_{(1,1)}(x_1,x_2)}$ *has already been computed*

$s_{(2)}(x_1,x_2) = \mathbf{s_{(1)}(x_1,x_2)}x_2 + s_{(2)}(x_1)$

$s_{(2)}(x_1) = x_1\mathbf{s_{(1)}(x_1)}$,

where functions that have already been computed are shown in **bold**.

Corollary 4.2.1 *If $\lambda = (m, 1^{p-1})$ then $F_{\lambda,n} = O(mp^2n)$.*

Proof: If $\mu \leq \lambda$ then $\mu = (m_1, 1^{k_1})$ where $m_1 \leq m$ and $k_1 \leq p-1$. Also $\mu_{k+1} - \mu_k = 0$ for $k \geq 2$. Thus using (4.19) we have

$$\begin{aligned}
F_{\lambda,n} &= 4p + \sum_{i=0}^{m}\sum_{j=0}^{p} 4 + \sum_{i=0}^{n}\sum_{j=0}^{m}\sum_{r=0}^{\min(i,p-1)} (2+p) \\
&= 4p + 4(m+1)p + m\sum_{i=0}^{p}\sum_{r=0}^{i}(2+p) + m\sum_{i=p+1}^{n}\sum_{r=0}^{p}(2+p) \\
&= 4p + 4(m+1)p + (m+1)(2+p)(p+1)\left(n - \frac{p+1}{2}\right) \\
&= O(mp^2 n).
\end{aligned}$$

□

Note: The complexity of computing the Schur function corresponding to this particular partition could be reduced by a factor of p by making a small change in Algorithm 4.2.1. The `while` loop in lines 210-250 is not needed in order to find the part for which $\nu_{i+1} < \nu_i$. For this particular partition a modification may be used to do this in $O(1)$ time. Since it does not change the fact that the complexity is still a low order polynomial, we will not pursue this issue further.

Corollary 4.2.2 *The cost of computing the Schur function $s_\lambda(x_1, x_2, \ldots, x_n)$, corresponding to a partition $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_p)$ using Algorithm 4.2.1 can be bounded by*

$$F_{\lambda,n} \leq 2(\lambda_1+1)(\lambda_2+1)^2 \ldots (\lambda_p+1)^2(n+5)p \tag{4.20}$$

Proof: We first compute a bound on the number $\sum_{\mu\leq\lambda} 1$ of all partitions less than or equal to λ. We have $\mu_i \leq \lambda_i$. Therefore μ_i can only take one of the values $0, 1, \ldots, \lambda_i$. Possibly not all of them because we must also have $\mu_i \leq \mu_{i-1}$ for $i > 1$. Therefore μ_i can take at most $\lambda_i + 1$ values. Therefore $\sum_{\mu\leq\lambda} 1 \leq (\lambda_1+1)(\lambda_2+1)\ldots(\lambda_p+1)$. Also trivially

$$\prod_{j=2}^{p}(\mu_i - \mu_{i+1} + 1) \leq \prod_{j=2}^{p}(\mu_j + 1) \leq \prod_{j=2}^{p}(\lambda_j + 1),$$

therefore

$$\begin{aligned}
F_{\lambda,n} &\leq 4p + 4\prod_{j=1}^{p}(\lambda_j+1) + \sum_{i=1}^{n}\sum_{\mu\leq\lambda}\left(2\prod_{j=2}^{p}(\lambda_j+1) + p\right) \\
&\leq 4p + 4\prod_{j=1}^{p}(\lambda_j+1) + n\prod_{j=1}^{p}(\lambda_j+1)\left(2\prod_{j=2}^{p}(\lambda_j+1) + p\right) \\
&\leq 2(\lambda_1+1)(\lambda_2+1)^2\ldots(\lambda_p+1)^2(n+5)p.
\end{aligned}$$

□

This justifies the entry for the determinant of a TP generalized Vandermonde matrix in Table 4.1.

In general the bound (4.20) is good for partitions that have only a few parts, and each part is "large". It is a vast overestimate if the partition is "tall" and "thin" (e.g. as in Corollary 4.2.1), i.e. if the partition has many parts each of which is small.

Computing the Schur function efficiently without using subtractions is a challenge because of the number of nonzero terms in s_λ. If one were to compute the Schur function using (4.10) (e.g. [51]) the cost would be at least [47, p.326]

$$\prod_{i,j} \frac{(n-i+j)}{h_{ij}} \approx O(n^{|\lambda|}), \tag{4.21}$$

where h_{ij} is the *hook length* at the (i,j)-th box, i.e. the number of boxes in λ to the right of or below (i,j), including the (i,j)-th box itself, and the product is taken over all boxes in λ, i.e. all $i,j \geq 1$ such that $\lambda_i \geq j$.

In a previous work [25] we presented an algorithm, in which we bounded the cost of computing $s_\lambda(x_1, x_2, \ldots, x_n)$ by

$$O\left(n(1+\lambda_1)^{\log n} \ldots (1+\lambda_p)^{\log n}\right),$$

exponentially faster than (4.21). This latest result bounds the complexity below

$$2(\lambda_1+1)(\lambda_2+1)^2 \ldots (\lambda_p+1)^2 p(n+5),$$

which is yet another exponential improvement over the previous result and allows for a substantial number of Schur functions to be computed in a polynomial amount of time, e.g. if the number of nonzero parts p is fixed and each part $\lambda_1, \lambda_2, \ldots, \lambda_p$ is a polynomial in n.

A final note on the possible loss of accuracy in computing the Schur function. Adding positive quantities preserves the relative accuracy, but very long sums can result in a substantial relative error. The number of terms in s_λ can grow exponentially as seen by (4.21). Even though we compute the Schur function using many fewer arithmetic operations, the relative error can potentially be as large as $O(n^{|\lambda|}\epsilon)$. Getting the answer right will involve arithmetic with precision as wide as $\log\left(n^{|\lambda|}\epsilon\right) = |\lambda|\log n + \log \epsilon$. The cost of running the algorithm in precision that grows this way with the data is only a low order polynomial in the size of the input and independent of the condition number of the initial problem. For the rest of this presentation we will assume that this phenomenon does not occur and that the Schur function is always computed with relative error that is bounded by $h(n)\epsilon$, where h is a polynomial in n.

4.3 Bidiagonal Decomposition of Inverses of TP Matrices

The following result is due to Gasca and Pena [37]. We give a shorter proof here.

Proposition 4.3.1 *If A is TP, then*

$$A^{-1} = U^{(1)}U^{(2)}\ldots U^{(n-1)}D^{-1}L^{(n-1)}L^{(n-2)}\ldots L^{(1)}, \tag{4.22}$$

where $L^{(k)}$ and $U^{(k)}$, $k = 1, 2, \ldots, n$ are unit lower and upper bidiagonal matrices respectively with the property that $L^{(k)}_{i+1,i} = 0$ and $U^{(k)}_{i,i+1} = 0$ for $i \le k-1$, and D^{-1} is a diagonal matrix. Each entry of $U^{(k)}$ and $L^{(k)}$ is a negated product of two quotients of minors of A. Each entry of D is a quotient of minors of A.

Proof: Let

$$A = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ & & \ddots & \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{bmatrix}.$$

For the LDU decomposition of A, where L and U are unit lower and unit upper triangular respectively, it is well known that

$$\begin{aligned} L_{ij} &= \frac{\det(A([1:j-1,i],1:j))}{\det(A(1:j,1:j))}, i > j \\ D_{ii} &= \frac{\det(A(1:i,1:i))}{\det(A(1:i-1,1:i-1))} \\ U_{ij} &= \frac{\det(A(1:i,[1:i-1,j]))}{\det(A(1:i,1:i))}, i < j \end{aligned} \tag{4.23}$$

We consider the process of *Neville elimination*, which is similar to the process of Gaussian Elimination, except that the zeros in the columns are created by subtraction of multiple of row i from row $i+1$ instead of subtracting a multiple of a fixed row from all other rows below it. For instance in order to make zeros in the first column, the process of Neville elimination is equivalent to the multiplication $A^{(1)} = L^{(1)}A$, i.e.

$$A^{(1)} = \begin{bmatrix} 1 & & & & \\ -a_{21}/a_{11} & 1 & & & \\ & -a_{31}/a_{21} & 1 & & \\ & & \ddots & \ddots & \\ & & & -a_{n1}/a_{n-1,1} & 1 \end{bmatrix} \cdot A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ 0 & a_{22}^{(1)} & \dots & a_{2n}^{(1)} \\ 0 & & \ddots & \\ 0 & a_{n2}^{(1)} & \dots & a_{nn}^{(1)} \end{bmatrix},$$

where $a_{ij}^{(1)} = \frac{a_{i-1,1}a_{ij} - a_{i-1,j}a_{i1}}{a_{i-1,1}}$.

Let $A^{(1)}, A^{(2)}, \dots, A^{(n-1)}$ be the sequence of matrices that one would obtain by the means of $n-1$ consecutive steps of Neville elimination and let $L^{(k)}$ be the unit lower bidiagonal matrix for which $A^{(k)} = L^{(k)}A^{(k-1)}$. The matrix $A^{(n-1)}$ is upper triangular and since the LU decomposition is unique, as long as L is unit lower triangular, we have $A^{(n-1)} = DU$. Rows 1 through k do not change after the $k-1$-st step of Neville elimination, therefore

$$A_{kj}^{(k-1)} = (DU)_{kj} = \frac{\det(A(1:k, [1:k-1, j]))}{\det(A(1:k-1, 1:k-1))}.$$

For the $k+1$-st to the last row of $A^{(k-1)}$ we have by analogy

$$A_{ij}^{(k-1)} = \frac{\det(A(i-k+1:i, [1:k-1, j]))}{\det(A(i-k+1:i-1, 1:k-1))}, i \geq k+1, \tag{4.24}$$

since one could replace rows 1 through k with rows $i-k+1$ through i of A and run the same k steps of Neville elimination. In this way the elements $a_{ij}^{(k-1)}, i = 1, 2, \dots, n$ would be on the k-th row, thus confirming (4.24). Therefore

$$\begin{aligned} L_{i+1,i}^{(k)} &= -\frac{A_{i+1,k}^{(k-1)}}{A_{ik}^{(k-1)}} \\ &= -\frac{\det(A(i-k+2:i+1, 1:k))}{\det(A(i-k+2:i, 1:k-1))} \cdot \frac{\det(A(i-k+1:i-1, 1:k-1))}{\det(A(i-k+1:i, 1:k))} \end{aligned} \tag{4.25}$$

for $i \geq k$. In order to obtain the upper unit bidiagonal factors of A^{-1} we apply the same argument to A^T arriving at:

$$\begin{aligned} U_{i,i+1}^{(k)} &= -\frac{A_{k,i+1}^{(k-1)}}{A_{ki}^{(k-1)}} \\ &= -\frac{\det(A(1:k, i-k+2:i+1))}{\det(A(1:k-1, i-k+2:i))} \cdot \frac{\det(A(1:k-1, i-k+1:i-1))}{\det(A(1:k, i-k+1:i))} \end{aligned} \tag{4.26}$$

for $i \geq k$. Thus we obtain (4.22), where D is defined as in (4.23). $\square$

Corollary 4.3.1 *The inverse of a TP matrix A can be expressed as a product of bidiagonal matrices, each of whose entries is a (possibly negated) quotient of minors of A (instead of the product of two quotients).*

Proof: We rewrite A^{-1} as

$$\begin{aligned} A^{-1} &= (U^{(1)}M_1^{-1}) \cdot (M_1 U^{(2)} M_2^{-1}) \cdots (M_{n-2} U^{(n-1)} M_{n-1}^{-1}) \cdot M_{n-1} \cdots \\ &\cdot D^{-1} \cdot N_{n-1} \cdot (N_{n-1}^{-1} L^{(n-1)} N_{n-2}) \cdots (N_2^{-1} L^{(2)} N_1)(N_1^{-1} L^{(1)}), \end{aligned}$$

where we choose the appropriate diagonal matrices M_k and $N_k, k = 1, 2, \ldots, n$ for the job, which differ from the indentity matrices only in the following entries $(M_k)_{ii} = A(1 : k, i-k+1 : i)$ for $i \geq k+1$ and $(N_k)_{ii} = A(i-k+1 : i, 1 : k)$ for $i \geq k+1$.

Let $\mathcal{L}^{(k)} = N_k L^{(k)} N_{k-1}^{-1}$ and $\mathcal{U}^{(k)} = M_k U^{(k)} M_{k-1}^{-1}$ for $2 \leq k \leq n-1$, $\mathcal{L}^{(n)} = N_{n-1}$, $\mathcal{U}^{(n)} = M_{n-1}$, $\mathcal{L}^{(1)} = N_1^{-1} L^{(1)}$ and $\mathcal{U}^{(1)} = U^{(1)} M_1^{-1}$. Then the matrices $\mathcal{L}^{(k)}$ and $\mathcal{U}^{(k)}$ equal the identity matrix except for the following entries:

- For $k = n$ we have

$$\begin{aligned} \mathcal{L}_{nn}^{(n)} &= \det(A(2 : n, 1 : n-1)) \\ \mathcal{U}_{nn}^{(n)} &= \det(A(1 : n-1, 2 : n)) \end{aligned}$$

- For $1 \leq k \leq n$ we have

$$\begin{aligned} \mathcal{L}_{kk}^{(k)} &= \det(A(2 : k, 1 : k-1)) \\ \mathcal{U}_{kk}^{(k)} &= \det(A(1 : k-1, 2 : k)) \end{aligned}$$

- For $1 \leq k \leq n$ and $i \geq k+1$ we have

$$\begin{aligned} \mathcal{L}_{ii}^{(k)} &= \frac{\det(A(i-k+2 : i, 1 : k-1))}{\det(A(i-k+1 : i, 1 : k))} \\ \mathcal{L}_{i,i-1}^{(k)} &= -\frac{\det(A(i-k : i-2, 1 : k-1))}{\det(A(i-k : i-1, 1 : k))} \\ \mathcal{U}_{ii}^{(k)} &= \frac{\det(A(1 : k-1, i-k+2 : i))}{\det(A(1 : k, i-k+1 : i))} \\ \mathcal{U}_{i-1,i}^{(k)} &= -\frac{\det(A(1 : k-1, i-k : i-2))}{\det(A(1 : k, i-k : i-1))}. \end{aligned} \tag{4.27}$$

Finally we obtain

$$A^{-1} = \mathcal{U}^{(1)} \mathcal{U}^{(2)} \ldots \mathcal{U}^{(n)} D^{-1} \mathcal{L}^{(n)} \mathcal{L}^{(n-1)} \ldots \mathcal{L}^{(1)}. \tag{4.28}$$

□

Remarks

1. Since every minor K of A^{-1} is quotient of minors of A (namely the quotient of the complement minor of K in A and $\det(A)$) we can easily obtain a bidiagonal decomposition of A as a product of bidiagonal matrices each entry of which is quotient of minors of A.

2. In the process of Gaussian Elimination we obtain the entries of the triangular factors as quotients of *bordered* leading minors, i.e. leading minors plus one additional row and column. In the process of Neville elimination we obtain the entries of the bidiagonal decomposition of the inverse as products of two quotients of *initial* minors, i.e. contiguous minors which include the first row or the first column.

3. The decomposition (4.22) is not "inferior" to (4.28) as both require about the same number of arithmetic operations to compute. We merely point out that both are possible and draw the attention to the similarity of (4.27) with (4.23).

4. Neville elimination is the "natural" way to perform elimination on TP matrices and has a direct relationship with how the totally positive matrices can be parametrized [37].

5. Strict Total Positivity ensures that we will never hit a zero pivot in the process of Neville Elimination.

6. If A is TP, the component matrices in the decompositions (4.22) and (4.28) have "checkerboard" sign patterns ($\text{sign}(M_{ij}) = (-1)^{i+j}$ or 0), allowing A^{-1} to be computed using either expression without subtractive cancellation obtaining each entry A^{-1} with small relative error. Either decomposition will also allow us to solve $Ax = b$ in $O(n^2)$ additional operations with small backward componentwise relative error and in the case of a right hand side with alternating sign pattern, with small forward componentwise relative error. See Proposition 4.4.1 for details. For TP matrices, being able to compute accurate and efficient initial minors is a sufficient condition for being able to compute an accurate solution to $Ax = b$ or an accurate inverse.

7. For TP Cauchy and TP Vandermonde matrices the decompositions (4.22) and (4.28) yield the well known Björck-Pereyra algorithms [8, 12]. As long as we can compute

quotients of minors accurately and efficiently the algorithms for solving $Ax = b$ resulting from (4.22) and (4.28) are applicable to any TP matrix and the computed solutions will satisfy the same error bounds as the solutions for the Björck-Pereyra-type methods.

8. We will show later how the new methods result in Björck-Pereyra-type algorithms for generalized Vandermonde matrices.

4.4 Björck-Pereyra-Type Methods for TP Generalized Vandermonde Matrices

In this section we will obtain an accurate bidiagonal decomposition of the inverse of a generalized Vandermonde matrix and use it to derive a Björck-Pereyra-type algorithm for solving generalized Vandermonde linear systems.

Let

$$G = \begin{bmatrix} x_1^{\lambda_n} & x_1^{1+\lambda_{n-1}} & \dots & x_1^{n-1+\lambda_1} \\ x_2^{\lambda_n} & x_2^{1+\lambda_{n-1}} & \dots & x_2^{n-1+\lambda_1} \\ & & \ddots & \\ x_n^{\lambda_n} & x_n^{1+\lambda_{n-1}} & \dots & x_n^{n-1+\lambda_1} \end{bmatrix}$$

be a TP generalized Vandermonde matrix, corresponding to a partition $\lambda = (\lambda_1, \lambda_2, \dots, \lambda_n)$, where $0 < x_1 < x_2 < \dots < x_n$. Let

$$V = \left[x_i^{j-1}\right]_{i,j=1}^{n}$$

be the ordinary Vandermonde matrix. For any $i \leq j, k \leq l \in \{1, 2, \dots, n\}$ such that $j - i = l - k$ we have from (4.9) that

$$\begin{aligned} \det(G_{i:j,k:l}) &= \det\left(\begin{bmatrix} x_i^{k-1+\lambda_{n-k+1}} & x_i^{k+\lambda_{n-k}} & \dots & x_i^{l-1+\lambda_{n-l+1}} \\ x_{i+1}^{k-1+\lambda_{n-k+1}} & x_{i+1}^{k+\lambda_{n-k}} & \dots & x_{i+1}^{l-1+\lambda_{n-l+1}} \\ & & \ddots & \\ x_j^{k-1+\lambda_{n-k+1}} & x_j^{k+\lambda_{n-k}} & \dots & x_j^{l-1+\lambda_{n-l+1}} \end{bmatrix}\right) \\ &= \det(V_{i:j,1:l-k+1}) \cdot s_{(\lambda_{n-l+1},\lambda_{n-l+2},\dots,\lambda_{n-k+1})}(x_i, x_{i+1}, \dots, x_j) \cdot \prod_{s=i}^{j} x_s^{k-1}. \end{aligned}$$

Now for the bidiagonal decomposition (4.22) of G^{-1} we have from (4.23), (4.25) and (4.26)

$$
\begin{aligned}
L_{i+1,i}^{(k)} &= -\frac{\det(G_{i-k+2:i+1,1:k})}{\det(G_{i-k+2:i,1:k-1})}\cdot\frac{\det(G_{i-k+1:i-1,1:k-1})}{\det(G_{i-k+1:i,1:k})}\\
&= -\frac{\det(V_{i-k+2:i+1,1:k})\cdot s_{(\lambda_{n-k+1:n})}(x_{i-k+2:i+1})}{\det(V_{i-k+2:i,1:k-1})\cdot s_{(\lambda_{n-k+2:n})}(x_{i-k+2:i})}\\
&\quad\cdot\frac{\det(V_{i-k+1:i-1,1:k-1})\cdot s_{(\lambda_{n-k+2:n})}(x_{i-k+1:i-1})}{\det(V_{i-k+1:i,1:k})\cdot s_{(\lambda_{n-k+1:n})}(x_{i-k+1:i})}\\
&= -\frac{s_{(\lambda_{n-k+1:n})}(x_{i-k+2:i+1})}{s_{(\lambda_{n-k+2:n})}(x_{i-k+2:i})}\cdot\frac{s_{(\lambda_{n-k+2:n})}(x_{i-k+1:i-1})}{s_{(\lambda_{n-k+1:n})}(x_{i-k+1:i})}\\
&\quad\cdot\prod_{j=i-k+1}^{i-1}\frac{x_{i+1}-x_{j+1}}{x_i-x_j}. &(4.29)\\
D_{ii}^{-1} &= \frac{\det(G_{1:i-1,1:i-1})}{\det(G_{1:i,1:i})} = \frac{s_{(\lambda_{n-i+2:n})}(x_{1:i-1})}{s_{(\lambda_{n-i+1:n})}(x_{1:i})}\cdot\prod_{j=1}^{i-1}\frac{1}{x_i-x_j}. &(4.30)\\
U_{i,i+1}^{(k)} &= -\frac{\det(G_{1:k,i-k+2:i+1})}{\det(G_{1:k-1,i-k+2:i})}\cdot\frac{\det(G_{1:k-1,i-k+1:i-1})}{\det(G_{1:k,i-k+1:i})}\\
&= -\frac{\det(V_{1:k,1:k})\cdot s_{(\lambda_{n-i:n-i+k-1})}(x_{1:k})\prod_{s=1}^{k}x_s^{i-k+1}}{\det(V_{1:k-1,1:k-1})\cdot s_{(\lambda_{n-i+1:n-i+k-1})}(x_{1:k-1})\prod_{s=1}^{k-1}x_s^{i-k+1}}\\
&\quad\cdot\frac{\det(V_{1:k-1,1:k-1})\cdot s_{(\lambda_{n-i+2:n-i+k})}(x_{1:k-1})\prod_{s=1}^{k-1}x_s^{i-k}}{\det(V_{1:k,1:k})\cdot s_{(\lambda_{n-i+1:n-i+k})}(x_{1:k})\prod_{s=1}^{k}x_s^{i-k}}\\
&= -x_k\cdot\frac{s_{(\lambda_{n-i:n-i+k-1})}(x_{1:k})}{s_{(\lambda_{n-i+1:n-i+k-1})}(x_{1:k-1})}\,\frac{s_{(\lambda_{n-i+2:n-i+k})}(x_{1:k-1})}{s_{(\lambda_{n-i+1:n-i+k})}(x_{1:k})}. &(4.31)
\end{aligned}
$$

Example 4.4.1 *Here is an example of the bidiagonal decomposition of the inverse of a particular Generalized Vandermonde matrix corresponding to a partition in only one part* $\lambda = (1)$, *for which we have* $s_{(1)}(x_1, x_2, \ldots, x_i) = x_1 + x_2 + \ldots + x_i$

$$
\left(\begin{bmatrix} 1 & x_1 & x_1^2 & x_1^4\\ 1 & x_2 & x_2^2 & x_2^4\\ 1 & x_3 & x_3^2 & x_3^4\\ 1 & x_4 & x_4^2 & x_4^4 \end{bmatrix}\right)^{-1} =
$$

$$
= \begin{bmatrix} 1 & -x_1 & & \\ & 1 & -x_1 & \\ & & 1 & -x_1\\ & & & 1 \end{bmatrix}\cdot\begin{bmatrix} 1 & & & \\ & 1 & -x_2 & \\ & & 1 & -x_2\\ & & & 1 \end{bmatrix}\cdot\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & -x_3(x_1+x_2+x_3)\\ & & & 1 \end{bmatrix}
$$

$$\cdot \mathrm{diag}\ (1, x_2 - x_1, (x_3 - x_2)(x_3 - x_1), (x_4 - x_3)(x_4 - x_2)(x_4 - x_1)(x_1 + x_2 + x_3 + x_4))^{-1}$$

$$\cdot \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & -\frac{(x_4-x_3)(x_4-x_2)}{(x_3-x_1)(x_3-x_2)} & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & & & \\ & 1 & & \\ & -\frac{x_3-x_2}{x_2-x_1} & 1 & \\ & & -\frac{x_4-x_3}{x_3-x_2} & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & & & \\ -1 & 1 & & \\ & -1 & 1 & \\ & & -1 & 1 \end{bmatrix}.$$

Given the bidiagonal decomposition of G^{-1} we can solve $Gy = b$ in $O(n^2)$ operations using the following

Algorithm 4.4.1 (Solving Generalized Vandermonde Systems) *If G is a generalized Vandermonde matrix determined by a vector $x = (x_1, x_2, \ldots, x_n)$ and a partition $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_n)$, the following algorithm computes the solution to $Gy = b$*

`function gvand`$([\lambda_1, \ldots, \lambda_n], [x_1, x_2, \ldots, x_n], [b_1, b_2, \ldots, b_n])$
$M_p = 1$; `for` $i = p - 1 : -1 : 1$ `do` $M_i = M_{i+1}(\lambda_{i+1} + 1)$
$N_\lambda = \mathtt{Nmu}(\lambda)$
`allocate sh`$(1 : N_\lambda, 1 : n, 1 : n)$.
`initialize(n,1,0)`

$d =$ `ones`(n)
`for` $i = 2 : n$
    `for` $j = i - 1 : -1 : 1$
        $d_{ij} = d_{i,j+1} \cdot (x_i - x_j)$

`NmuL=zeros`(n)
`for` $i = 1 : n$
    $\mathtt{NmuL}(i, i) = \lambda_i$
    `for` $j = i + 1 : n$
        $\mathtt{NmuL}(i, j) = \mathtt{NmuL}(i, j - 1) \cdot M_{j-i} + \lambda_j$

`for` $k = 1 : n - 1$
    `for` $i = n - 1 : -1 : k$
        $L^{(k)}_{i+1,i} = \frac{\mathtt{sch}(i-k+2,i+1,1,\mathtt{NmuL}(n-k+1,n))}{\mathtt{sch}(i-k+1,i,1,\mathtt{NmuL}(n-k+1,n))} \cdot \frac{\mathtt{sch}(i-k+1,i-1,1,\mathtt{NmuL}(n-k+2,n))}{\mathtt{sch}(i-k+2,i,1,\mathtt{NmuL}(n-k+2,n))} \cdot \frac{d_{i+1,i-k+2}}{d_{i,i-k+1}}$
        $b_{i+1} = b_{i+1} - b_i \cdot L^{(k)}_{i+1,i}$

`for` $i = 1 : n$
    $D^{-1}_{ii} = \frac{\mathtt{sch}(1,i-1,1,\mathtt{NmuL}(n-i+2,n))}{\mathtt{sch}(1,i,1,\mathtt{NmuL}(n-i+1,n))} \cdot \frac{1}{d_{i,1}}$

210 $\quad\quad b_i = b_i \cdot D_{ii}^{-1}$

220 `for` $k = n-1 : -1 : 1$

230 $\quad\quad$ `for` $i = k : n-1$

240 $\quad\quad\quad U_{i,i+1}^{(k)} = x_k \cdot \frac{\texttt{sch}(1,k-1,1,\texttt{NmuL}(n-i+2,n-i+k))}{\texttt{sch}(1,k-1,1,\texttt{NmuL}(n-i+1,n-i+k-1))} \cdot \frac{\texttt{sch}(1,k,1,\texttt{NmuL}(n-i,n-i+k-1))}{\texttt{sch}(1,k,1,\texttt{NmuL}(n-i+1,n-i+k))}$

250 $\quad\quad\quad b_i = b_i - b_{i+1} \cdot U_{i,i+1}^{(k)}$

260 `return` b

A Matlab implementation of Algorithm 4.4.1 is given in Section 4.10. We now give a line-by-line explanation and account for its cost.

The solver is called by supplying the vectors x, λ and b. They are all assumed to have the same length n. If λ is shorter we can always pad it with zeros at the end (which we do in the actual implementation). Lines 20-40 serve a very similar purpose to their counterparts in Algorithm 4.2.1. The only difference is that the array `sh` is now three-dimensional and the value of $s_\mu(x_j, \ldots, x_m)$ is to be stored in $\texttt{sh}(N_\mu, j, m)$. The cost so far is $4n$ and the storage is $(\lambda_1+1)(\lambda_2+1)\ldots(\lambda_n+1)n^2$.

Lines 50-80 generate an n-by-n array d such that

$$d_{ij} = (x_i - x_{i-1})(x_i - x_{i-2})\ldots(x_i - x_j).$$

They cost n^2.

In lines 90-130 we generate an array `NmuL` in which the (i,j)-th entry will contain the value of

$$\texttt{Nmu}(\lambda_i, \lambda_{i+1}, \ldots, \lambda_j),$$

i.e. the unique integer that represents the partition $(\lambda_i, \lambda_{i+1}, \ldots, \lambda_j)$. This integer would have to be passed to the function `sch` when the Schur function corresponding to this partition needs to be computed. Precomputing those integers allows us to form the parameters for the call to the function $\texttt{sch}(k,l,1,\texttt{NmuL}(i,j))$ at $O(1)$ cost instead of the equivalent $\texttt{sch}(k,l,1,\texttt{Nmu}([\lambda_i, \lambda_{i+1}, \ldots, \lambda_j]))$, which costs $O(n)$. Since `sch` would need to be called $O(n^2)$ times this trick saves us a factor of n. The cost of lines 90-130 is $n^2 + O(n)$.

If $G^{-1} = U^{(1)}U^{(2)}\ldots U^{(n-1)}D^{-1}L^{(n-1)}L^{(n-2)}\ldots L^{(1)}$ is the decomposition (4.22) for G^{-1} then the system $Gy = b$ is solved as $y = G^{-1}b$ in lines 150-250 by using this bidiagonal decomposition and multiplying right-to-left. The elements $L_{i+1,i}^{(k)}, U_{i,i+1}^{(k)}$ and D_{ii} are computed using the formulas (4.29),(4.30) and (4.31).

Apart from the cost of computing the Schur functions, the cost of lines 90 – 190 is $8n^2 + O(n)$.

All Schur functions computed on lines 90 – 190 are of the form $s_\mu(x_j, x_{j+1}, \ldots, x_l)$, where $\mu \leq \lambda$. Let $F_{\lambda,n}$ be defined as in Theorem 5 in Section 4.2. Since computing $s_\lambda(x_i, x_{i+1}, \ldots, x_j)$ results in also computing $s_\mu(x_i, x_{i+1}, \ldots, x_k)$ for all $i \leq k \leq j$ and all $\nu \leq \lambda$, the cost of computing all Schur functions involved in the expressions (4.29),(4.30) and (4.31) is

$$\sum_{i=1}^{n} F_{\lambda,n-i+1}.$$

Thus we obtain the following

Corollary 4.4.1 *The cost of Algorithm 4.4.1 is* $(\lambda_1+1)(\lambda_2+1)\ldots(\lambda_n+1)n^2$ *for storage and*

$$10n^2 + \sum_{i=1}^{n} F_{\lambda,n-i+1} + O(n) \tag{4.32}$$

for arithmetic. Furthermore (4.32) may be bounded by

$$\left(2(\lambda_1+1)(\lambda_2+1)^2\ldots(\lambda_p+1)^2 p + 10\right)(n+5)n + O(n).$$

This justifies the entries for the inverse, NENP, and solving linear systems with small forward and backward error for TP generalized Vandermonde matrices in Table 4.1.

We now turn our attention to the accuracy of the computed solution with the following proposition

Proposition 4.4.1 *[12, 43] If the relative error in every entry of* $L^{(i)}, U^{(i)}$ D^{-1} *is small (i.e. does not exceed a value* $\eta = h(n)\epsilon$ *where* $h(n)$ *is a polynomial in* n*),* $\hat{y}$ *is the computed solution of* $Gy = b$ *and* $\hat{G}$ *is such that* $\hat{G}\hat{y} = b$*, then the following error bounds are valid:*

1. $$|y - \hat{y}| \leq O(nh(n))\epsilon|G^{-1}||b|$$

2. *If* b *has alternating sign pattern (i.e.* $(-1)^i f_i$ *have the same sign for all* $i = 1, 2, \ldots, n$*) then solution is computed with a small componentwise relative forward error*

$$|y - \hat{y}| \leq O(nh(n))\epsilon|y|;$$

3. *The componentwise relative backward error is small*

$$|G - \hat{G}| \leq O(n^2 h(n))\epsilon|G| + O(\epsilon^2)$$

4. *The componentwise residual error is also small:*

$$|b - G\hat{y}| \le O(n^2 h(n))\epsilon |G||\hat{y}| + O(\epsilon^2)$$

A proof is available in [12, 43], we will just sketch it here. The bidiagonal factors have positive diagonals and non-positive off diagonals. This property is inherited from the total positivity. Thus the product of all bidiagonals may be formed obtaining G^{-1} without suffering any subtractive cancellation. If the right hand side has alternating sign pattern then the solution may also be formed without subtractive cancellation, proving 1. and 2. It is very important that the solution be computed right-to-left and not as $G^{-1} \cdot b$, because the latter is not backward stable. The parts 3. and 4. are equivalent and follow from the fact that the multiplication by the bidiagonal factors of G^{-1} is backwards stable and the fact that the inverses of those bidiagonal factors are totally positive.

4.5 Computing Accurate Inverses Using Cramer's Rule

Although an accurate inverse can be readily computed as a consequence of the Björck-Pereyra-type methods applied to either TP Cauchy or TP (generalized) Vandermonde matrices, Cramer's rule offers yet another way of computing accurate inverses through the computation of accurate minors. In this section we briefly review this method for Cauchy and TP (generalized) Vandermonde matrices.

For a Cauchy matrix $C = \left[\frac{1}{x_i + y_j}\right]_{i,j=1}^{n}$, where $x_{1:n}$ and $y_{1:n}$ are vectors such that $x_i + y_j \neq 0$ for all i, j, there is a well-known [12] formula

$$\det(C) = \frac{\prod_{i<j}(x_j - x_i)(y_j - y_i)}{\prod_{i,j}(x_i + y_j)}, \tag{4.33}$$

which allows the accurate computation of its determinant (or any minor, which is also Cauchy). The matrix C is invertible if $x_i \neq x_j$ and $y_i \neq y_j$ for all $i \neq j$ and (4.33) combined with the Cramer's rule yields

$$(C^{-1})_{ij} = \frac{\prod_{k=1}^{n}(x_j + y_k)(x_k + y_i)}{(x_j + y_i)\prod_{k=1,k\neq j}^{n}(x_j - x_k)\prod_{k=1,k\neq i}^{n}(y_i - y_k)}.$$

Since there is no subtractive cancellation in the above formula, the relative accuracy will be preserved and an accurate inverse can be computed in $O(n^2)$ time (one would need to precompute all expressions $\prod_{k=1}^{n}(x_j + y_k)$, $\prod_{k=1}^{n}(x_k + y_i)$, $\prod_{k=1,k\neq j}^{n}(x_j - x_k)$, and

$\prod_{k=1,k\neq i}^{n}(y_i - y_k)$ to achieve that cost). Total positivity is not needed in order to compute an accurate C^{-1}.

Now we turn our attention to the TP generalized Vandermonde matrix

$$G = \left[x_i^{j-1+\lambda_{n-j+1}}\right]_{i,j=1}^{n},$$

corresponding to a partition $\lambda = (\lambda_1, \lambda_2, \ldots \lambda_n)$, where $x_n > \ldots > x_1 > 0$. Any $(n-1) \times (n-1)$ minor of G, obtained by removing row k and column l equals

$$G^{(k,l)} = \left[x_i^{j-1+\lambda_{n-j+1}}\right]_{i,j=1;i\neq k,j\neq l}^{n}$$

and corresponds to partition $\lambda^{(k,l)} = (\lambda_1 + 1, \lambda_2 + 1, \ldots, \lambda_{n-l} + 1, \lambda_{n-l+2}, \ldots, \lambda_n)$. This minor is also a generalized Vandermonde determinant, therefore

$$\det(G^{(k,l)}) = s_{\lambda^{(k,l)}}(x_1, \ldots, x_{k-1}, x_{k+1}, \ldots, x_n) \cdot \det(V(x_1, \ldots, x_{k-1}, x_{k+1}, \ldots, x_n)),$$

where V is the ordinary Vandermonde matrix. We have

$$\begin{aligned}
(G^{-1})_{lk} &= (-1)^{k+l}\frac{\det(G^{(k,l)})}{\det(G)} \\
&= (-1)^{k+l}\frac{s_{\lambda^{(k,l)}}(x_1, \ldots, x_{k-1}, x_{k+1}, \ldots, x_n)\det(V(x_1, \ldots, x_{k-1}, x_{k+1}, \ldots, x_n))}{\det(V(x_1, \ldots, x_n))s_\lambda(x_1, \ldots, x_n)} \\
&= (-1)^{n-l}\frac{s_{\lambda^{(k,l)}}(x_1, \ldots, x_{k-1}, x_{k+1}, \ldots, x_n)}{s_\lambda(x_1, \ldots, x_n)} \cdot \prod_{i=1,i\neq k}^{n} \frac{1}{x_k - x_i}
\end{aligned} \tag{4.34}$$

for all k, l.

Computing explicit G^{-1} using Cramer's rule is in general more expensive than using the bidiagonal decomposition of G^{-1} to do the same, except in some cases, e.g. the ordinary Vandermonde matrix as we describe in Section 4.6.

4.6 Inverting Vandermonde Matrices

Following Section 4.5 we consider a TP Vandermonde matrix $V = \left[x_i^{j-1}\right]_{i,j=1}^{n}$, where $0 < x_1 < x_2 < \cdots < x_n$. Our goal is to compute V^{-1} to high relative accuracy componentwise.

Using the Björck-Pereyra method [8] to achieve this goal by simply forming the product of the bidiagonal factors of V^{-1} that it produces costs $2n^3$. The total positivity of V ensures no subtractive cancellation can occur in forming the product.

Here we present an alternative method, relies on the accurate computation of the minors through accurate computation of Schur functions and Cramer's rule to compute V^{-1} in only $\frac{5}{12}n^3$ time, which is an improvement over the previous best result by a factor of 4.8.

Following Section 4.5 we need to compute the quantities in (4.34). For V we have $\lambda = (0)$ therefore $\lambda^{(k,l)} = (1^{n-l})$ and $s_\lambda(x_1, x_2, \ldots, x_n) = 1$. The factor $\prod_{i=1, i\neq k}^{n}(x_k - x_i)^{-1}$ need only be computed once for each k. The formula that we will use for computing $s_{(1^m)}(x_1, \ldots, x_i)$ follows from (4.14):

$$s_{(1^m)}(x_1, x_2, \ldots, x_i) = s_{(1^m)}(x_1, x_2, \ldots, x_{i-1}) + s_{(1^{m-1})}(x_1, x_2, \ldots, x_{i-1}) \cdot x_i. \tag{4.35}$$

The process of computing $s_{(1^m)}(x_1, \ldots, x_{k-1}, x_k)$ is such that $s_{1^r}(x_1, \ldots, x_i)$ will be computed in the process for all $r \leq m$ and $i \leq k$. We can now use (4.35) to compute $s_{(1^m)}(x_1, \ldots, x_{k-1}, x_j)$ from $s_{(1^m)}(x_1, x_2, \ldots, x_{i-1})$ and $s_{(1^{m-1})}(x_1, x_2, \ldots, x_{i-1})$ which have been computed in the previous computation.

Algorithm 4.6.1 *The given a vector* $x = (x_1, x_2, \ldots, x_n)$, *where* $0 < x_1 < \ldots < x_n$ *the following algorithm computes* V^{-1} *to high relative accuracy componentwise:*

```
function B = invvand(x)
global s, n
n = length(x)
nn = ⌊n/2⌋
s(n, n) = 0        % allocate space for an n-by-n array
s(:, 1) = ones(n, 1)
d = schurtable(x(n), x(1 : n − 1), 2)
B(:, n) = s(n − 1, n : −1 : 1)′/d
for k = 2 : nn
    d = schurtable(x(n − k + 1), [x(1 : n − k), x(n − k + 2 : n)], n − k + 1)
    B(:, n − k + 1) = s(n − 1, n : −1 : 1)′/d
d = schurtable(x(1), x(n : −1 : 2), 2)
B(:, 1) = s(n − 1, n : −1 : 1)′/d
for k = n − 1 : −1 : nn + 1
    d = schurtable(x(n − k + 1), [x(n : −1 : n − k + 2), x(n − k : −1 : 1)], n − k + 1)
    B(:, n − k + 1) = s(n − 1, n : −1 : 1)′/d
```

```
170 for k = 1 + mod(n, 2) : 2 : n
180     B(k, :) = −B(k, :)

300 function d=schurtable(x, z, m)
310 global s, n
320 s(1, 2) = z(1)
330 for i = m : n − 1
340     for j = 2 : i + 1
350         s(i, j) = s(i − 1, j) + s(i − 1, j − 1) ∗ z(i)
360 d = 1
370 for l = 1 : n − 1
380     d = d ∗ (x − z(l))
```

A Matlab implementation of this algorithm is given in Section 4.10.

We now give a line-by-line description and account for the cost. Lines 10-60 are self explanatory. They cost $n+3$. The function `schurtable` in lines 300-390 takes as a parameter a vector z that will usually be $(x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$ a value x that will usually stand for x_i (the missing variable from z) and a parameter m that would indicate that the table s has already been built up to level $m-1$.

The function modifies the array s so on exit we have $s(i,j) = s_{(1^{i-1})}(z_1, z_2, \ldots, z_j)$ for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n-1$. It also reports one additional variable

$$dd = \prod_{i=1}^{n-1} (x - z(i)).$$

This is the inverse of the last factor in (4.34). A call to `schurtable` costs $n(n+1) - m(m-1)$. Line 70 computes $s_{(1^{i-1})}(x_1, x_2, \ldots, x_j)$ for $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, n-1$. Now $s(n-1, :)/dd$ is the last column of B in reverse order. Line 80 reverses the order accordingly. For the second-to-last column of B we need $s_{(1^{i-1})}(x_1, x_2, \ldots, x_{n-k-1}, x_{n-k+1}, \ldots, x_n)$ for all $i = 1, 2, \ldots, n$, where $k = 1$. We can compute those quantities using the function `schurtable`. Since we already have $s_{(1^{i-1})}(x_1, x_2, \ldots, x_{n-2})$ from the previous computation, we only have to recompute the last row of s. Thus `schurtable` is called to start at the $n-k = n-1$-st row of s. We continue in a similar manner for $k = 2, 3, \ldots, \left\lfloor \frac{n}{2} \right\rfloor$. By line 120 we have computed half the entries in B and in lines 120-160 we do the same for the other half. In line 170-180 we adjust the signs to ensure sign$B(i,j) = (-1)^{i+j}$ and return B.

Lines 70-80 cost $O(n^2)$ and lines 90-110 cost

$$O(n^2) + \sum_{i=\lfloor \frac{n}{2} \rfloor}^{n} (n^2 - i^2) = O(n^2) + \frac{n^3}{2} - \frac{n^3}{3} + \left(\frac{n}{2}\right)^3 \frac{1}{3} = \frac{5}{24} n^3 + O(n^2).$$

Lines 120-130 cost the same as lines 70-80, i.e. $O(n^2)$ and lines 140-160 cost the same as lines 90-110. Lines 170-180 cost $O(n^2)$. Thus we obtain the following

Proposition 4.6.1 *The cost of Algorithm 4.6.1 is*

$$\frac{5}{12} n^3 + O(n^2)$$

for arithmetic and n^2 for storage.

And because all computations preserve the relative accuracy, each entry of V^{-1} is computed to high relative accuracy.

4.7 Accurate LDU Decomposition of TP Matrices. Application to TP (generalized) Vandermonde Matrices

Computing an LDU decomposition with small relative error componentwise is an important task for TP matrices, because it allows us to solve linear systems with small componentwise relative backward error by doing a forward and a backward substitution [17]. Another reason why this is an important task is the fact that if an LDU decomposition from Gaussian Elimination with complete pivoting is available to us and is computed with small relative error, then we can use the algorithm of Demmel [22] to compute its SVD to high relative accuracy.

Having said that, the Björck-Pereyra-type methods are always better for solving a TP linear system compared to using an LDU decomposition, because they guarantee both small componentwise relative forward *and* backward error. Also, we do not know how to compute LDU with complete pivoting with small relative error componentwise for any class of Vandermonde matrices we are considering. Despite those facts we look at the problem anyway.

If no pivoting is desired, then an accurate LDU of a TP matrix may be obtained at $O(n^3)$ cost if an accurate bidiagonal decomposition (4.22) of the inverse is available. The matrix L may be obtained by solving $L^{(n-1)}L^{(n-2)}\ldots L^{(1)}x = b$ for every column of the identity matrix. Solving $L^{(i)}y = z$ for each $i = 1, 2, \ldots, n-1$ costs $O(n)$ flops. If z has only nonnegative terms, then y will also be nonnegative componentwise and be computed to with small relative error since there is no subtractive cancellation. The matrix U can be computed in a similar way.

He we present an alternative method for computing an LDU decomposition using Schur functions, whose cost only depends quadratically on n. With this approach partial pivoting may be used as well. Let G be a TP (generalized) Vandermonde matrix $G = \left[x_i^{j-1+\lambda_{n-j+1}}\right]_{i,j=1}^{n}$, where $\lambda_1 \geq \ldots \geq \lambda_p > 0 = \lambda_{p+1} = \ldots = \lambda_n$ are integers and $x_n > \ldots > x_1 > 0$. We can afford partial pivoting in G since it only permutes the x_i's and we only need the x_i's to be positive in order to compute accurate minors, not necessarily sorted in increasing order. Complete pivoting may also be incorporated using this approach, but the cost may be exponential, since we may have to compute minors corresponding to partitions that are $O(n^2)$ larger than $|\lambda|$.

If $G = LDU$, where L and U are unit lower and unit upper triangular respectively then

$$D_{ii} = \frac{\det(G(1:i,1:i))}{\det(G(1:i-1,1:i-1))},$$

$$U_{ij} = \frac{\det(G(1:i,[1:i-1,j]))}{\det(G(1:i,1:i))}, i < j, \quad \text{and} \quad L_{ij} = \frac{\det(G([1:j-1,i],1:j))}{\det(G(1:j,1:j))}, i > j.$$

All minors in the above expressions are generalized Vandermonde determinants and can thus be computed to high relative accuracy using the algorithm from Section 4.2. A variation of this approach is used in [71].

We will now obtain a bound on the cost of computing the LDU decomposition of G. We have

$$\det(G(1:i,1:i)) = \det(V(1:i,1:i)) \cdot s_{(\lambda_{n-i+1},\ldots,\lambda_p)}(x_{1:i})$$

$$\det(G([1:i-1,j],1:i)) = \det(V([1:i-1,j],1:i)) \cdot s_{(\lambda_{n-i+1},\ldots,\lambda_p)}(x_{[1:i-1,j]})$$

$$\det(G(1:i,[1:i-1,j])) = \det(V(1:i,1:i)) \cdot s_{(\lambda_{n-j+1}+j-i,\lambda_{n-i+2},\ldots,\lambda_p)}(x_{1:i})$$

where V is the ordinary Vandermonde matrix. Therefore

$$D_{ii} = \frac{\det(V(1:i,1:i))s_{(\lambda_{n-i+1},\ldots,\lambda_p)}(x_{1:i})}{\det(V(1:i-1,1:i-1))s_{(\lambda_{n-i+2},\ldots,\lambda_p)}(x_{1:i-1})} = \frac{s_{(\lambda_{n-i+1},\ldots,\lambda_p)}(x_{1:i})}{s_{(\lambda_{n-i+2},\ldots,\lambda_p)}(x_{1:i-1})}\prod_{j=1}^{i-1}(x_i - x_j)$$

for $i = 1, 2, \ldots, n$,

$$U_{ij} = \frac{\det(V(1:i,1:i)) \cdot s_{(\lambda_{n-j+1}+j-i,\lambda_{n-i+2},\ldots,\lambda_p)}(x_{1:i})}{\det(V(1:i,1:i)) \cdot s_{(\lambda_{n-i+1},\ldots,\lambda_p)}(x_{1:i})} = \frac{s_{(\lambda_{n-j+1}+j-i,\lambda_{n-i+2},\ldots,\lambda_p)}(x_{1:i})}{s_{(\lambda_{n-i+1},\ldots,\lambda_p)}(x_{1:i})}$$

for $1 \leq i < j \leq n$ and

$$L_{ij} = \frac{V([1:j-1,i],1:i) \cdot s_{(\lambda_{n-j+1},\ldots,\lambda_p)}(x_{1:j-1,i})}{\det(V(1:j,1:j)) \cdot s_{(\lambda_{n-j+1},\ldots,\lambda_p)}(x_{1:j})} = \frac{s_{(\lambda_{n-j+1},\ldots,\lambda_p)}(x_{1:j-1,i})}{s_{(\lambda_{n-j+1},\ldots,\lambda_p)}(x_{1:j})} \prod_{k=1}^{j-1} \frac{x_i - x_k}{x_j - x_k}$$

for all $1 \leq j < i \leq n$. Computing all factors of the form $\prod_{j=1}^{i-1}(x_i - x_j)$ and $\prod_{k=1}^{j-1} \frac{x_i - x_k}{x_j - x_k}$ in D_{ii} and L_{ij} costs $O(n^2)$. Additionally we have to compute

- $s_\mu(x_{1:i})$ for all $i \leq n$ and $\mu \leq \lambda$, which costs $F_{\lambda,n}$, where $F_{\lambda,n}$ is defined as in Theorem 5 in Section 4.2;
- $s_\mu(x_{1:i-1,j})$ for all $n \geq j \geq i \geq 1$ and $\mu \leq \lambda$, which costs an additional $O(F_{\lambda,n} \cdot n)$;
- $s_\mu(x_{1:i})$ for all $i \leq n$ and $\mu_1 \leq \lambda_1 + n - 1, \mu_i \leq \lambda_i, i = 2, \ldots, p$, which costs not more than $F_{(\lambda_1+n-1,\lambda_2,\ldots,\lambda_p),n} \leq F_{\lambda,n} \cdot n$.

Therefore the cost of getting an accurate LDU decomposition of G can be bounded by

$$O\left((\lambda_1+1)(\lambda_2+1)^2 \ldots (\lambda_p+1)pn^2\right),$$

which explains the entry in Table 4.1.

Our interest in the LDU decomposition that results from GECP is justified by the fact that we can compute the SVD to high relative accuracy of any matrix for which we have an accurate LDU decomposition from GECP. For a TP generalized Vandermonde matrix corresponding to a partition λ, complete pivoting may result in the need for an accurate computation of minors corresponding to partitions that are $O(n)$ larger than λ. While computing those minors accurately is now possible faster than before, it may still be exponential. Thus the entries for "any minor", "GECP" and "SVD" on the row for TP generalized Vandermonde matrices in Table 4.1. Also the entry for "any minor" in the row for a TP Vandermonde matrix.

4.8 Numerical Experiments

We performed extensive numerical experiments and confirmed the correctness and cost of our algorithms. We selected two of those tests and report them below.

We tested Algorithm 4.4.1 as follows.

We chose a partition $\lambda = (6, 3, 2, 1, 1)$, $n = 12$ random nodes $x_i \in [2, 3]$, which we then sorted in increasing order, and a right hand side $b = (-1, 1, -1, 1, -1, 1, -1, 1, -1, 1, -1, 1)^T$ with alternating signs. This is equivalent to interpolating the data b with a polynomial of the form

$$f(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5 + a_6 x^6 + a_7 x^8 + a_8 x^9 + a_9 x^{11} + a_{10} x^{13} + a_{11} x^{17}$$

at the nodes x_i, $i = 1, 2, \ldots, 12$. This resulted in a TP matrix $G_\lambda(x_1, x_2, \ldots, x_n)$ that was moderately ill conditioned $\kappa(G) = 3.7 \cdot 10^{18}$. The choice of alternating sign right hand side guaranteed that the solution will be computed with no subtractive cancellation with Algorithm 4.4.1.

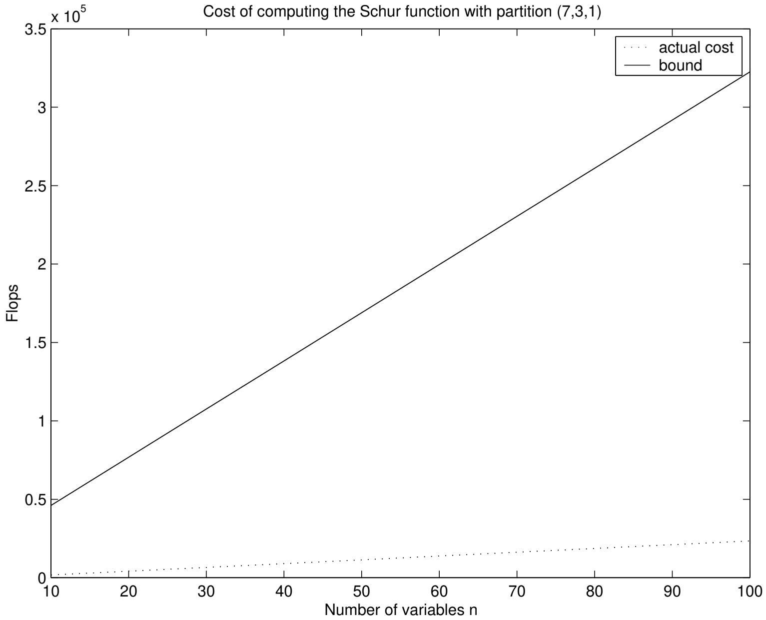
We computed the solution to $Gz = b$ using Algorithm 4.4.1 and also using 100 digit arithmetic in Mathematica [72], and got the same result to 14 digits. Note that we generated the random numbers in Matlab output them as decimal numbers to a file, read them into Mathematica, converted them to 100 decimal digit big floats, and then formed the explicit matrix which we then solved with. In other words, we more than likely made some errors in the 16th place just in forming the matrix in Mathematica. The relative gap rel_gap_{ik} between the x_i's was at least 10^{-3}, thus the problem was very well conditioned and we could expect that the solution computed by Mathematica and the one computed by Algorithm 4.4.1 will differ in not more than three digits in the last place. The results in Table 4.8 confirmed our claims that the problem is well-conditioned, even though $\text{cond}(G) = 3.7 \cdot 10^{18}$ and the traditional algorithm computed a solution that was only good in the first digit. Since Algorithm 4.4.1 computes the Schur function at every step this tests also asserts that Algorithm 4.2.1 computes the Schur function correctly.

We also tested Algorithm 4.2.1 in order to verify bound (4.20). We choose a number of random partitions λ and computed the Schur function $s_\lambda(x_1, x_2, \ldots, x_n)$ for several different values of n and confirmed the bound (4.20) as well as the linear dependence of the cost of computing the Schur function on n. For the partition $\lambda = (7, 3, 1)$ we present the results from the test in Figure 4.1.

Table 4.2: Solving a 12×12 generalized Vandermonde linear system, $\lambda = (6, 3, 2, 1, 1)$

Nodes	Mathematica	Algorithm 4.4.1	Traditional
2.0594766805425224	-1.0555833300893850e12	-1.0555833300893848e12	-1.0708496032167412e12
2.0940348689727863	4.1382711534659341e12	4.1382711534659316e12	4.1982342784471191e12
2.1568348650743552	-7.1458326298047432e12	-7.1458326298047402e12	-7.2495749151291846e12
2.3449660139029209	7.0736899709677744e12	7.0736899709677725e12	7.1765860476713262e12
2.3472440864992130	-4.3450312586851128e12	-4.3450312586851128e12	-4.4083608888447139e12
2.4163531061463885	1.6416125390612351e12	1.6416125390612349e12	1.6655874274458679e12
2.6563511105968578	-3.2574038855981763e11	-3.2574038855981757e11	-3.3050732345287665e11
2.7184145910794202	1.5677076891114130e10	1.5677076891114128e10	1.5907444718112751e10
2.7423051335090571	-2.9091221498536110e09	-2.9091221498536110e09	-2.9519596631778803e09
2.8840219496751516	4.3989708280861162e07	4.3989708280861154e07	4.4640199653213106e07
2.9550986332751350	-5.8977191662033915e05	-5.8977191662033915e05	-5.9853019293252868e05
2.9582142929074067	1.1347682330511120e02	1.1347682330511122e02	1.1517641591347063e02

Figure 4.1: The cost of computing the Schur function $\lambda = (7, 3, 1)$



4.9 Conclusions and Open Problems

We have explained the phenomenon of the Björck-Pereyra methods for all TP matrices whose initial minors are computable accurately and efficiently. We presented a new method for computing the Schur function that is exponentially faster than previous methods and only linear in the number of variables. This in turn allowed us to derive Björck-Pereyra-type methods for the class of Generalized Vandermonde matrices and proved that in many cases the methods are efficient and always accurate.

We classified a large number of structured matrices for which various accurate computations and decompositions are possible.

There are a number of open problems remaining, including the following:

1. Can we compute an SVD to high relative accuracy in polynomial time for generalized Vandermonde matrices?

2. Do efficient methods exist for solving linear systems, inverting or computing SVDs to high relative accuracy for the class of Totally Positive matrices in general?

3. Can we fill more of the entries in Table 4.1?

4.10 Matlab Implementation

Schur Function Implementation

The following Matlab code is an implementation of Algorithm 4.2.1.

```
% function f=schur1(l,x). Computes the Schur function corresponding
% to partition l with variables specified in x. It is named schur1
% to distinguish it from the Matlab function schur

function f=schur1(l,xx);
  global x sh Lp Lmax n lma
  x=xx;
  n=length(x);
  Lp=0;
  while (length(l)>Lp)&(l(Lp+1)>0)
     Lp=Lp+1;
  end % number of parts in l
  l=l(1:Lp);
  Lmax=l+1;
```

```
  sh(prod(Lmax),1,n)=-1; % allocate space
  if Lp>0
     lma(Lp)=1;
     for i=Lp-1:-1:1
        lma(i)=lma(i+1)*Lmax(i+1);
     end
     initialize(1,1,0);
     f=sch(1,n,1,nmu(1));
  else
     f=1;
  end
  clear x Lmax sh Lp n lma;
```

Following we have the recursive procedure that implements lines 100-270 of Algorithm 4.2.1.

```
% Given vector x and partition, represented by an integer l, sch(m,j,k,l)
% computes the Schur function s_l(x_m,...,x_j), where the partitions
% mu, such that l-mu is horizontal strip, are generated to have
% l(i)=mu(i) for i=1,2,...,k. The parameter k is designed to keep
% track of the recursion. from outside the function should be called
% with sch(m,j,1,l)

function f=sch(m,j,k,l);
  global x Lp Lmax n sh lma
  s=1;
  if (m<=j)&(l>0)
    t=sh(l+1,m,j);
    if (k==1)&(t>-1)
      s=t;
    elseif part(l,j-m+2)>0
      s=0;
    elseif part(l,j-m+1)>0
      t=part(l,j-m+1);
      s=s*prod(x(m:j))^t *sch(m,j,1,l-t*sum(lma(1:j-m+1)));
    elseif j==m
        s=s*x(m)^l(1);
    else
      i=k;
      s=sch(m,j-1,1,l);
      if (k==1)&(part(l,1)>part(l,2))
        i=2;
        s=s+sch(m,j,1,l-lma(1))*x(j);
      end
      while part(l,i)>0
```

```
        if part(l,i)>part(l,i+1)
          if part(l,i)>1
             s=s+x(j)*sch(m,j,i,l-lma(i));
          else
             s=s+x(j)*sch(m,j-1,1,l-lma(i));
          end
        end
        i=i+1;
      end
    end
    if k==1
      sh(l+1,m,j)=s;
    end
  end
  f=s;
```

Finally, we have the functions `initialize`, `part` and `Nmu` – lines 300-520 of Algorithm 4.2.1.

```
function f=initialize(t,k,l);
  global Lmax Lp sh n lma
  if k<=Lp
     m=Lmax(k)-1;
     if (k>1)
        m=min(m,part(l,k-1));
     end
     for i=1:m
        l=l+lma(k);
        sh(l+1,1:t,1:n)=-ones(t,n);
        initialize(t,k+1,l);
     end
  end

% given an integer pn that represents a partition, this function returns
% part i of the partition in O(1) time
function f=part(pn,i)
   global lma
   if i>length(lma)
      f=0;
   else
      if i==1
         f=floor(pn/lma(i));
      else
         f=floor(mod(pn,lma(i-1))/lma(i));
      end
```

```
   end

% nmu computes the unique integer that represents the partition l
function f=nmu(l)
   global Lmax Lp
   f=0;
   for i=1:Lp
      f=Lmax(i)*f;
      if i<=length(l)
         f=f+l(i);
      end
   end
```

Björck-Pereyra-type Algorithm for generalized Vandermonde Matrices

The following code implements Algorithm 4.4.1 – the Björck-Pereyra-type algorithm for solving generalized Vandermonde linear systems.

It takes as an input three row vectors - the partition λ, the vector of x_i's and the right hand side b. The first part is straight forward - it uses the bidiagonal decomposition (4.22) of a generalized Vandermonde matrix to compute the solution. The bidiagonal factors are computes using the formulas (4.29), (4.30) and (4.31). The underlying function `sch` computes the Schur function as explained in Algorithm 4.2.1 in Section 4.2.

```
% function f=gvand(l,x,b). Solves Gf=b, where G_{ij}=x_i^{j-1+l(n-j+1)} is
% a generalized Vandermonde matrix, corresponding to a partition l
% if length(b) differs from length(b), then the longer is truncated
% to match the shorter l may be specified with the nonzero parts only,
% it is automatically padded with zeros

function f=gvand(l,xx,b);
  global x sh Lp Lmax n lma

  x=xx;
  lma=[];

  if length(x)>length(b)
     x=x(1:length(b));
  end
  if length(x)<length(b)
     b=b(1:length(x));
  end
  n=length(x);
```

```
Lp=0;
while (length(l)>Lp)&(l(Lp+1)>0)
   Lp=Lp+1;
end % number of parts in l
l=l(1:Lp);
Lmax=l+1;

if Lp>0
  sh(prod(Lmax),n,n)=-1; % allocate space

  lma=ones(1,n);
  for i=Lp-1:-1:1
     lma(i)=lma(i+1)*Lmax(i+1);
  end
  initialize(n,1,0);
end

l=[l zeros(1,n-length(l))];
d=ones(n);
for i=2:n
   for j=i-1:-1:1
      d(i,j)=d(i,j+1)*(x(i)-x(j));
   end
end

NmuL=zeros(n+1);
for i=1:n
   NmuL(i,i)=l(i)*lma(1);
   for j=i+1:n
      NmuL(i,j)=NmuL(i,j-1)+l(j)*lma(j-i+1);
   end
end

for k=1:n-1
  for i=n-1:-1:k
    b(i+1)=b(i+1)-b(i)*sch(i-k+2,i+1,1,NmuL(n-k+1,n))/ ...
                       sch(i-k+1,i,1,NmuL(n-k+1,n))* ...
                       sch(i-k+1,i-1,1,NmuL(n-k+2,n))/ ...
                       sch(i-k+2,i,1,NmuL(n-k+2,n))* ...
                       d(i+1,i-k+2)/d(i,i-k+1);
  end
end
```

```
  for i=1:n
    b(i)=b(i)*sch(1,i-1,1,NmuL(n-i+2,n))/sch(1,i,1,NmuL(n-i+1,n))/d(i,1);
  end

  for k=n-1:-1:1
    for i=k:n-1
       b(i)=b(i)-b(i+1)*x(k)*sch(1,k-1,1,NmuL(n-i+2,n-i+k))/ ...
                            sch(1,k-1,1,NmuL(n-i+1,n-i+k-1))/ ...
                            sch(1,k,1,NmuL(n-i+1,n-i+k))* ...
                            sch(1,k,1,NmuL(n-i,n-i+k-1));
    end
  end
  f=b;
  clear x Lmax sh Lp n lma;
```

Inverse of a TP Vandermonde Matrix

Here is a Matlab implementation of Algorithm 4.6.1 from Section 4.6.

```
function B=invvand(x);
global s n

n=max(size(x));

nn=floor(n/2);

s(n,n)=0;
s(:,1)=ones(n,1);

d=schurtable(x(n),x(1:n-1),2);
B(:,n)=s(n-1,n:-1:1)'/d;
for k=2:nn
   d=schurtable(x(n-k+1),[x(1:n-k) x(n-k+2:n)],n-k+1);
   B(:,n-k+1)=s(n-1,n:-1:1)'/d;
end

d=schurtable(x(1),x(n:-1:2),2);
B(:,1)=s(n-1,n:-1:1)'/d;
for k=n-1:-1:nn+1
        d=schurtable(x(n-k+1),[x(n:-1:n-k+2) x(n-k:-1:1)],n-k+1);
        B(:,n-k+1)=s(n-1,n:-1:1)'/d;
end

for k=1+mod(n,2):2:n
    B(k,:)=-B(k,:);
```

```
end


function d=schurtable(x,z,m)
global s n
  s(1,2)=z(1);
  for i=m:n-1
    for j=2:i+1
      s(i,j)=s(i-1,j)+s(i-1,j-1)*z(i);
    end
  end

  d=1;
  for l=1:n-1
     d=d*(x-z(l));
  end
```

Chapter 5

Stability of Rounded off Inverses Under Iteration A Generalization of a Theorem by Harold Diamond

Let f be a monotone and convex (or concave) function on a real interval and let g be the inverse function. We fix a positive integer N and for any expression x we denote by $[x]$ the value of the expression x rounded to N significant binary digits. We call the number $\epsilon = 2^{-N}$ machine precision and assume that the arithmetic is performed in accordance with the IEEE 754 Standard for binary floating point arithmetic [4] in standard rounding mode, i.e. each result is rounded to the nearest even floating point number. We use capital letters for the rounded-off floating point values of the lowercase variables or functions, e.g. $X = [x], F(x) = [f(x)]$, etc. Let $H(x) = G(F(x))$, the composition of F and G. For any X it is shown that

$$H(H([x])) = H(H(H([x])))$$

and there exist simple strictly convex functions for which $H([x]) \neq H(H([x]))$.

This is a generalization of a theorem by Harold Diamond [26], which required strict convexity (or concavity of f), but did not impose restrictions on how the arithmetic rounds the ambiguous (or halfway) cases.

Finally we prove that $H([x]) = H(H([x]))$ for the following functions: $f(x) = x + C$; $f(x) = C \cdot x$; $f(x) = C/x$; $f(x) = e^x$ and $f(x) = \ln(x)$.

5.1 Motivation

Correctly Rounded Transcendental Functions?

We address the issue of correctly rounding the computed values of transcendental functions for floating point arguments. In the event the function value is too close to a halfway point between two floating point numbers it may take arbitrarily long to compute the last digit correctly. In this situation we are faced with the "Table Maker's Dilemma" – does one stop the computations after computing a fixed number of extra digits and round, possibly getting the last digit wrong, or continue the computations for a finite, but not a priori known time?

Why is it hard to correctly round transcendental functions?

Computing a value for a transcendental function f for a floating point argument x involves computing an approximate value $\hat{f}(x)$ and an error bound η such that $|f(x) - \hat{f}(x)| \leq \eta|f(x)|$. To ensure that all digits of $\hat{F}(x)$ (the floating point representation of $\hat{f}(x)$), except possibly the last one, are computed correctly, we ask that η be less than a unit in the last place.

Getting the last digit correctly, however, can be a challenge when $f(x)$ is very close to a halfway point h between two floating point numbers. Then we need an error bound that is no longer a priori fixed but depends on $f(x)$

$$\eta < |f(x) - h|/|f(x)|,$$

where h is the halfway point between the two floating point numbers surrounding $f(x)$. Thus to ensure that we round the result correctly we may have to compute an arbitrary number of extra digits.

Why do we want correctly rounded transcendental functions?

There can be no argument against a correct result if it can be computed at a reasonable cost. If the cost is unreasonable, then we should address the drawbacks of computing an inexpensive result that is almost always correct, except in a very few instances when only the last digit may be wrong. We will call this an *almost correctly rounded* result.

The properties that a correctly rounded function will possess, that the almost correctly rounded may not, include:

1. Sign symmetry ($f(a) = -f(-a) \Rightarrow \hat{F}(-a) = -\hat{F}(a)$).

2. Weak monotonicity ($f(a) < f(b) \Rightarrow \hat{F}(a) \leq F(b)$).

3. An error bounded by 0.5 *units in the last place (ulps)* instead of (say) 0.501 ulps.

4. Consistency ($\hat{F}(a)$ identical) across different platforms and implementations.

Unless a function is correctly rounded, sign symmetry and especially weak monotonicity could be a challenge. But sign symmetry and weak monotonicity can be ensured for almost correctly rounded functions by using a formula to compute them that has that property. The weak monotonicity is of questionable value anyway, since the very first floating point computation following the evaluation of a transcendental function may destroy the weak monotonicity. For example $f(x) = x - \sin x$ may not be monotonic for correctly rounded $\sin x$ [48].

There is no evidence that a slightly wider error bound (0.501 ulps instead of 0.5 ulps) will undermine the proof of any theorem (more on that later).

Consistency, however, is another matter. Today different machines, or just different math libraries can return different values for the same transcendental function evaluated at the same argument. In a heterogeneous parallel computing environment this could result in different processes following different paths – a problem that may be extremely difficult to debug [10, 11, 21].

Therefore, specifications need to be developed for computed transcendental functions if for no other reason then to at least ensure consistency.

History of correct rounding

The point of floating point computations is to establish a relationship between the input and the output.

When this relationship is algebraic it is, in principle, possible to decide whether the desired relationship between the input and the output exists. Such a decision can be made in a in a finite amount of time (albeit possibly doubly exponential in the length of the expression) using known means, namely the Tarski's resolution procedure [57].

No algorithm is known or believed to exist for deciding if certain relationships exist between the input and the output of a transcendental function evaluation in general. In other words we have no decision procedure in general for deciding if a computed transcendental function is correctly rounded.

The current best method that is known to us for checking if a transcendental function is correctly rounded is to exhaust all possible values of the argument, after possibly eliminating part of the domain on which there can be no rounding ambiguity [52]. Although this method can be applied to the exponential and logarithmic functions in double precision, its cost grows exponentially with the width of the precision and is unlikely to be feasible for wider precisions or functions without tight domain or range.

Specifications for computed transcendental functions?

We have established the following.

1. Correctly rounded transcendental functions may be very expensive to compute, with no a priori bound on the cost.

2. Even if we had an implementation of a transcendental function that we suspect to be correctly rounded, we have no known means of proving that it is such.

3. Almost correctly rounded functions will likely be adequate for practical purposes if they can be implemented consistently, and sign symmetry and weak monotonicity are guaranteed.

Therefore we look at weaker specifications for computed transcendental functions, less perfectionist than "correctly rounded" that may be adequate for practical purposes. The list of these properties should include at least:

1. Sign symmetry.

2. Weak monotonicity.

3. An error bounded by (say) 0.501 ulps.

and at most

4. All properties (independent of number of significant digits) that a computed transcendental function would have if it were correctly rounded.

Obviously properties 1–3 above are insufficient to ensure consistency and property 4 may be too much to ask.

In looking at possible candidates for property 4. above we come across Harry Diamond's theorem. It is an important property that guarantees that drift will not occur in repeated computations. Thus we would like to raise the following questions:

1. Should the specifications for computed transcendental functions be such that the latter satisfy the Harry Diamond's theorem?

2. How can one verify that a computed transcendental function conforms to the Harry Diamond's theorem?

3. Is the Harry Diamond's theorem a property of correctly rounded transcendental functions that if forced upon the computed (almost correctly rounded) transcendental functions can help us achieve consistency?

5.2 Introduction

We generalize Harold Diamond's theorem [26] by making only a small modification of his original proof to allow for the functions to have flat sections, i.e. be non-strictly convex or concave. In order to make this presentation self contained we include Harold Diamond's original proof with minor changes. Most of the contents of this chapter are taken verbatim from [26], except for section 5.6.

For those readers who are familiar with the original paper, here are the changes and contributions we made:

- We do not assume strict convexity (or concavity)

- We require that the arithmetic rounds to the nearest floating point number, and when this is ambiguous (in the halfway cases) we require that the rounding be to the nearest even. This is the default mode of the current IEEE Standard 754 for binary floating point arithmetic.

- The only changes we made in the proof are replacing some of the strict inequalities with non strict inequalities, and using the "rounding to even" properties to prove that everything will still round the same way as in the original paper.

- The identity $H(H((X)) = H(X)$ holds for the following functions $f(x) = x + C$; $f(x) = C \cdot x$; $f(x) = C/x$; $f(x) = e^x$ and $f(x) = \ln(x)$.
- We also provide an example of a simple quadratic function that will require three steps to converge for certain values of the argument.

We continue with the introduction following Harold Diamond.

Let f denote a continuous strictly monotonic real valued function defined on some real interval $\mathcal{I}$ and let g denote the inverse function. For R a radix, N a positive integer, x a real number and ϕ any real valued function we will denote by capital letters the rounded off values of the variables and the functions. E.g. $X = [x]$ and $\Phi = [\phi(x)]$ will denote the composition of the function $x \mapsto \phi(x)$ followed by rounding off to N significant figures in the base R. We will use capital letters instead lowercase letters for variables whose value we know to be a floating point number.

Let $h = h^{(1)} = G \circ F$ be the composition of F and G. Since h can only take floating point values we have $H = h$. From now on we will use H instead of h to attest to the fact that $H(x)$ is a floating point number for any real number x.

We define $\mathcal{I}^* = \mathcal{I}^{*(1)}$, the domain of $H^{(1)}$, to be the set of $x \in \mathcal{I}$ for which $F(x) \in f(\mathcal{I})$. Similarly, for $k \geq 2$ we define $H^{(k)} = H \circ H^{(k-1)}$ on the domain $\mathcal{I}^{*(k)}$ consisting of all $x \in \mathcal{I}^{*(k-1)}$ for which $F \circ H^{(k-1)}(x) \in f(\mathcal{I})$. The object of this article is to see whether H is, in some sense, an identity function on numbers having at most N significant digits. We consider the recursion

$$[x],\ \ F([x]),\ \ H([x]),\ \ F(H([x])),\ \ H^{(2)}([x]),\ldots \tag{5.1}$$

and ask whether any of the equations

(1A) $H([x]) = [x]$

(1B) $F(H([x])) = F([x])$

(1C) $H^{(2)}([x]) = H([x])$

(1D) $F(H^{(2)}([x])) = F(H([x]))$

(1E) $H^{(3)}([x]) = H^{(2)}([x])$

... are valid for any real number x.

In concrete terms (which motivated the investigation), suppose we have a machine which very accurately performs a functional operation and then rounds off its results to N significant figures. We enter $X = [x]$ and successively form

$$F(X), G(F(X)), F(G(F(X))), \ldots$$

and ask whether, after a fixed number of steps, we get values repeated every other time.

Let us first discuss the rounding rules. To fix our ideas, from here until the statement of the theorem assume that the radix $R = 10$, the number of significant figures $N = 2$, and the interval $\mathcal{I} = \{t : .1 < t < 1\}$. Any rounding rule $[\cdot]$ assigns the nearest two digit decimal in case it is unique. Specific rounding rules assign one of the two nearest values in the remaining cases. The ambiguous cases in $\mathcal{I}$ are the 90 numbers $.105, .115, \ldots, .985, .995$.

It can happen that none of the equations $(1\ldots)$ holds. We give two examples of this phenomenon.

First take the rounding rule to be

$$[x] = .01\lfloor 100x + .5\rfloor \quad (x \in \mathcal{I}),$$

where $\lfloor t\rfloor$ = the greatest integer not exceeding t. This shifts each of the 90 ambiguous numbers upward. Let $f(t) = t + .005$. Then for $X = x = [x] \in \mathcal{I}$ we have

$$F(X) = X + .01,\ G(X) = X,\ H(X) = X + .01.$$

This example exhibits the so-called drift phenomenon. Drift can be eliminated in the present case by changing the rounding rule to eliminate the bias toward $+\infty$. One such rounding rule is "round to even": $[.105] = .10$, $[.115] = .12$, $[.125] = .12$, etc. Details and further references can be found in [63].

Our second example is more extreme in that drift occurs regardless of how one rounds in the ambiguous cases. Let ϕ be a function of period .01 which satisfies $\phi(0) = \phi(.01) = -.001$, $\phi(.009) = -.009$, and ϕ is linear on each of the intervals $(0, .009)$ and $(.009, .01)$. Let $f(t) = t + \phi(t)$ for $t \in \mathcal{I} = (.1, 1)$. For $X = x = [x] \in \mathcal{I}$ we have

$$F(X) = X,\quad G(X) = X + .01,\quad H(X) = X + .01.$$

These examples suggest that a convexity condition is needed to obtain positive results. It is easy to see that the relation (1A) still need not be true. For example, let $f(t) = t^2$ for

$0 < t < \infty$ and $x = .34$. Thus $F(.34) = .12$, $G(.12) = .35$. We shall show later that the relations (1B)–(1D) can also fail for a convex monotonic function f. However, stability is achieved for further iterates as we will show.

From now on we assume that the rounding is to the nearest floating point number, and when that is ambiguous (i.e. in the halfway cases) the rounding is to the nearest even, i.e. to the nearest floating point number that has an even least significant digit.

What this means in particular is that if X is a floating point number that is not a power of the radix, and y is a real number that rounds off to X under the specified rounding rules then any number z such that

$$|z - X| \leq |y - X|$$

must also round off to X. The difference that the "rounding to even" makes here is that we are able to put non strict inequality above. Unless we have "rounding to even" the halfway cases may drift by jumping from one off to the next and so on. In order to prevent drift we must either require strict convexity (or concavity) as in [26] or require that the rounding be done to the nearest even as we do here.

Here is an outline of the argument. We start with a floating point number X, we take $y = g(F(X))$. If $Y = X$, $f(Y)$ will have to round to $f(y)$ because of monotonicity and we're done. If y rounds away from x we use convexity to show that that $f(Y)$ will round to $f(y)$ except if $|f(y)| = R^k$ for some k. The problem with $|f(y)| = R^k$ is that the floating point numbers to the left of R^k are R times as dense as they are to the right. At this point $f(Y)$ being no further from $f(y)$ than $f(X)$ is does not imply they both round to $f(y)$. If this is the case we simply take another step and show that on the next step we are away from a power of the radix (unless we stabilize sooner).

5.3 Statement of Results

Theorem 6 *Let $\mathcal{I}$ denote a real interval and f a real valued monotone function which is convex or concave (perhaps not strictly) on $\mathcal{I}$. Let R denote a radix and N a positive integer such that $R > 3$ if $N = 1$, and let $[\cdot]$, $H^{(k)}$ and $\mathcal{I}^{*(k)}$ $(k = 1, 2, \ldots)$ be as defined above. Then (1E) holds for all $x \in \mathcal{I}^{*(3)}$.*

Remarks.

A. The result depends on the rounding rule. The rounding mode must be set to "rounding to nearest even" in the ambiguous (halfway) cases.

B. For simplicity in stating the theorem we have used the same radix R and number of significant figures N for both the domain and range of the function f: Actually, this is unnecessary for the proof, and so our result could be restated with radices R and R' and round off to N and N' digits. We need only assume $R > 3$ if $N = 1$ and $R' > 3$ if $N' = 1$. See [56] for discussion of a related linear problem. The requirement $R > 3$ if $N = 1$ is slightly stronger than the $R^N \geq 3$ requirement in the original paper [26]. We view this as insignificant since our proof only excludes base three single digit arithmetic.

C. For some special functions a stronger theorem holds. For example, one can show that if $f(x) = 1/x$ and the radix equals 10, then (1B) holds generally and (1A) holds for each interval $\mathcal{I}$ of the form $[10^k, 10^{k+0.5}]$, where k is any integer and N is any positive integer.

5.4 Proof of the Theorem

We shall use the following notation throughout. Let $x = X = [x] \in \mathcal{I}^{*(3)}, y = g(F(X))$ (so that $f(y) = F(X)$), and $Y = [y] = H(X)$. We can assume that X, y and Y are all distinct, for otherwise the theorem holds trivially.

We shall say that $[0] = 0$. Of course we have $[a] \neq 0$ if $a \neq 0$. If $f(a) = 0$ for some $a = [a] \in \mathcal{I}$, then $F(a) = 0, g(F(a)) = a$, and $h(a) = a$. In this case the theorem holds. Similar remarks apply if $g(b) = 0$ for some $b = [b] \in f(\mathcal{I})$. Thus, we can and shall exclude any occurrence of 0 except as a possible value of X in case $0 \in \mathcal{I}^{*(3)}$.

We begin by considering the possible orderings of X, y, Y. Since $Y = [y], y$ cannot be closer to X than it is to Y. In other words,

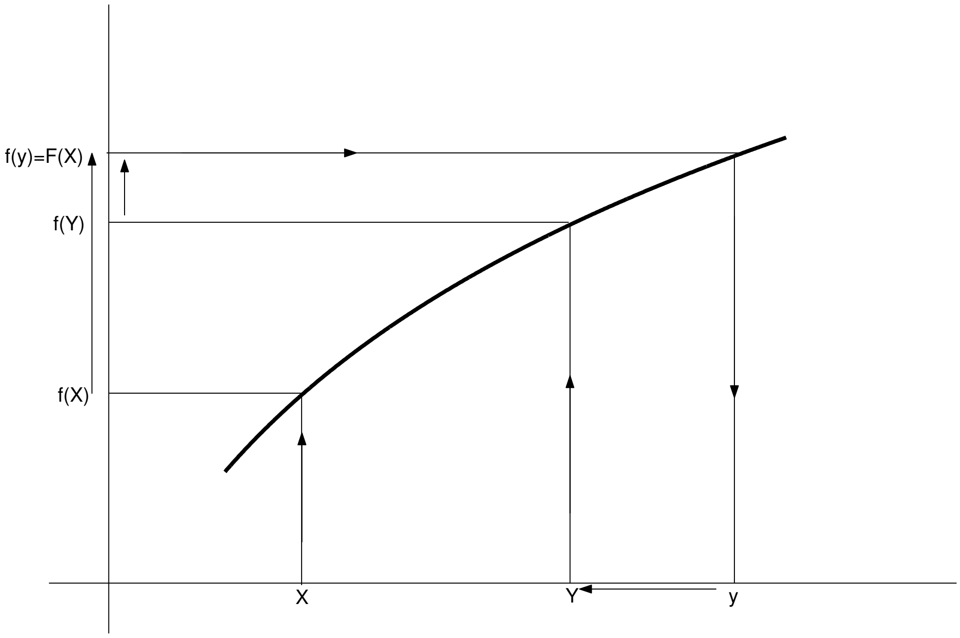
$$|y - Y| \leq |X - y|; \tag{5.2}$$

and the configurations $Y < X < y$ and $Y > X > y$ are impossible. Now we have the simple

Lemma 5.4.1 *Let f be monotone, $X = [x], y = g(F(X))$, and $Y = [y]$. Assume that Y lies between X and y. Then $F(X) = F(Y)$.*

Proof: Since f is monotone, $f(Y)$ lies between $f(X)$ and $f(y)$. Since $f(X)$ rounds off to $F(X) = f(y)$, so does $f(Y)$. See also Figure 5.1

Figure 5.1: Y is between X and y



□

Thus, if we have $X < Y < y$ or $X > Y > y$, then $F(X) = F(Y) = F(H(X))$. Applying $H \circ G$ to each side of this equation, we obtain $H^{(2)}(X) = H^{(3)}(X)$; i.e. the theorem holds in this case. (When in subsequent arguments we obtain the formula $F(X) = F(Y)$, we shall simply say that the theorem is proved in that case.)

We turn our attention to the remaining possible arrangements $X < y < Y$ and $X > y > Y$. There are two main cases for us to consider:

$$\left|\frac{f(Y) - f(y)}{Y - y}\right| \leq \left|\frac{f(y) - f(X)}{y - X}\right| \tag{5.3}$$

or

$$\left|\frac{f(Y) - f(y)}{Y - y}\right| > \left|\frac{f(y) - f(X)}{y - X}\right| \tag{5.4}$$

We remark that (5.3) holds if

- $X < y < Y, f \uparrow, f$ concave, or
- $X < y < Y, f \downarrow, f$ convex, or
- $Y < y < X, f \uparrow, f$ convex, or

- $Y < y < X, f \downarrow, f$ concave,

and (5.4) holds in the remaining four cases. It is not necessary to treat all these cases individually.

We assume first that (5.3) holds. Later we shall treat (5.4) by transforming it into (5.3). We have by (5.2) and (5.3) that

$$|f(Y) - f(y)| \leq |f(y) - f(X)|, \tag{5.5}$$

i.e.

$$|f(Y) - F(X)| \leq |F(X) - f(X)|.$$

This inequality implies that $F(Y) = F(X)$ (and hence the theorem holds) provided that either

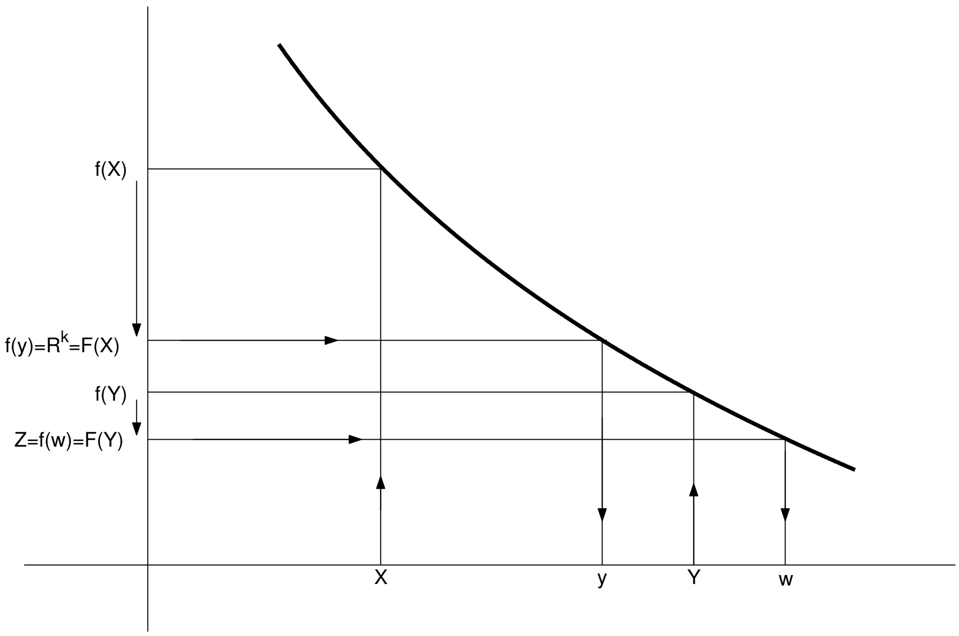
$$|F(X)| \neq R^k \text{ for any integer } k \tag{5.6}$$

or

$$|F(X)| = R^k \text{ for some } k \text{ and } |f(X)| < |F(X)| < |f(Y)|. \tag{5.7}$$

Now we shall prove the theorem in case (3) holds and $|f(X)| > |F(X)| = R^k > |f(Y)|$. We need a bit more notation. Let $Z = F(Y)$ and $w = g(Z)$, i.e. $f(w) = F(Y)$.

Figure 5.2: The case $X < y < Y < w$



We claim that $f(X), f(y), f(Y)$, and $f(w)$ all have the same sign. $\mathrm{sgn} f(X) = \mathrm{sgn} f(y)$ since $f(y) = F(X)$. By (5.5) and the round off rules we have

$$|f(Y) - f(y)| \leq |f(y) - f(X)| \leq \frac{1}{2} \cdot R^{k+1-N} \tag{5.8}$$

Since $|f(y)| = R^k$, it follows that $\mathrm{sgn} f(Y) = \mathrm{sgn} f(y)$. For $f(w) = F(Y)$ we use the fact that $|f(Y)| < R^k$ and the round off rules to conclude that $|f(w) - f(Y)| \leq \frac{1}{2} \cdot R^{k-N}$. Thus

$$|f(w) - f(y)| \leq |f(w) - f(Y)| + |f(Y) - f(y)| \leq R^{k-N} \frac{R+1}{2}; \tag{5.9}$$

and, as in the case of $f(Y)$, we deduce that $f(w)$ and $f(y)$ have the same sign.

Next we give an upper bound for $|f(w)|$. We have $|f(w) - f(Y)| \leq |f(Y) - f(y)|$, since $f(Y)$ rounds off to $f(w)$ rather than to $f(y)$. Also, we have $|f(w) - f(Y)| = |f(y)| - |f(Y)|$ since $|f(y)| > |f(Y)|$ and $\mathrm{sgn} f(y) = \mathrm{sgn} f(Y)$. Thus

$$|f(w)| \leq |f(Y)| + |f(w) - f(Y)| \leq |f(Y)| + |f(y) - f(Y)| = |f(y)|.$$

We conclude this part of the argument differently according to the relative size of $|f(w)|$ and $|f(Y)|$. First, suppose $|f(w)| \geq |f(Y)|$. We have $f(w)$ included between $f(Y)$ and $f(y)$; and we can apply the lemma with $f(y), f(Y), f(w)$, and g in place of X, y, Y, and f, respectively. We deduce that $W = [w] = [y] = Y$, i.e. $H^{(2)}(X) = H(X)$, and the theorem is proved in this case.

Now suppose that $|f(w)| < |f(Y)| < |f(y)| = R^k < |f(X)|$. We repeat the foregoing analysis with X, y, and Y replaced by Y, w, and W, respectively. We show that the process must now terminate by the time we reach (5.6).

In case W lies between Y and w, then by the lemma $F(Y) = F(W)$, i.e.

$$F(H(X)) = F(H^{(2)}(X)); \tag{5.10}$$

and the theorem is proved in this case.

Next suppose that W does not lie between Y and w. Then as in (5.2), Y cannot be between W and w, so w lies between Y and W. Also, $f(X), f(y), f(Y)$, and $f(w)$ lie in linear order, since their absolute values do and these numbers are all of one sign. It follows by monotonicity that X, y, Y, and w and hence X, y, Y, w, and W lie in linear order.

The assumed inequality (5.3) implies that f is not steeper near Y than it is near X. It follows that f is not steeper near W than it is near Y. Thus (5.3) holds with X, y and Y

replaced by Y, w, and W, respectively. The analogue of (5.5) now holds:

$$|f(W) - F(Y)| \leq |F(Y) - f(Y)|.$$

It follows that $F(W) = F(Y)$, i.e. (5.10), and hence the theorem is established, provided that

$$|f(w)| = |F(Y)| \neq R^l \text{ for any integer } l. \tag{5.11}$$

Now (5.9) and the fact that $R > 3$ if $N = 1$ imply that

$$|f(w)| \geq |f(y)| - |f(y) - f(w)| \geq R^k \left(1 - \frac{R+1}{2R^N}\right) > R^{k-1}.$$

Thus $R^{k-1} < |f(w)| < R^k$, hence (5.11) holds. This concludes that the proof of the theorem in case f satisfies (5.3).

Now consider the other main case, in which (5.4) is assumed to hold. That is, the graph of f gets steeper as we move from X toward Y.

Since y lies between X and $Y, f(y)$ lies between $f(X)$ and $f(Y)$ by monotonicity. By the round off of $f(Y)$ we have

$$|f(w) - f(Y)| = |F(Y) - f(Y)| \leq |f(Y) - f(y)|.$$

Thus $f(y)$ lies between $f(X)$ and $f(w)$, so y lies between X and w.

It follows from (5.4) and the above that f has a steeper secant line over the interval between w and Y than over the interval between y and Y, i.e.

$$\left|\frac{f(w) - f(Y)}{w - Y}\right| > \left|\frac{f(Y) - f(y)}{Y - y}\right|.$$

It is convenient here to call $f(y) = \xi, f(Y) = \eta$, and $f(w) = \theta$. The preceding inequality can be rewritten as

$$\left|\frac{g(Z) - g(\eta)}{\theta - \eta}\right| < \left|\frac{g(\eta) - g(\xi)}{\eta - \xi}\right|.$$

Now we have the situation of (5.3) with ξ, η, θ, and g in place of X, y, Y, and f. It follows from the preceding argument that

$$G(F(G(F(G(\xi))))) = G(F(G(\xi)))$$

or since $\xi = f(y) = F(X)$, we have $H^{(3)}(X) = H^{(2)}(X)$.

The proof of the theorem is now complete in case f satisfies either of the inequalities (5.3) or (5.4).

5.5 An Example

We give an example of a monotone concave function for which equations (1A)-(1D) fail to hold. Let $R = 10, N = 2, f(t) = 115 - 35/(t - 97)$ on $\mathcal{I} = (98, \infty)$, and $g(t) = f^{-1}(t) = 97+35/(115-t)$ for $t < 115$. We have $F(110) = 110, G(110) = 100, F(100) = 100, G(100) = 99, F(99) = 97$ or 98 (according to the round off rule used). Of course $G(97) = G(98) = 99$ as predicted by the theorem.

5.6 Stability of Some Simple Functions

We now prove (1C) for the following functions $f(x) = x + C$; $f(x) = C \cdot x$, $f(x) = C/x$, $f(x) = e^x$ and $f(x) = \ln(x)$ under the conditions of IEEE Standard 754 for binary floating point arithmetic in default rounding mode – to the nearest even floating point number.

The functions $f(x) = C \cdot x$ and $f(x) = C/x$

Negation is an exact operation, so without loss of generality we can assume $C > 0$, i.e. both functions are convex.

For $f(x) = C \cdot x$ we have $F(X) = C \cdot X \cdot (1 + \delta)$ where $|\delta| < 2^{-N}$ and $H(X) = G(F(X)) = X \cdot (1 + \delta)$. Since $|\delta| < 2^{-N}$ we have $H(X) = X$, except if $|X| = 2^k$ for some k and $\delta < -2^{-N-1}$ in which case X will round to Y where $Y = \pm(2^k - 2^{k-N-1})$. Now $w = H(Y) = Y(1+\delta_1)$, where $|\delta_1| < 2^{-N}$. Now since Y is not a power of 2 we can conclude that $[w] = Y$.

The proof goes by analogy in the case of $f(x) = C/x$.

The function $f(x) = x + C$

See also [63]. Again we follow the proof of the theorem until we reach the case $|f(w)| < |f(Y)|$. Since $f(a) - f(b) = a - b$ for any a and b we have $|f(w) - f(Y)| < |f(Y) - f(y)|$. Note the strict inequality here. For we are in binary arithmetic with rounding to even there are no other floating point numbers between $f(y)$ and $f(w)$. Also $f(Y)$ could not have been halfway between $f(y) = \pm 2^k$ for some k and $f(Y)$ for it would have rounded to "even", i.e. to $f(y) = \pm 2^k$. Therefore $|w - Y| < |Y - y|$, i.e. $[w] = Y$ unless possibly in the case $|Y| = 2^s$ for some integer s.

Let $|Y| = 2^s, |f(y)| = 2^k$ and $|F(Y)| = 2^k(1-2^{-N})$. $|F(Y)|$ is string of N ones and the exact result of $|f(Y)| = |Y + C|$ must have had a zero as $N+1$-st bit for $|F(z)|$ to equal $2^k(1-2^{-N})$.

We consider several cases:

If $C > 0$, $0 < w < Y < y < X$ then $Y = 2^s, f(y) = 2^k$, $k > s$ and $F(Y) = 2^k(1-2^{-N}) = \underbrace{111\ldots 11}_{N} \times 2^{k-1}$. $f(Y) = x + C = \underbrace{111\ldots 110}_{N+1} xxx\ldots \times 2^{k-1}$ and since $Y = 2^s$, $s \geq k - N$ we must have $C = 2^k(1-2^{-N}) - 2^s$ exactly, which means $F(Y) = f(Y)$ and $g(F(Y)) = Y$, i.e. $H^{(2)}(X) = H(X)$ and we are done in this case.

The case $C < 0$, $0 > w > Y > y > X$ is done by analogy.

The only remaining case that we need to consider is $C < 0$, $0 < w < Y < y < X$ for the case $C > 0$, $0 > w > Y > y > X$ will follow by analogy.

Again we need to split this into several cases. If $-C/2 \leq X \leq -2C$ then $f(X) = X + C$ is computed exactly and we are done. If $X > -2C$ then $y - X = f(y) - f(X) \leq 2^{k-N} < X(1-2^{-N})$. Therefore there are no floating point numbers between Y and x. Now we have $Y - w < y - Y < X - y$, therefore $2(Y - w) < y - Y$ and since we are in binary arithmetic w must round to Y, and we are done.

If $0 < X < -C/2$ we must have $0 < X < y < Y < w$. Now $|w - Y| < |y - Y|$ so w must round off to Y and we are done.

The exponential function $f(x) = e^x$

We first consider the case $|X| \geq 1 + 2\epsilon$. We have $F(X) = e^X(1-\zeta)$ where $|\zeta| < \epsilon$ and therefore $\ln(e^X(1-\zeta)) = X + \ln(1-\zeta) = X - \zeta + \zeta^2/2 - \ldots$. We have $|-\zeta + \zeta^2/2 - \ldots| < |\zeta|\sum_{i=0}^{\infty}|\zeta|^i = |\zeta|/(1-|\zeta|) < |\zeta|(1+2|\zeta|) < |X|\epsilon$. Therefore $H(X) = X$.

Let $-1 - 2\epsilon < X < 1 + 2\epsilon$, i.e. $|X| \leq 1 + \epsilon$. Then following the proof of Theorem 1, we can conclude that $F(H(X)) = F(X)$ unless

- $Y < y < X$ and $F(X) = 2^k$ and $|f(Y)| < |F(X)| < |f(X)|$. This case is only possible if $F(X) = 2^{\pm 1}$. We can easily verify that $\left[e^{[\ln(2^{\pm 1})]}\right] = 2^{\pm 1}$, i.e. $F(H(X)) = F(X)$.

- $X < y < Y < w$. We have $|f(w) - f(Y)| < |f(Y) - f(y)|$ because of the rounding rules. Coupled with the convexity of f

$$\frac{f(w) - f(Y)}{w - Y} > \frac{f(Y) - f(y)}{Y - y}$$

we get $|w - Y| < |Y - y|$, which implies $H(H(X)) = W = Y = H(X)$, unless $X < y < Y < w < 0$ and $Y = -2^k$ for some $k \leq 0$. If $k = 0$ we can easily verify that $\lceil \log(\lceil e^{-1} \rfloor) \rfloor = -1$. If $k \leq -1$ then

$$\frac{f(Y) - f(y)}{Y - y} = f'(\xi) < 1,$$

for some $\xi < 0$ and therefore $f(Y) - f(y) = f(Y) - F(X) < Y - y < \epsilon/2 < f(Y)\epsilon$, because $f(Y) > f(-0.5) > 0.5$. This implies $F(Y) = F(X)$.

The natural logarithm function $f(x) = \ln(x)$

For the natural logarithm function we have that $H(H(X)) = H(X)$ in every case in which $F(H(X)) = F(X)$ holds for the exponential function. Therefore we only need to prove $H(H(X)) = H(X)$ when $f(X) < f(y) < f(Y) < f(w)$, which means $X < y < Y < w$. We can assume without loss of generality that there are no floating point numbers between X and Y, for otherwise we could replace X with the floating point number between X and Y, closest to Y and we would still be in the same situation.

The function $f(x) = \ln(x)$ is concave and we also have that $|f(Y) - f(w)| < |f(y) - f(Y)| < |f(X) - f(y)|$. This implies $|Y - w| < |y - Y| < |X - y|$. Therefore $|Y - w| < 2|X - Y|$. Since X and Y differ by at most a unit in the last place and we are in binary arithmetic w will round to Y, which implies $H(H(X)) = H(X)$ in this case as well.

Please note that for $f(x) = \ln(x)$ and $f(x) = e^x$ we make no assumption on how the rounding is done in the ambiguous halfway cases. Since for rational arguments neither of these functions can take rational values we can never hit the halfway points for arguments that are floating point numbers.

5.7 Another Example

Here we present a simple example due to W. Kahan of an increasing, strictly convex quadratic polynomial, which will require three iterations to stabilize.

We interpolate the data in Table 5.1.

We have

$$f(x) = \frac{187}{380\epsilon}x^2 + \left(\frac{194}{95} - \frac{187}{190\epsilon}\right)x - \frac{99}{95} + \frac{79\epsilon}{80} + \frac{187}{380\epsilon}$$

For this quadratic polynomial we have $H^{(2)}(1) \neq H(1)$. Thus we have established the existence of simple functions that will require at least three steps to stabilize.

Table 5.1: Example of a function that requires three steps to converge

x	$X = 1$	$y = 1 - \epsilon - \frac{\epsilon}{17}$	$Y = 1 - \epsilon$	$w = 1 - \frac{3\epsilon}{2} - \frac{\epsilon}{17}$
$f(x)$	$1 + \epsilon - \frac{\epsilon}{80}$	1	$1 - \frac{\epsilon}{2} - \frac{\epsilon}{16}$	$1 - \epsilon$

Chapter 6

Fast Parallel Method for Estimating the Power Spectrum of the Cosmic Microwave Background Radiation

The cosmic microwave background (CMB) radiation provides us with a snapshot of the Universe as it was some 300,000 years after the Big Bang. As the earliest possible view of the Universe it is the best probe we have of conditions in the very early Universe. In particular, anisotropies in the CMB contain information on virtually all cosmological parameters of interest, including the geometry of the universe, the baryon density, the Hubble expansion rate, the cosmological constant, etc.

Realizing the extraordinary scientific potential of the CMB requires precise measurements of these tiny anisotropies over a significant fraction of the sky at very high resolution. Current methods for analysis of the CMB scale as

$$O\left(n^3\right),$$

where n is the number of pixels observed. Since the PLANCK satellite, to be launched in 2007, will observe tens of millions of pixels current algorithms will no longer be applicable even on supercomputers.

The matrices involved in the map making and power spectrum estimation of the CMB are structured, which allowed us to design new, parallel, structure exploiting algorithms which run in time

$$O\left(n^{3/2}\right)$$

and scale nearly linearly with the number of processors on parallel platforms.

The code was implemented in C on the IBM SP at Lawrence Berkeley National Laboratory (LBL) in collaboration with Julian Borrill and Christopher Cantalupo at the CMB Astrophysics Research Program at LBL. The code will be used to analyze the data to be delivered by the PLANCK satellite in 2009. We presented these algorithms at 2001 PLANCK Workshop on Image Processing held in Pisa, Italy, July 4-6, 2001 which brought together over 200 researchers from around the world working on the PLANCK satellite experiment to discuss design of the experiment, algorithms and analysis of the CMB data.

Notation

Since there are many matrices, vectors and constants involved in the computations, we list them here for easy reference.

Constants

m	Number of time samples observed
n	Number of pixels observed. Most important dimension
b	Bandwidth of the band-Toeplitz matrix M^{-1}
k	Number of multipole powers
$\mathcal{P}$	Number of processors

Vectors

d	The map; An n-vector; d_i = CMB temperature of pixel i
t	The time stream. An m vector of temperature readings

Matrices

A	An $m \times n$ pointing matrix. $A_{ij} = 1$ if pixel j is observed at time i
T	An $m \times m$ band Toeplitz inverse noise correlation matrix
N	An $n \times n$ matrix of the noise correlation matrix between the pixels
S	An $n \times n$ matrix of the CMB signal correlations between the pixels
D	$D = N + S$. An $n \times n$ correlation matrix.

6.1 Introduction

6.1.1 What is the CMB?

Looking Far Away Means Looking Back in Time

The Cosmic Microwave Background (CMB) [66, 16] provides us with a view of the early universe. We see astronomical objects as they were in the past because of the time it takes light to travel across space. When we look at the Sun, we see it as it was 8 minutes ago, because it takes light 8 minutes to travel from the Sun to us. It takes light 4 years to reach us from the next nearest star, and 2 million years to reach us from Andromeda, the nearest galaxy. Looking far away in distance is like looking far back in time. When we look at the CMB, we are looking back billions of years, to a time before the first stars and galaxies were formed, when the universe was only a few hundred thousand years old.

What can the CMB tell us about the Universe?

Measurements of the Cosmic Microwave Background (CMB) directly probe conditions of the early universe and are a powerful tool for examining properties of the universe, including:

- The geometry of the universe
- Whether the universe will expand forever or collapse
- The amount of matter in the universe
- The amount and nature of dark matter and dark energy
- The expansion rate of the universe
- The age of the universe
- The primordial seeds of galaxies and clusters

The answer to each of those questions depends on different cosmological constants. The values of those constants can be determined by information about the power spectrum of the CMB [66, 16] (as defined in section 6.1.5), and it is the power spectrum of the CMB that we would like to compute, given observations of the CMB.

How the CMB formed

About 15 billion years ago the Universe came into being by a hot Big Bang. When the universe was very young, it was much hotter and denser than the center of our Sun. As it expanded, the universe became cooler and less dense.

There were small density variations in the Universe from one place to another, which would eventually grow gravitationally to become the galaxies and clusters of galaxies we see around us today. These density variations were extremely weak, but strong enough to affect the temperature of the photons. The more dense regions correspond to hot spots in the CMB and the less dense regions correspond to cold spots in the CMB.

When the universe was roughly 300,000 years old, it had cooled enough that the protons and electrons could come together to form neutral hydrogen atoms. This made the universe transparent; the photons could travel through it without scattering off the electrons. These photons, which last interacted with electrons when the universe was 300,000 years old, form relic radiation which is today called the CMB.

Observing the CMB

The CMB has a nearly uniform temperature across the entire sky: 2.73 degrees Kelvin. Superimposed on this uniform background are tiny (about 10 parts per million) variations in temperature. This departure from uniformity is called anisotropy and is exactly what we would like to observe and analyze.

6.1.2 Matrix structures we will use

A *Toeplitz* matrix is a matrix of the form

$$T = [a_{j-i}]_{i,j=1}^{m} = \begin{bmatrix} a_0 & a_1 & a_2 & & \cdots & a_{m-1} \\ a_{-1} & a_0 & a_1 & \ddots & & a_{m-2} \\ a_{-2} & a_{-1} & a_0 & \ddots & \ddots & a_{m-3} \\ & a_{-2} & a_{-1} & \ddots & \ddots & a_{m-4} \\ & & \ddots & \ddots & \ddots & \\ \vdots & & \ddots & \ddots & \ddots & \\ & & & a_{-1} & a_0 & a_1 \\ a_{-m+1} & a_{-m+2} & & & a_{-1} & a_0 \end{bmatrix}.$$

If $a_{b+1} = a_{b+2} = \ldots = a_{m-1} = 0 = a_{-m+1} = \ldots = a_{-b-1}$ then T is called *band-Toeplitz* and b is called *bandwidth.*

6.1.3 Data analysis problem defined

Making an observation with small enough statistical and systematic error bars to resolve the detailed shape of the angular power spectrum of the CMB requires a very sensitive detector scanning a significant fraction of the sky at very high resolution. Reducing tens of millions of individual observations to an angular power spectrum of a thousand multipoles is a computationally challenging task. Each observation contains detector noise as well as signal on the sky; the signal on the sky includes not only the CMB but also foreground sources such as interstellar dust; and both the noise and the signal components of nearby observations are correlated. Current algorithms that estimate the power spectrum scale as the number of map pixels cubed in floating point operation count. Current balloon observation are generating maps with hundreds of thousands of pixels and the PLANCK satellite to be launched in 2007 will increase this number to tens of millions at which time the current algorithms will no longer be useful even on supercomputing platforms.

On the bright side the matrices that are involved in the power spectrum estimation are highly structured giving hope that faster structure exploiting algorithms may be constructed for the analysis of the upcoming data.

6.1.4 The data we are given

We are given

- A vector t of m temperature observations made at equal time intervals that we will call *time samples*;
- An $m \times n$ pointing matrix A. At any given time we are only observing only one pixel and the matrix A tells us which pixel this is[1]:

$$A_{ij} = \begin{cases} 1 & \text{if pixel } j \text{ was observed at time } i \\ 0 & \text{otherwise.} \end{cases} \tag{6.1}$$

Therefore if $A_{ij} = 1$ then pixel j was observed at time i and its temperature is t_i.

[1]Datasets may include information about polarization yielding pointing matrices with more complicated structures. Designing fast algorithms for those matrix structures is still an open problem and a topic of current research.

- $n \times n$ signal correlation matrices $S_1, S_2, \ldots, S_k$. These matrices are diagonal in spherical harmonics space (see [59] and section 6.3) and we will use fast spherical harmonics transforms later on to form matrix vector products with these matrices.

- Let T^{-1} be the $m \times m$ matrix of the correlations between the noise in the different time samples. We are given T and know that it is band-Toeplitz with bandwidth b. Both the band and the bandwidth are determined by the properties of the measuring equipment and are independent of the duration of the experiment m. This property has important consequences in the analysis of the convergence of the algorithms later on.

6.1.5 Quantities we would like to compute

- The maximum likelihood pixel map d underlying the observations t. We call this "map making". It is straightforward [14]:

$$d = \left(A^T T A\right)^{-1} A^T T t. \tag{6.2}$$

- The multipole powers $C = (C_1, C_2, \ldots, C_k)$ (the power spectrum) of the CMB, where each C_i is a scalar. They are estimated as follows [14]. Let

$$S = \sum_{l=1}^{k} C_l S_l, \quad N = \left(A^T T A\right)^{-1} \quad \text{and} \quad D = N + S. \tag{6.3}$$

Then C is the most likely CMB power spectrum underlying the original observation t, i.e. the maximizer of the following log likelihood function [14]:

$$\mathcal{L}(C) = -\frac{1}{2}\left(d^T D^{-1} d + \mathrm{Tr}(\ln D)\right) \tag{6.4}$$

To find the maximizer of $\mathcal{L}(C)$ we use Newton-Raphson iteration to search the k dimensional parameter space for a zero of the derivative of the log likelihood function. We calculate the first two derivatives of the log likelihood function with respect to the C_i's.

$$\frac{\partial \mathcal{L}}{\partial C_i} = \frac{1}{2}\left(d^T D^{-1} S_i D^{-1} d - \mathrm{Tr}\left[D^{-1} S_i\right]\right) \tag{6.5}$$

$$\frac{\partial^2 \mathcal{L}}{\partial C_i \partial C_j} = -d^T D^{-1} S_i D^{-1} S_j D^{-1} d + \frac{1}{2}\mathrm{Tr}\left[D^{-1} S_i D^{-1} S_j\right]. \tag{6.6}$$

Then starting with an initial guess C_0 we iterate

$$C_{i+1} = C_i + \delta C_i,$$

where

$$\delta C_i = -\left(\left[\frac{\partial^2 \mathcal{L}}{\partial C^2}\right]^{-1} \frac{\partial \mathcal{L}}{\partial C}\right)_{C=C_i}$$

until $\delta C_i \sim 0$ to the desired accuracy.

Dimensions of the data

Here are the values of the dimensions of the data for the current datasets and the expected dimensions of the the PLANCK satellite data to be received in 2007.

Table 6.1: Dimensions of the data

Dimension	Current datasets	PLANCK
m	10^8	10^{10}
n	10^6	10^8
k	10	3000
b	10^5	10^5

Relationship among the dimensions

To correctly study and evaluate the cost of algorithms we need to be aware of how the different dimensions are related. The number of multipole powers k will not grow past 3000 because of the finite beam size in the observation equipment. The bandwidth b of the band-Toeplitz matrix T depends on the properties of the observation equipment only and since the plans are to use the same equipment in the future b will likely stay pretty much the same independent of the size of the experiment (n). In each experment each pixel will likely be observed not more than 1000 times, therefore

$$m \leq 1000n. \tag{6.7}$$

The number of pixels n is the most important parameter in each experiment.

6.1.6 Cost of current algorithms

The current algorithms [14] use structure-ignoring $O(n^3)$ algorithms for manipulating matrices of size n (solving, matrix-matrix multiplication, inversion, etc.) and cost [14]:

$$O(k^2 n^3).$$

Thus they will likely be inapplicable to the larger PLANCK size datasets in view of the magnitude of the size of the matrices n we expect.

6.1.7 New Results

We design new parallel algorithms for map making and power spectrum estimation that exploit the structure of the matrices involved the the computations. The map making algorithm (computing d in (6.2)) takes only

$$621 m \log b \leq 10^7 n$$

arithmetic operations, where the constant 10^7 applies to both datasets in Table 6.1.5. The power spectrum estimation (estimating the maximizer of (6.4)) costs only

$$O\left(k^2 n^{3/2}\right).$$

Both algorithms scale almost linearly with the number of processors $\mathcal{P}$ on a parallel machine.

We also prove important theoretical results, namely that the algorithms we have designed will succeed on PLANCK size datasets despite concerns about scaling and convergence. For the map making algorithm

- We prove that the preconditioned conjugate gradients (PCG) applied to solve $N^{-1}c = x$ for the purpose of computing the matrix vector product Nx, will always converge in about the same number of steps independent of the size of the experiment, i.e. independent of the number of observations m and the number of observed pixels n. (N is defined in (6.3)).
- We identify the best choice of diagonal preconditioner for PCG.

For the map making and power spectrum estimation algorithms.

- We prove that one should not worry about rounding errors contaminating the results because the problems are very well-conditioned (condition number ≈ 100) and the effect of rounding errors is much smaller than the effect of the uncertainty of the original data.

- For the purposes of minimizing condition numbers and speeding up PCG we recommend that the design of the upcoming PLANCK satellite experiment be such that all pixels are scanned about the same number of times.

Exploitable Matrix structure

There is a wealth of mathematical structure that can be used to speed up computations, namely

- T is band-Toeplitz, thus fast algorithms may be used for matrix-vector multiplication [42] in time $O(m \log b)$.

- A has only one nonzero element per row.

- S and S_i are diagonal in spherical harmonics space. Under the reasonable assumption that the pixels in our maps are situated along rows of the same declination and that the number of rows in our maps is about equal to the square root of the total number of pixels n, i.e. the pixels form a large subset of a roughly square "screen", we can perform fast forward and backward spherical harmonic transformations in $O(n^{3/2})$ time [33, 15]. Thus we can form a matrix vector product with each of the matrices S and S_i in $O(n^{3/2})$ time.

- The matrices N, T, S, S_i and D are *positive definite* correlation matrices, thus preconditioned conjugate gradient algorithms may be used for solving linear systems involving those matrices.

6.2 Fast Algorithm for Computing Nx and Map Making.

We wish to compute $z = Nx$ but we are given $N^{-1} = A^T T A$ explicitly. So we present an $O(m \log b)$ preconditioned conjugate gradients algorithm for computing z that uses our ability to multiply quickly by N^{-1}. Then we apply this algorithm to map making.

6.2.1 Band-Toeplitz matrix vector product

For the symmetric band-Toeplitz matrix $T = [a_{|j-i|}]_{i=1}^{m}$ with bandwidth b ($a_i = 0$ for $i > b$) we will now show how to form a band-Toeplitz matrix vector product Tx in $O(m \log b)$ time. In our applications b will be about 10^5, whereas n will be the number of pixels that will range up to 10^8.

An $p \times p$ *circulant* matrix R is a Toeplitz matrix such that

$$R = [r_{j-i}]_{i,j=1}^{p}$$

and $r_i = r_{i-p}$. The circulant matrices have the attractive property that they are diagonalized by the Discrete Fourier Transform (DFT) $R = F^T \cdot G \cdot F$, where F is the matrix of the DFT and G is diagonal (with the spectrum of R on the diagonal) [42]. Thus a matrix-vector product with a circulant matrix can be computed in $O(p \log p)$ time. An $m \times m$ Toeplitz matrix T can be multiplied by a vector in $O(m \log m)$ time [42] by embedding the matrix T into a $(2m-1) \times (2m-1)$ circulant matrix R with first row $(a_0, a_1, \ldots, a_{m-1}, a_{-m+1}, a_{-m+1}, \ldots, a_{-1})$. Then computing

$$R \cdot \begin{bmatrix} x \\ 0 \end{bmatrix} = \begin{bmatrix} Tx \\ * \end{bmatrix},$$

where the symbol 0 stands for a zero vector of size $m-1$ and the symbol "*" for a vector of size $m-1$ that we discard. A band-Toeplitz matrix-vector product can be formed in $O(m \log b)$ time instead of $O(m \log m)$, where b is the bandwidth of the Toeplitz matrix with the use of Algorithm 6.2.1 below.

The Fourier transform of a vector of size p may be computed in $5p \log p$ time and there are optimized FFT programs in the public domain. Our choice was FFTW [36]. This particular FFT implementation, which also implements the inverse FFT (multiplication by F^T) had the following features, some of which are shared with other FFT implementations:

- FFTs of size that is a power of 2 run fastest.
- FFTW when called to apply FFT on consecutive vectors runs faster than multiple calls to FFTW to apply FFT to a single vector.

We will design our algorithms to exploit both features.

Algorithm 6.2.1 (Band-Toeplitz matrix-vector product) *Let T be an $m \times m$ band-Toeplitz matrix with band $a = (a_1, a_2, \ldots, a_b)$ of and let x be an m-vector. The following algorithm computes the product Tx in $O(m \log b)$ time.*

```
10   function x =BT(a, x, m)
20       Select FFT size p = 2^⌊log_2 7b⌋
30       g = [a,zeros(p − 2b + 1), a(n − 1 : −1 : 0)]
40       G = FFT(g)
50       q = ⌈(n+2b)/(p−2b)⌉
60       y =zeros(pq, 1)
70       z = [zeros(b, 1), x, zeros(b, 1)]
80       for i = 0 : q − 1
90           y(ip : (i + 1)p − 1) = z(i(p − 2k) : i(p − 2k) + p − 1)
100      Apply q size p FFTs to y, (which is faster, but equivalent to
         "for j=0:q-1 do y(iq:(i+1)q-1)=FFT(y(iq:(i+1)q-1))")
110      for i = 0 : q − 1
120          y(ip : (i + 1)p − 1) = diag(G) · y(ip : (i + 1)p − 1)
130      Apply q size p Inverse FFTs to z
140      x = z(b + 1 : n + b)
150      for i = 0 : q − 1
160          x(i(p − 2b) : i(p − 2b) + q − 1) = x(iq : (i + 1)q − 1)
170      return x
```

Note: The selection of the size of the FFT p is optimized at run time. It is clear that p should not be too small ($p - 2b$ entries of x are effectively transformed by a single FFT) and larger size FFTs run slower than smaller ones. Our experience has been that the above setup is typically optimal.

Here is how Algorithm 6.2.1 works. On line 20 we select a size $p \approx 7b$ for a circulant matrix C in which a $6b \times 6b$ band-Toeplitz matrix T_1 with band a will be embedded. The vector b is the first row of R (line 30). On line 40 the vector $G = FFT(g)$ is the vector with the spectrum of R and we have

$$R = F^T \cdot \mathrm{diag}(G) \cdot F, \tag{6.8}$$

where F is the matrix of the discrete Fourier transform. It is clear now that a matrix vector product $R \cdot x$ can be formed in $O(p \log p)$ time. Later we will divide a vector of length $n+2b$ into pieces of size $p-2b$ each and in line 50 we see how many pieces we will get. On line 60 we initialize a zero vector of size pq. Before we explain line 70 we need to explain that multiple multiplications by R will be equivalent to a multiplication by T.

If $r = Tx$ then

$$r_i = \sum_{j=-b}^{b} a_j x_{i+j}.$$

Also if $w = (x_{i-b}, x_{i-b+1}, \ldots, x_{i-b+p-1})$ and $v = Cw$ we will also have

$$v_i = \sum_{j=-b}^{b} a_j x_{i+j}.$$

In fact we will also have $v_j = r_j$ for $j = i, i+1, \ldots, i+p-b-1$. Therefore we can compute $r_i, \ldots, r_{i+p-b-1}$ by multiplying by C instead of T, computing $p-2b$ entries at a time.

The repeated multiplication by C is equivalent to multiplying x by an infinite Toeplitz matrix and this is fine for all but the top k and the bottom k entries of Tx. In order to "create the corners" of T we pad the vector x with b zeros in front and back.

In order to compute the first $p-2b$ entries of Tx using this process, we need to form $C \cdot z(1:p)$ for the following $p-2b$ entries we need to form $C \cdot z(p-2b+1 : p-2b+p)$, which is problematic since the vectors $z(1:p)$ and $z(p-2b+1 : p-2b+p)$ overlap. To overcome this difficulty we "spread out" the vector p (lines 80-90) so that each FFT will be applied to a nonoverlapping portion of a new vector y and we will perform q FFTs on contiguous parts of y, each containing p elements.

Lines 100-130 implement the formula (6.8) on each of the q pieces of y each piece of length p. On line 140 we discard the k entries in front and back that were introduced in line 70 to "create the corners" of T. Finally we collect the middle sections of each of the q pieces of x, each of length p.

Proposition 6.2.1 (Cost of Algorithm 6.2.1) *The cost of Algorithm 6.2.1 in arithmetic operations is bounded by* $20n \log b$.

Proof: Algorithm 6.2.1 performs $q \leq \frac{m}{4b}$ forward and backward FFTs each of size $p \leq 7b$ in addition to a scaling of a vector of size pq. Each size p FFT costs $5p \log p$ [36]. The total cost is thus bounded by

$$\frac{m}{4b} 2 \cdot (5 \cdot 7b \cdot \log(7b)) + \frac{m}{4b} 7b \leq 24m \log b.$$

□

6.2.2 The matrix vector product $N^{-1}x$

We have $N^{-1} = A^T T A$ where A is a pointing matrix, defined in (6.1). Since A only has one nonzero per row a multiplication by A (or A^T) is straightforward is takes only m time. Therefore a matrix-vector product $N^{-1}x$ can be formed in $26m \log b$ time.

6.2.3 The matrix vector product Nx

Since N is positive definite we can form the product $z = Nx$ by solving $N^{-1}z = x$ using applying preconditioned conjugate gradients. We know how to multiply by N^{-1} fast, so we only need to discuss preconditioning.

6.2.4 Preconditioning

The numerical properties of T

The entries of the band-Toeplitz matrix T represent the correlations in the observed data in time and those correlations are intrinsic properties of the instruments. In other words the band of N^{-1} or its bandwidth are independent of the number of time samples (observations) m. Therefore for larger experiments we will simply have a larger band-Toeplitz matrix with the same band and bandwidth. The condition number of a band-Toeplitz matrix remains nearly the same as m changes and the band and bandwidth remain the same [62]. For the current experiments the condition number of T is approximately 10^2 and based on the above arguments we can expect that the T matrices from future experiments will likely have very similar condition numbers.

The numerical properties of A

Since A has only one nonzero per row we can conclude that $A^T A = F$, where F is diagonal

$$F = \mathrm{diag}(h_1, h_2, \ldots, h_n) \tag{6.9}$$

and h_i equals the number of times the i-th pixel is observed.

Therefore $A_1 = A \cdot F^{-1/2}$ is orthogonal and

$$\mathrm{cond}(A_1^T \cdot T A_1) \leq \mathrm{cond}(T) \approx 10^2, \tag{6.10}$$

where the condition number of a matrix V is defined as $\text{cond}(V) = ||V|| \cdot ||V^{-1}||$. We have inequality above because A_1 is not square. Because of (6.10) we conclude that F is an appropriate preconditioner, which will guarantee that the condition number of the preconditioned matrix N^{-1} will be about 10^2 *independent* of the number of times samples m and the number of pixels observed n.

The preconditioned conjugate gradients algorithm applied to today's datasets converge in about 30 steps for relative residual 10^{-4}. Based on the analysis above we can conclude that the preconditioned conjugate gradients algorithm will likely converge in a similar number of steps on the larger datasets that will be available in the future.

Algorithm 6.2.2 (Computing Nx) *The following algorithm computes the matrix-vector product $z = Nx$ by using conjugate gradients to solve $N^{-1}z = x$.*

```
function z = Nx(x)
  F = A^T A                 ...(diagonal, only takes m time)
  Call pcg, passing x and Nix as a matrix-vector multiplication function

function z = Nix(x)         ... computes z = N^-1 x
  z = Ax
  y = BT(a, z)  ... using Algorithm 6.2.1
  z = A^T y
```

Cost of Algorithm 6.2.2

PCG will likely converge in about 30 steps to tolerance 10^{-4}, which is adequate for our purposes, and each step costs $26m \log b$. Because of (6.7) and the fact that b is about 10000 (Table 6.1.5) we conclude that forming the product Nx using Algorithm 6.2.2 will cost not more than

$$10^7 n.$$

For PLANCK size datasets this number is about 10^{15} or one petaflop, a large but feasible computation.

6.2.5 Map making

The map of the CMB is our first major goal and we now present an algorithm for its computation. The map is computed according to the formula [14]

$$d = (A^T T A)^{-1} A^T T t = N \cdot (A^T T t)$$

Algorithm 6.2.3 (Map making) *The following algorithm makes a map of the CMB*

10 `function` $d = \mathtt{map}(t)$

20 $\quad y = Tt$ `... using Algorithm` *6.2.1*

30 $\quad z = A^T y$

40 $\quad d = Nz$ `... using Algorithm` *6.2.2*

Line 30 is straightforward "observing" of the sky and takes m time.

Cost of Algorithm 6.2.3

The total cost of Algorithm 6.2.3 is less than

$$1.1 \cdot 10^7 n. \tag{6.11}$$

6.2.6 Parallelism

Algorithm 6.2.1

Algorithm 6.2.1 is embarrassingly parallel, since the q block-circulant matrix-vector products are independent and can be performed simultaneously. The initialization phase (lines 20-50) need only be performed once and is a very minor amount of work (one FFT of size approximately $2^{16} = 65536$). Thus the parallel cost of algorithm 6.2.1 applied to the matrix T is

$$(26 m \log b) / \mathcal{P},$$

i.e. it scales perfectly with the number of processors $\mathcal{P}$, and our practical implementations have confirmed that assertion (see section 6.5).

Algorithm 6.2.2

In the process of forming the product $N^{-1}x = A^T T A x$, forming the product $z = Ax$ is embarrassingly parallel as so is $y = Tz$ as we just discussed. Forming $A^T z$ requires one `MPI_AllReduce` operation [67] on a vector of size n and therefore costs $2n \log \mathcal{P}$ for communication. Therefore the parallel cost of Algorithm 6.2.2 will not exceed

$$10^7 n/\mathcal{P} \quad \text{arithmetic and} \quad 60n \log \mathcal{P} \quad \text{for communication.}$$

It is now clear that Algorithm 6.2.2 will scale almost linearly with $\mathcal{P}$. This expectation has been confirmed in our numerical experiments.

6.3 Solving $Dx = c$

Since $D = S + N$ is a symmetric positive definite matrix we can use preconditioned conjugate gradients algorithm to solve $Dx = c$. To do so we need a fast algorithm to form $Dx = Sx + Nx$.

As mentioned in the introduction the matrices S and S_i are diagonal in spherical harmonics space and we have fast parallel forward and inverse spherical harmonics transformations [33, 15] that allow us to form the matrix-vector products Sx and $S_i x$ in

$$6n^{3/2} \tag{6.12}$$

floating point operations. We also make the assumption that the pixels we observe will be located along rows of the same declination and that the number of rows in a given map will be about $\sqrt{n}$. This assumption is justified by previous experiments and the ideas about the design of the PLANCK experiment.

We can form Nx in $10^7 n$ time as explained in section 6.2.

Both the algorithm for forming Nx and the algorithm for forming Sx scale almost linearly with the number of processors $\mathcal{P}$ on a parallel machine, therefore the matrix-vector product Dx can also be formed in parallel and will also scale nearly linearly with $\mathcal{P}$.

Algorithm 6.3.1 (Solving $Dx = c$) *Given x the following algorithm computes $x = D^{-1}c$ by using conjugate gradients.*

```
function x = SolveD(c)
  Use conjugate gradients to solve Dx = c,
     where the multiplication by D is done as follows

function y = MultD(z)
```

$v = Nx$ ... `using Algorithm` *6.2.2*

$w = Sx$ ... `using spherical harmonic transforms`

$y = v + w$

Cost of Algorithm 6.3.1

The cost of using preconditioned conjugate gradients to solve $Dx = c$ equals the sum of the costs of forming Sx and Mx times the number of iteration steps for convergence.

We now argue that the number of iteration steps should be independent of the size of the problem – number of pixels n or number of time samples m.

The number of iterations that pcg takes to converge is directly related to the condition number of the matrix we are trying to solve with.

The condition number of N can bounded by

$$\begin{aligned} \text{cond}(N) &= \text{cond}(A^T T A) \\ &= \text{cond}(F^{-1/2} A_1^T T A_1 F^{-1/2}) \\ &\leq \text{cond}(A_1^T T A_1) \cdot \text{cond}(F) \\ &\leq 100 \cdot h_{\max}/h_{\min} \\ &\leq 10^5, \end{aligned}$$

where $F = \text{diag}(h_1, h_2, \ldots, h_n)$ is the matrix (6.9) and h_i equals the number of times pixel i is observed. Since every pixel is observed at least once and at most about 1000, we have $\text{cond}(N) \leq 10^5$. It has also been our practical experience that $10^4 \leq \text{cond}(N) \leq 10^5$.

The condition number of S is independent of the size of the problem. The spectrum of S is the spectrum of the CMB and is therefore a fixed quantity.

Since $\text{cond}(N)$ and $\text{cond}(S)$ are bounded, N and S are positive definite and $D = S + N$ we can claim that $\text{cond}(D) \leq \text{cond}(N) + \text{cond}(S)$ is bounded independent of the size of the matrices. Therefore we can expect that the convergence of the conjugate gradients (CG) algorithm when applied to D will be independent of the size of the problem. It has been our experience that $\text{cond}(D) \leq 10^5$. It has also usually taken not more than 100 steps for CG to compute $D^{-1}x$ with relative residual less than 10^{-4} and given the above we can safely assume that similar convergence rates will apply for larger datasets.

Therefore we can conclude that the cost of Algorithm 6.3.1 is bounded by

$$10^{10} n + 600 n^{3/2}. \tag{6.13}$$

For PLANCK size datasets this is bounded by 10^{18} or one exaflop.

6.3.1 Parallelism

Since the matrix-vector produces Dx and Nx can be formed in parallel and scale nearly linearly with $\mathcal{P}$, Algorithm 6.3.1 will also scale nearly linearly. This has also been our practical experience.

6.4 Estimating the Power Spectrum of the CMB

This phase is much more computationally intensive than the map making in section 6.2. The hard part is estimating the traces appearing in (6.5) and (6.6). We start with a Monte-Carlo algorithm for estimating the trace of a matrix that only requires a matrix-vector multiplication.

6.4.1 Trace estimator

Our Monte Carlo approach is based on the following proposition [5, 45, 27]:

Proposition 6.4.1 *Let H be an $m \times m$ square symmetric positive definite matrix and V be a discrete random variable which takes the values ± 1 with equal probability and let z be a vector of m independent samples from V. Then $z^T H z$ is an unbiased estimator of* $\mathrm{Tr}(H)$, *i.e.*

$$E(z^T H z) = \mathrm{Tr}(H).$$

Therefore in order to estimate the trace of a matrix we only need to be able to multiply by that matrix fast. We will see that this will be the situation in our case.

6.4.2 Computing the derivatives of the log likelihood function $\mathcal{L}$

We start with two algorithms for estimating the traces that appear in the first (6.5) and the second (6.6) derivatives.

Algorithm 6.4.1 (First derivatives $\frac{\partial \mathcal{L}}{\partial C_i}$) *The following algorithm estimates $\frac{\partial \mathcal{L}}{\partial C_i}$ in (6.5)*

```
function f=dLdC(i)
```

$x = D^{-1}d$ `... using Algorithm` *6.3.1*

$w = S_i x$ `... using spherical harmonic transforms`

$f = x^T w$

$g = \mathrm{Tr}\left(D^{-1}S_i\right)$ `estimated using Proposition` *6.4.1*

  `The computation` $g_i = z^T \cdot \left(D^{-1}S_i\right) \cdot z$ `is done as`

  $w = S_i z$ `... using spherical harmonic transforms`

  $x = D^{-1}w$ `... using Algorithm` *6.3.1*

  $g_i = z^T x$

$f = (f - g)/2$

Algorithm 6.4.2 (Second derivatives $\frac{\partial^2 \mathcal{L}}{\partial C_i \partial C_j}$) *The quantities* $\frac{\partial^2 \mathcal{L}}{\partial C_i \partial C_j}$ *in (6.6) are estimated using the following algorithm.*

`function f=d2LdCbb(i,j)`

$x = D^{-1}d$ `... using Algorithm` *6.3.1*

$w = S_j x$ `... using spherical harmonic transforms`

$y = D^{-1}w$ `... using Algorithm` *6.3.1*

$w = S_i y$ `... using spherical harmonic transforms`

$f = -w^T x$

$g = \mathrm{Tr}\left(D^{-1}S_i D^{-1}S_j\right)$ `estimated using Proposition` *6.4.1*

  `The computation` $g_i = z^T \cdot \left(D^{-1}S_i D^{-1}S_j\right) \cdot z$ `is done as`

  $w = S_j z$ `... using spherical harmonic transforms`

  $x = D^{-1}w$ `... using Algorithm` *6.3.1*

  $w = S_i x$ `... using spherical harmonic transforms`

  $x = D^{-1}w$ `... using Algorithm` *6.3.1*

  $g_i = z^T x$

$f = f + g/2$

6.4.3 Power spectrum estimation algorithm

Here is the power spectrum estimation algorithm using Newton-Raphson iteration [14].

Algorithm 6.4.3 (Power spectrum estimation) *The following algorithm estimates the power spectrum of the CMB.*

```
Start with an initial guess C_0
for b = 1 : k
   Compute the first derivatives ∂L/∂C_b   ... using Algorithm 6.4.1
for b = 1 : k
   for j = 1 : k
      Compute the second derivatives ∂²L/(∂C_b ∂C_j)   ... using Algorithm 6.4.2
Compute an update δC_i = − ( [∂²L/∂C²]^{-1} ∂L/∂C )|_{C=C_i}
Iterate until δC_i ~ 0 to the desired accuracy
```

Cost of Algorithm 6.4.3

We assume that the trace estimators in both Algorithms 6.4.1 and 6.4.2 can estimate the trace in a fixed number of steps every time (which has certainly been our experience). It has also usually taken about 10 iterations for Algorithm 6.4.3 to reach a maximum of the log likelihood function $\mathcal{L}$.

The only algorithms that are used, other than dot products, are Algorithm 6.3.1 and multiplication by S_i. We must invert a matrix of size k in order to compute the update δ_i, but because of the relative size of k in Table 6.1.5 we ignore this cost in the analysis below. Since the first derivative must be computed for every $i = 1, 2, \ldots, k$ and the second derivative must be computed for every pair (i, j) the total cost of Algorithm 6.4.3 is

$$O\left(k^2 n^{3/2}\right).$$

To get an idea about the magnitude of the cost of Algorithm 6.4.3 for PLANCK size datasets we need to make certain assumptions based on our practical experience. We assume that the trace estimator (Proposition 6.4.1) will produce reasonable estimates in about 100 iterations. We also assume that the Newton-Raphson iteration will converge in less than 10 steps. Under those assumptions the cost of Algorithm 6.4.3 for PLANCK size datasets is bounded by

$$k^2 \cdot \left(2 \cdot 10^{13} n + 1.2 \cdot 10^5 n^{3/2}\right).$$

6.4.4 Parallelism

The parallelism in the algorithm for estimating the power spectrum of the CMB is inherited from the parallelism in each component of the algorithm (forming $D^{-1}x$ and multiplication by S_i). We can therefore expect nearly linear scaling with the number of processors. This expectation has been met in our current tests on model data.

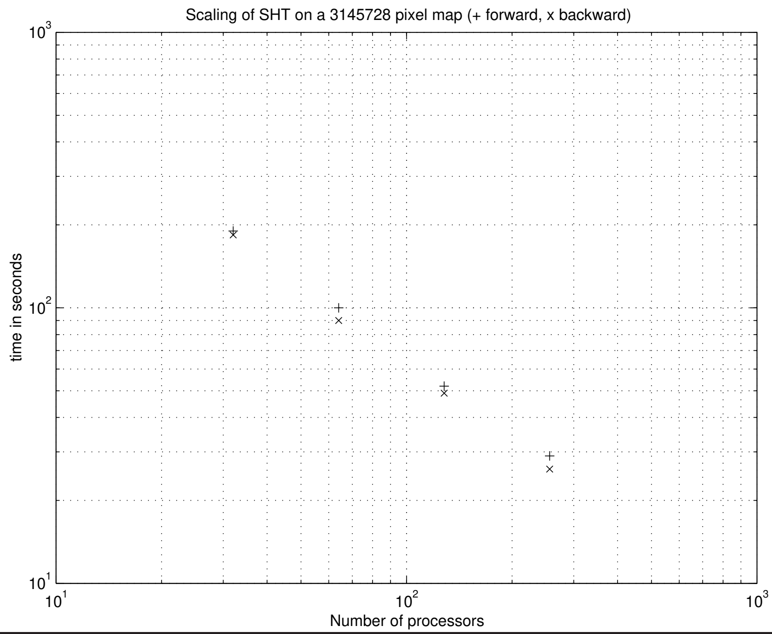
6.5 Numerical Experiments

We have tested our algorithms to verify convergence and parallelism on small test data. We performed our tests on the parallel supercomputers at Lawrence Berkeley National Laboratory. The spherical harmonics transforms were tested on the IBM SP cluster and Algorithm 6.2.2 was tested on the Cray T3E.

We performed the following tests:

1. We tested the parallel scaling of the algorithms for matrix-vector multiplication for the matrices S_l. Figure 6.1 confirms our predictions for near linear scaling [15].

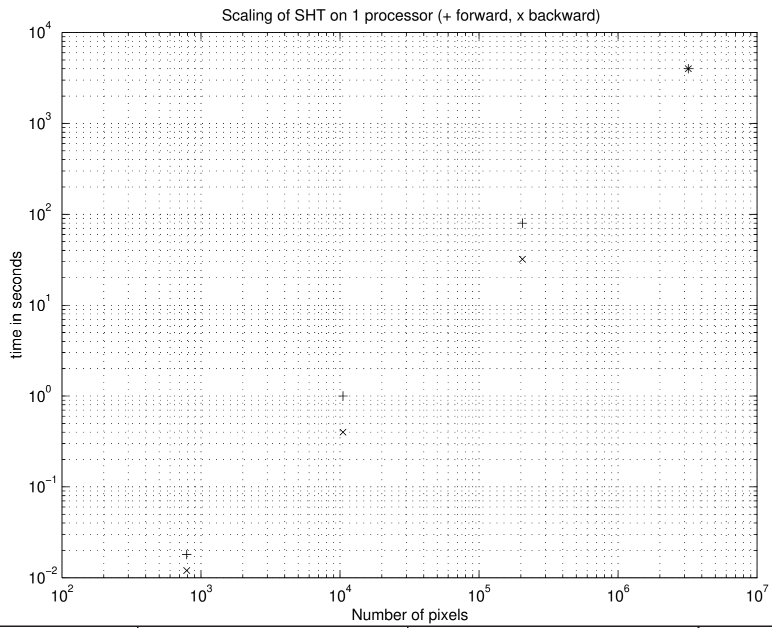
Figure 6.1: Parallel speedup of the fast spherical harmonic transforms



Number of processors $\mathcal{P}$	Forward transform (seconds)	Backward transform (seconds)
32	190	184
64	100	90
128	52	49
256	29	26

2. We also tested formula (6.12) by running the spherical harmonic transform algorithms on several maps with different number of pixels n. Our numerical experiments shown in Figure 6.2 confirm that the spherical harmonic algorithms do scale as $O(n^{3/2})$ [15].

Figure 6.2: Scaling of the fast spherical harmonic transforms with the size n of the maps



Number of pixels n	Forward transform (seconds)	Backward transform (seconds)	$O(n^{3/2})$ (normalized)
$7.9 \cdot 10^{2}$	$1.2 \cdot 10^{-2}$	$1.8 \cdot 10^{-2}$	$1.8 \cdot 10^{-2}$
$1.0 \cdot 10^{4}$	$4.0 \cdot 10^{-1}$	$1.0 \cdot 10^{+0}$	$8.7 \cdot 10^{-1}$
$2.0 \cdot 10^{5}$	$3.2 \cdot 10^{+1}$	$8.0 \cdot 10^{+1}$	$7.4 \cdot 10^{+1}$
$3.0 \cdot 10^{6}$	$4.0 \cdot 10^{+3}$	$4.0 \cdot 10^{+3}$	$4.6 \cdot 10^{+3}$

3. We tested the convergence of Algorithm 6.2.2 to confirm that it converges in about 30 steps to relative residual 10^{-4} and present our numerical results in Figure 6.3.

4. We made a map using algorithm 6.2.3 and compared it with the map produced by the traditional algorithm in Figure 6.4.

5. Finally, we tested the parallel performance of Algorithm 6.2.2 as we ran it on different number of processors. Figure 6.5 confirms our predictions for near linear speedup with the number of processors.

Figure 6.3: Convergence of Algorithm 6.2.2

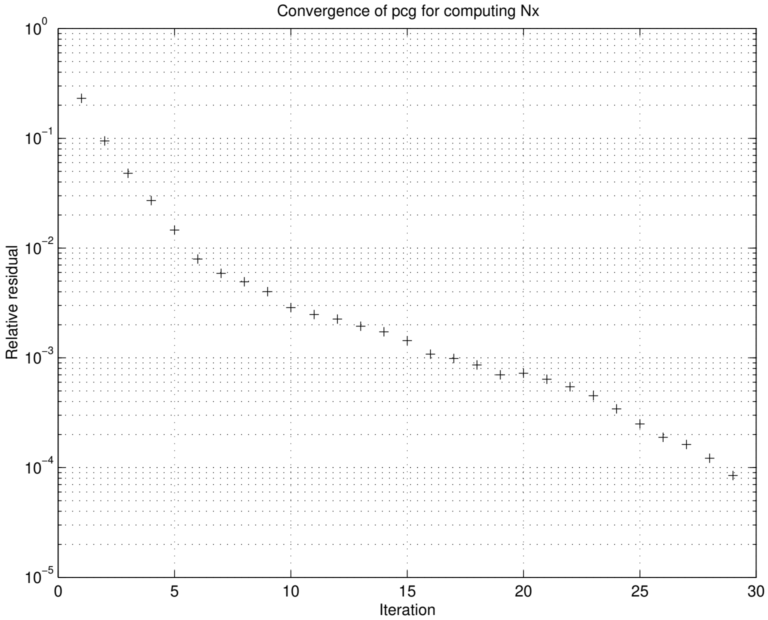
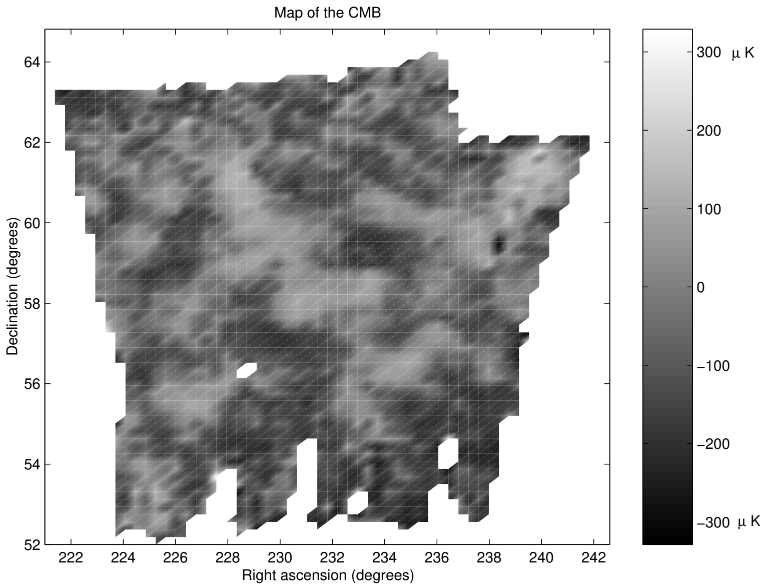
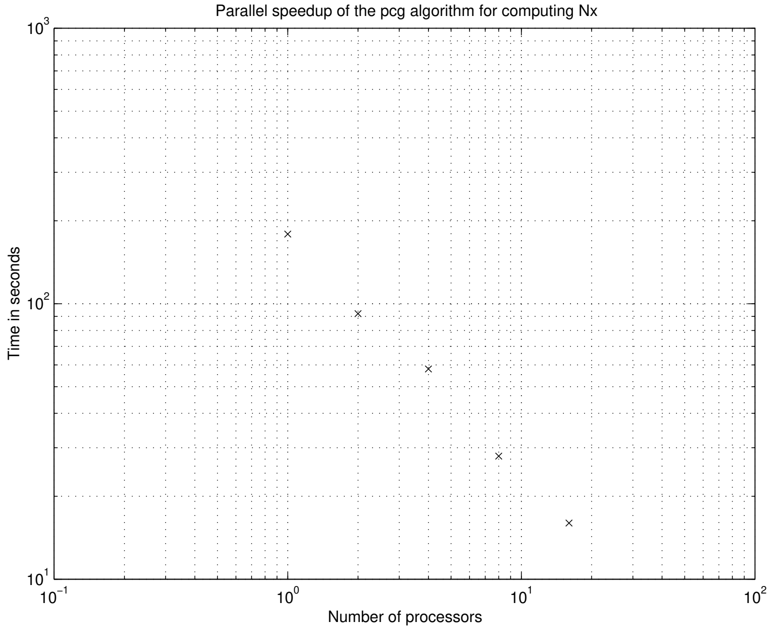


Figure 6.4: An image of the Cosmic Microwave Background Radiation



A map (d_1) of the CMB as computed by Algorithm 6.2.3 on a simulated dataset. It is visually indistinguishable from the map (d_2) computed by the traditional algorithm [14] because $\frac{||d_1-d_2||}{||d_2||} = 1.6 \cdot 10^{-3}$.

Figure 6.5: Parallel speedup of Algorithm 6.2.2



Number of processors $\mathcal{P}$	1	2	4	8	16
Time in seconds	179	92	58	28	16

6.6 Conclusions

We have developed fast structure exploiting parallel algorithms for map making and power spectrum estimation of the CMB. Our algorithms scale nearly linearly on multiple processors and are about two orders of magnitude faster than the current structure-ignoring algorithms. Even though our new algorithms perform similarly to the traditional algorithms on the current small test datasets, the full potential of the new algorithms will not be unleashed until large datasets become available in 2007. On these datasets the constants for the costs hidden in the big-O notation, which are independent of the size of the datasets can be successfully amortized.

Chapter 7

Computational Approach to Polynomial Identities in Matrix Algebras

We present an efficient parallel computer method for calculation with polynomial identities of matrices over a field K of characteristic 0. Using this method we partially prove a conjecture (in the cases $n = 4$ and $n = 5$) that all polynomial identities of degree $2n + 2$ in the matrix algebra of $n \times n$ matrices over K are are consequences of the standard identity s_{2n}.

7.1 Introduction

The polynomial identities of the algebra $M_n(K)$ of $n \times n$ matrices over a field K of characteristic 0 are among the most intriguing topics in the theory of algebras with polynomial identities. Since 1950, when Amitsur and Levitzki proved their famous theorem that $M_n(K)$ satisfies the so called *standard identity* of degree $2n$, many results have been established. Nevertheless, the picture is far from a satisfactory form and only the structure of the polynomial identities of the 2×2 matrices is well known. There are many open problems and conjectures concerning the bases of polynomial identities of $M_n(K)$, and one of the most interesting questions is that of finding the minimal degree of the polynomial identities which do not follow from the standard identity.

Some of the facts in the theory have been obtained, or at least conjectured, as a result of extensive calculations by hand or computer. We develop further the computational methods of Drensky [30, 32] for calculations with the polynomial identities of matrices and obtain the following new results:

- We present a new divide-and-conquer method for computing the complete skew-symmetric polynomial

$$s_m(x_1, x_2, \ldots, x_m) = \sum_{\sigma \in S_m} \mathrm{sign}(\sigma) x_{\sigma(1)} x_{\sigma(2)} \ldots x_{\sigma(m)}, \tag{7.1}$$

  where S_m is the symmetric group of order m, i.e. the group of all permutations σ on m objects, and each x_i is an $n \times n$ matrix of numerical values. Our method is exponentially faster than a straightforward evaluation of the explicit polynomial, which was the method used in prior work [30].

- Using this algorithm we were able to perform extensive computations and to prove the following result:

  All polynomial identities in $M_n(K)$ of degree $2n+2$ are consequences of the standard identity of degree $2n$ for $n = 4, 5$.

7.2 Survey

Let K be a fixed field of characteristic 0. We denote by $A = K\langle X \rangle$ the free associative algebra with a countable set of free generators $X = \{x_1, x_2, \ldots\}$. An element $f(x_1, \ldots, x_m) \in A$ is called a *polynomial identity* for a K-algebra R if $f(r_1, \ldots, r_m) = 0$ for any $r_1, \ldots, r_m \in R$. The set of all polynomial identities of R, denoted by $T(R)$, is a *T-ideal*, i.e. a two-sided ideal of A which is invariant under all endomorphisms of A. When the T-ideal $T(R)$ of a K-algebra R is nontrivial ($T(R) \neq (0)$), then R is called a *PI-algebra*. Any set of generators of $T(R)$ as a T-ideal is called a *basis of the polynomial identities of* R.

The T-ideal of the polynomial identities of the matrix algebra $M_n(K)$ is one of the most interesting objects in the theory of PI-algebras and its applications [61, 46, 64, 35, 29].

In 1950, Amitsur and Levitzki [3] proved that $M_n(K)$ satisfies the standard polynomial of degree $2n$

$$s_{2n} = s_{2n}(x_1, \ldots, x_{2n}) = \sum_{\sigma \in S_{2n}} (\mathrm{sign}\sigma) x_{\sigma(1)} \ldots x_{\sigma(2n)}$$

and that $s_{2n} = 0$ is the only polynomial identity of degree at most $2n$ for $M_n(K)$.

One of the central problems in the theory of PI-algebras is to find a basis of the polynomial identities for the algebra of $n \times n$ matrices, $n \geq 2$. This search is justified by the fact that every T-ideal has a finite basis [49, 50, 68]. The complete answer is known for 2×2 matrices only: The standard identity $s_4(x_1, x_2, x_3, x_4)$ and the Hall identity in two variables $[[x, y]^2, x]$ form a minimal basis for the T-ideal of the polynomial identities of $M_2(K)$ [28] ($[x, y] = xy - yx$ is the commutator of x and y).

For $n > 2$ Leron [53] proved in 1973 that all polynomial identities of degree $2n + 1$ for $M_n(K)$ are consequences of the standard identity s_{2n}. For a more detailed survey we refer the reader to [6].

The following problems related to the polynomial identities of $M_n(K)$ are well known: Let $n \geq 4$.

- What is the minimal degree of the polynomial identities for $M_n(K)$ which do not follow from s_{2n}?
- Are all polynomial identities of degree $2n + 2$ for the matrix algebra $M_n(K)$ consequences of the standard identity s_{2n}? (For $n = 3$ the affirmative answer follows from [31].)

The following conjecture is stated in [6], which includes a more detailed history and survey of PI-algebras.

Conjecture 7.2.1 *All polynomial identities in $M_n(K)$ of degree $2n + 2$ are consequences of the standard identity $s_{2n} = 0$ of degree $2n$ for $n \geq 3$.*

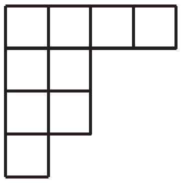
We will prove this conjecture for $n = 4$ and $n = 5$.

Usually the results on polynomial identities are expressed in the language of the representation theory of the symmetric group S_k. It is well known that over a field of characteristic zero every T-ideal is generated by its multilinear elements. If V_k is the vector space of all multilinear polynomials of degree k in $x_1, \ldots, x_k$, the properties of the T-ideal $T(R)$ are determined by the properties of $V_k \cap T(R)$ for every k. The symmetric group S_k acts on the left on V_k by permutation of the variables. Under this action V_k is an S_k-module and the space $V_k \cap T(R)$ is an S_k-submodule of V_k.

We repeat some of the discussion from Chapter 4 for completeness. The irreducible representations of the symmetric group S_k are described by *partitions* of k and *Young* *diagrams*. Let

$$\lambda = (\lambda_1, \ldots, \lambda_r),\ \lambda_1 \geq \ldots \geq \lambda_r \geq 0,\ |\lambda| = \lambda_1 + \ldots + \lambda_r = k$$

be a partition of k (notation $\lambda \vdash k$). We relate to the partition $\lambda = (\lambda_1, \ldots, \lambda_r) \vdash k$ its Young diagram D_λ with r rows of $\lambda_1, \ldots, \lambda_r$ boxes. For example, if $\lambda = (4, 2^2, 1)$ then

$D_\lambda =$ 

We denote by M_λ the irreducible S_k-module corresponding to λ. Then

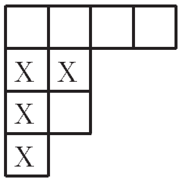
$$V_k \cap T(R) \cong \sum_\lambda m_\lambda(R) M_\lambda, \tag{7.2}$$

where $m_\lambda(R)$ is the multiplicity of M_λ.

The dimensions of the irreducible S_k-modules are given by the *hook formula*. For a partition $\lambda = (\lambda_1, \ldots, \lambda_r) \vdash k$ with $\lambda'_1, \ldots, \lambda'_{\lambda_1}$ being the lengths of the columns of the diagram D_λ, we have

$$\dim M_\lambda = d_\lambda = \frac{n!}{\prod_{i,j} h_{ij}},$$

where $h_{ij} = \lambda_i - j + \lambda'_j - i + 1$ is the *hook length* of the (i,j)-th box. For example, if $\lambda = (4, 2^2, 1)$ then the hook length of the box in position $(2, 1)$ is $h_{21} = 4$:



For a partition $\lambda = (\lambda_1, \ldots, \lambda_r) \vdash k$ a *Young tableau* T_λ is a Young diagram whose boxes are filled in with the integers $1, 2, \ldots, k$ in an arbitrary way. The tableau T_λ is *standard* if its entries increase from top to bottom in the columns and from left to right in the rows. The number of the standard tableaux associated to D_λ is d_λ and the set of all standard tableaux of shape λ is denoted by $D(\lambda)$.

7.3 Computational Approach to Showing All Polynomial Identities of Degree $2n+2$ Follow from $s_{2n}=0$

From [3], s_{2n} is a polynomial identity for $M_n(K)$, i.e. $\langle s_{2n}\rangle^T \subseteq T(M_n(K))$ and thus

$$m_\lambda\left(\langle s_{2n}\rangle^T\right) \le m_\lambda(M_n(K)), \text{ for all } \lambda \vdash 2n+2, \tag{7.3}$$

where $\langle s_{2n}\rangle^T$ denotes the T-ideal of A generated by s_{2n}. To prove that all polynomial identities of degree $2n+2$ follow from s_{2n} we need to show that equality holds in (7.3), namely

$$m_\lambda\left(\langle s_{2n}\rangle^T\right) = m_\lambda(M_n(K)) \text{ for all } \lambda \vdash 2n+2. \tag{7.4}$$

From [7], we know the exact values of $m_\lambda(\langle s_{2n}\rangle^T)$ for all partitions λ of $2n+2$, $n \ge 3$. For $n=4$ and $n=5$ we will compute bounds $n_\lambda(M_n(K)) \ge m_\lambda(M_n(K))$. We will establish that $n_\lambda(M_n(K)) = m_\lambda(\langle s_{2n}\rangle^T)$ for all $\lambda \vdash 2n+2$ thus proving (7.4).

The bounds $n_\lambda(M_n(K)) \ge m_\lambda(M_n(K))$ will be computed as follows [6]:

Let $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_r) \vdash 2n+2$ and $\lambda'_1, \lambda'_2, \ldots, \lambda'_s$ be the lengths of the columns of D_λ, where $s=\lambda_1$. Let T_λ be a standard Young tableaux, whose boxes are filled with the numbers t_{ij} and

$$w_{T_\lambda}(x_1, x_2, \ldots, x_{2n+2}) = \sum_{\sigma_1\in S_{\lambda'_1},\ldots,\sigma_s\in S_{\lambda'_s}} \operatorname{sign}(\sigma_1)\cdot\operatorname{sign}(\sigma_2)\cdot\ldots\cdot\operatorname{sign}(\sigma_s)\cdot x_{z_1}x_{z_2}\ldots x_{z_{2n+2}}, \tag{7.5}$$

where $z_1, z_2, \ldots, z_{2n+2}$ are such that $z_{t_{ij}} = \sigma_j(i)$. (In other words, for every possible permutation of each column of T_λ we form a monomial of degree $2n+2$ whose sign equals the product of the signs of all permutations and each variable $x_{\sigma_j(i)}$ is in position t_{ij}.)

If $N = \operatorname{span}(w_{T_\lambda}(x_1, \ldots, x_{2n+2}) \mid T_\lambda \in D(\lambda))$ then $d_\lambda - \dim_K N \ge m_\lambda(M_n(K))$ [6]. We consider a homogeneous linear system

$$\sum_{T_\lambda\in D(\lambda)} \xi_{T_\lambda} w_{T_\lambda}(x_1, \ldots, x_{2n+2}) = 0, \tag{7.6}$$

where the $\xi_i, i = 1, 2, \ldots, d_\lambda$ are the unknowns and $w_{T_\lambda}(x_1, \ldots, x_{2n+2})$ are the coefficients. For each (random) $2n+2$-tuple $(x_1, \ldots, x_{2n+2})$, where $x_i \in M_n(K)$ we can obtain $n^2 = \dim_K M_n(K)$ equations of the linear system (7.6). Therefore we select m (where m is at least $\frac{d_\lambda}{n^2}$) random $2n+2$-tuples $(x_1^{(j)}, \ldots, x_{2n+2}^{(j)})$, $j = 1, 2, \ldots, m$ and compute $F_{T_\lambda}^{(j)} =$

$w_{T_\lambda}(x_1^{(j)}, \ldots, x_{2n+2}^{(j)}) \in M_n(K)$ for all $j = 1, 2, \ldots, m$ and all $T_\lambda \in D(\lambda)$. We then form an $(n^2 m \times d_\lambda)$ matrix G^λ

$$G^\lambda_{(j-1)n^2+ln+k,\#(T_\lambda)} = \left(F^{(j)}_{\#(T_\lambda)}\right)_{kl}$$

where $k, l = 1, 2, \ldots, n$; $j = 1, 2, \ldots, s$ and $\#(T_\lambda) = 1, 2, \ldots, d_\lambda$ is the index of T_λ in the lexicographical ordering of $T_\lambda \in D(\lambda)$. We define $n_\lambda(M_n(K)) = d_\lambda - \operatorname{rank}(G^\lambda)$. We have

$$n_\lambda(M_n(K)) = d_\lambda - \operatorname{rank}(G^\lambda) \geq d_\lambda - \dim_K(N) \geq m_\lambda(M_n(K)).$$

Since $m_\lambda(M_n(K)) = m_\lambda(M_n(\mathbb{Q}))$ [6] we can work over $\mathbb{Q}$ instead of over K. In fact since a homogeneous linear system has a rational solution only when it has an integer solution we can form D by only using matrices with integer entries. To avoid roundoff and under/overflow we can work over a finite field $\mathbb{Z}_q$, where q is a sufficiently large prime. Clearly

$$n_\lambda(M_n(\mathbb{Z}_q)) \geq n_\lambda(M_n(K)).$$

Our goal is to establish that $n_\lambda(M_n(\mathbb{Z}_q)) = m_\lambda\left(\langle s_{2n}\rangle^T\right)$ for every $\lambda \vdash 2n+2$. We will see that this is the case for $n = 4, 5$.

We compute $n_\lambda(M_n(\mathbb{Z}_q))$ using the following algorithm.

Algorithm 7.3.1 *The following algorithm computes $n_\lambda(M_n(\mathbb{Z}_q))$ for all $\lambda \vdash 2n+2$.*

```
10  for all λ ⊢ 2n + 2
20      for j = 1 : m, where m > d_λ/n^2
30          Generate random matrices x_1, ..., x_{2n+2} ∈ M_n(Z_q)
40          for all T_λ ∈ D(λ)
50              Compute w_{T_λ}^{(j)} according to (7.5)
60              G^λ_{(j-1)n^2+ln+k,#(T_λ)} = (w_{T_λ}^{(j)})_{kl} for l,k = 1,2,...,n,
70      n_λ(M_n(Z_q)) = d_λ - rank(G^λ)
```

Computing each $w_{T_\lambda}^{(j)}$ according to (7.5) costs $(2n+1)n^3 \cdot \lambda_1'! \ldots \lambda_s'!$ and finding the rank of G^λ costs $\frac{2}{3}d_\lambda^3$. Choosing $m = \left\lfloor \frac{d_\lambda}{n^2} \right\rfloor + 10$ in line 20 appears to always be sufficient. Therefore the cost of Algorithm 7.3.1 is

$$\sum_{\lambda \vdash 2n+2} \left((d_\lambda + 10n^2)n(2n+1) \cdot \lambda_1'! \ldots \lambda_s'! + \frac{2}{3} d_\lambda^3 \right). \tag{7.7}$$

The most computationally intensive part is the computation of the polynomials w_{T_λ} using (7.5). Each summation in (7.5) is over a symmetric group of λ'_k elements $k = 1, 2, \ldots, \lambda_1$ and costs $O(\lambda'_k!)$. The cost of lines 10-60 alone is

$$\sum_{\lambda \vdash 2n+2} (d_\lambda + 10n^2)n(2n+1) \cdot \lambda'_1! \ldots \lambda'_s! \tag{7.8}$$

For $n = 5$ this number is

$$1.5 \cdot 10^{13}, \tag{7.9}$$

a computation that would take at least several months on a serial machine[1] .

7.4 New Approach

We here present a new divide-and-conquer algorithm for computing the complete skew-symmetric polynomial

$$f(x_1, \ldots, x_k) = \sum_{\sigma \in S_k} \operatorname{sign}(\sigma) x_{\sigma(1)} x_{\sigma(2)} \cdots x_{\sigma(k)}, \tag{7.10}$$

whose straightforward evaluation according to (7.10) is

$$k!(n^2 + (k-1)n^3). \tag{7.11}$$

We can rewrite (7.10) as

$$f(x_1, \ldots, x_k) = \sum_{i_1 < \ldots < i_{\lfloor k/2 \rfloor}} (-1)^{(i_1, \ldots, i_k)} f(x_{i_1}, \ldots, x_{i_{\lfloor k/2 \rfloor}}) f(x_{i_{\lfloor k/2 \rfloor + 1}}, \ldots, x_{i_k}), \tag{7.12}$$

where the summation is over all $\lfloor k/2 \rfloor$-tuples $1 \leq i_1 < \ldots < i_{\lfloor k/2 \rfloor} \leq k$. The numbers $1 \leq i_{\lfloor k/2 \rfloor + 1} < \ldots < i_k \leq k$ are such that $(i_1, \ldots, i_k)$ is a permutation of $(1, 2, \ldots, k)$ and $(-1)^{(i_1, \ldots, i_k)}$ is its sign. The expression (7.12) can be applied recursively to further reduce the cost.

For example

$$\begin{aligned} f(x_1, x_2, x_3, x_4) &= \sum_{\sigma \in S_4} (-1)^\sigma x_{\sigma(1)} x_{\sigma(2)} x_{\sigma(3)} x_{\sigma(4)} && (7.13) \\ &= [x_1, x_2][x_3, x_4] - [x_1, x_3][x_2, x_4] + [x_1, x_4][x_2, x_3] \\ &\quad + [x_2, x_3][x_1, x_4] - [x_2, x_4][x_1, x_3] + [x_3, x_4][x_1, x_2]. && (7.14) \end{aligned}$$

[1]The arithmetic is performed in $\mathbb{Z}_q$ instead of floating point, which in itself slows the computation by a factor of at least five.

The expression (7.13) involves 72 matrix-matrix multiplications and 23 matrix-matrix additions or subtractions, whereas (7.14) involves only 18 matrix-matrix multiplications and 17 matrix-matrix additions or subtractions.

To estimate the cost of computing $f(x_1, \ldots, x_k)$, where $x_i \in M_n(K)$ using (7.12) recursively we assume that k is a power of 2. In the process of recursive computation we would need to compute all $f(x_i, x_j) = [x_i, x_j] = x_i x_j - x_j x_i$, $i < j$, which costs

$$(2n^3 + n^2) \binom{k}{2} \binom{2}{1}.$$

Computing all $f(x_{i_1}, x_{i_2}, x_{i_3}, x_{i_4})$, $1 \le i_1 < i_2 < i_3 < i_4 \le k$ costs

$$(2n^3 + n^2) \binom{k}{4} \binom{4}{2}.$$

Finally, the cost of computing $f(x_1, x_2, \ldots, x_k)$ is

$$(2n^3+n^2)\left[\binom{k}{k}\binom{k}{k/2} + \binom{k}{k/2}\binom{k/2}{k/4} + \binom{k}{k/4}\binom{k/4}{k/8} + \cdots\right]. \quad (7.15)$$

We estimate this expression by using the Stirling's formula

$$m! = \sqrt{2\pi m} e^{-m} m^m.$$

We have

$$\binom{m}{m/2} = \frac{m!}{(m/2)!(m/2)!} = \frac{\sqrt{2\pi m}(m/e)^m}{\pi m (m/2e)^m} = \frac{2^m}{\sqrt{\pi m/2}}.$$

Therefore (7.15) can be bounded by

$$(2n^3 + n^2) \binom{k}{k/2}^2 \log k \approx (2n^3 + n^2) \frac{\log k}{2\pi k} 4^k.$$

This should be compared with (7.11), which is exponentially larger.

The computational savings resulting from using the expression (7.12) instead of (7.10) to compute the complete skew-symmetric polynomial are presented in Table 7.1. Even though the cost of computing the complete skew-symmetric polynomial using (7.12) recursively is still exponential, it is exponentially faster than (7.10).

Table 7.1: Cost of evaluating the complete skew-symmetric polynomial s_k, $n = 5$

k	Using (7.12)	Using (7.10)
2	$5.5 \cdot 10^2$	$5.5 \cdot 10^2$
4	$5.0 \cdot 10^3$	$1.9 \cdot 10^4$
6	$2.2 \cdot 10^4$	$9.8 \cdot 10^5$
8	$1.5 \cdot 10^5$	$7.2 \cdot 10^7$
10	$7.9 \cdot 10^5$	$8.3 \cdot 10^9$
12	$5.5 \cdot 10^6$	$1.3 \cdot 10^{12}$

The expression (7.12) can be trivially extended to compute quantities of the form

$$f(x_1, \ldots, x_k, y_1, y_2, \ldots, y_k) = \sum_{\sigma \in S_k} \mathrm{sign}(\sigma) x_{\sigma(1)} y_1 x_{\sigma(2)} y_2 \ldots x_{\sigma(k)} y_k,$$

where the x_i's are interlaced with other variables in fixed positions. Since the expression (7.12) cannot be generalized to double sums and $\lambda'_1 \geq \lambda_2 \geq \ldots \geq \lambda'_s$ we will only apply (7.12) for the summation over $S_{\lambda'_1}$.

We incorporate that change in Algorithm 7.3.1 to obtain:

Algorithm 7.4.1 (Divide-and-conquer) *The following algorithm computes $n_\lambda(M_n(\mathbb{Z}_q))$ for all $\lambda \vdash 2n+2$.*

10 `for all` $\lambda \vdash 2n+2$
20   `for` $j = 1 : m$, `where` $m > \frac{d_\lambda}{n^2}$
30   `Generate random matrices` $x_1, \ldots, x_{2n+2} \in M_n(\mathbb{Z}_q)$
40   `for all` $T_\lambda \in D(\lambda)$
50          $w^{(j)}_{T_\lambda} = 0$
60   `for` $\sigma_2 \in S_{\lambda'_2}, \ldots, \sigma_s \in S_{\lambda'_s}$
70   `using` *(7.12)* `compute`
            $a = \sum_{\sigma_1 \in S_{\lambda'_1}} \mathrm{sign}(\sigma_1) \cdot \mathrm{sign}(\sigma_2) \cdot \ldots \cdot \mathrm{sign}(\sigma_s) \cdot x_{z_1} x_{z_2} \ldots x_{z_{2n+2}}$,
            `where` $z_{t_{ij}} = \sigma_j(i)$
80             $w^{(j)}_{T_\lambda} = w^{(j)}_{T_\lambda} + a$
90             $G^\lambda_{(j-1)n^2+ln+k, \#(T_\lambda)} = \left(w^{(j)}_{T_\lambda}\right)_{kl}$ `for` $l, k = 1, 2, \ldots, n,$
100    $n_\lambda(M_n(\mathbb{Z}_q)) = d_\lambda - \mathrm{rank}(G^\lambda)$

The cost of generating the matrices G^λ using Algorithm 7.4.1 (lines 10-90) for $n = 5$ is approximately

$$2.6 \cdot 10^{10}, \tag{7.16}$$

Table 7.2: Nonzero values of $n_\lambda(M_5(\mathbb{Z}_q))$

$\lambda \vdash 12$	$n_\lambda(M_5(\mathbb{Z}_q))$	$m_\lambda\left(\langle s_{2n}\rangle^T\right)$
$(4,2,1^6)$	2	2
$(4,1^8)$	4	4
$(3,2^2,1^5)$	2	2
$(3,2,1^7)$	8	8
$(3,1^9)$	9	9
$(2^3,1^6)$	4	4
$(2^2,1^8)$	8	8
$(2,1^{10})$	5	5
(1^{12})	1	1

i.e. about 600 times less than (7.9), which is the cost of the equivalent computation using the traditional algorithm.

7.5 Parallelism

The lines 30-90 of Algorithm 7.4.1 are embarrassingly parallel for different values of j and can thus be performed on different processors with equal load. For line 100 we developed our own parallel Gaussian elimination which performed the elimination over the finite field $\mathbb{Z}_q$ as opposed to converting an existing package like ScaLAPACK [9] to do arithmetic over $\mathbb{Z}_q$, because this involves programming and this line was not the bottleneck. Although our implementation of Gaussian elimination did not scale linearly with the number of processors, it was satisfactory for our purposes.

7.6 New Results

We completed the computer calculations for the polynomial identities for $M_n(K)$ of degree $2n+2$ for $n = 4, 5$ using 64 processors on the Cray T3E at the Lawrence Berkeley National Laboratory in about 4 hours. The results show that our upper bounds $n_\lambda(M_n(\mathbb{Z}_q))$ coincide with $m_\lambda\left(\langle s_{2n}\rangle^T\right)$. In particular, for $n = 5$ we computed $n_\lambda(M_n(\mathbb{Z}_q)) = 0$ except for the partitions λ in Table 7.2.

In order to give an idea about the size of the calculations we present the results for $n = 5$ and degree 12 in Table 7.3. Each row of the table contains the partition λ, the dimension d_λ of the related irreducible S_{12}-module, and the time for computer calculation

Table 7.3: Computation time in seconds for $\lambda \vdash 12$

$\lambda \vdash 12$	d_λ	sec	$\lambda \vdash 12$	d_λ	sec	$\lambda \vdash 12$	d_λ	sec
(12)	1	0	$(6,2^3)$	1925	35	$(4,3,1^5)$	2079	198
$(11,1)$	11	0	$(6,2^2,1^2)$	3564	252	$(4,2^4)$	1485	74
$(10,2)$	54	0	$(6,2,1^4)$	2100	54	$(4,2^3,1^2)$	3080	657
$(10,1^2)$	55	0	$(6,1^6)$	462	1	$(4,2^2,1^4)$	2376	274
$(9,3)$	154	1	$(5^2,2)$	1320	14	$(4,2,1^6)$	945	18
$(9,2,1)$	320	0	$(5^2,1^2)$	1485	20	$(4,1^8)$	165	1
$(9,1^3)$	165	0	$(5,4,3)$	2112	53	(3^4)	462	3
$(8,4)$	275	0	$(5,4,2,1)$	5775	1106	$(3^3,2,1)$	2112	193
$(8,3,1)$	891	5	$(5,4,1^3)$	3520	245	$(3^3,1^3)$	1650	92
$(8,2^2)$	616	2	$(5,3^2,1)$	4158	366	$(3^2,2^3)$	1320	48
$(8,2,1^2)$	945	6	$(5,3,2^2)$	4455	491	$(3^2,2^2,1^2)$	2673	380
$(8,1^4)$	330	0	$(5,3,2,1^2)$	7700	19622	$(3^2,2,1^4)$	1925	146
$(7,5)$	297	0	$(5,3,1^4)$	3564	976	$(3^2,1^6)$	616	5
$(7,4,1)$	1408	16	$(5,2^3,1)$	3520	908	$(3,2^4,1)$	1155	33
$(7,3,2)$	1925	40	$(5,2^2,1^3)$	3696	991	$(3,2^3,1^3)$	1408	79
$(7,3,1^2)$	2376	69	$(5,2,1^5)$	1728	107	$(3,2^2,1^5)$	891	18
$(7,2^2,1)$	2079	50	$(5,1^7)$	330	1	$(3,2,1^7)$	320	1
$(7,2,1^3)$	1728	29	(4^3)	462	3	$(3,1^9)$	55	0
$(7,1^5)$	462	1	$(4^2,3,1)$	2970	514	(2^6)	132	0
(6^2)	132	0	$(4^2,2^2)$	2640	412	$(2^5,1^2)$	297	1
$(6,5,1)$	1155	9	$(4^2,2,1^2)$	4455	1740	$(2^4,1^4)$	275	1
$(6,4,2)$	2673	100	$(4^2,1^4)$	1925	146	$(2^3,1^6)$	154	0
$(6,4,1^2)$	3080	148	$(4,3^2,2)$	2970	516	$(2^2,1^8)$	54	0
$(6,3^2)$	1650	26	$(4,3^2,1^2)$	4158	397	$(2,1^{10})$	11	0
$(6,3,2,1)$	5632	876	$(4,3,2^2,1)$	5775	1003	(1^{12})	1	0
$(6,3,1^3)$	3696	209	$(4,3,2,1^3)$	5632	861			

in seconds on 64 processors.

We shall state as a theorem the result obtained as a combination of theoretical conclusions and computer experiments:

Theorem 7 *For $n=4$ and $n=5$ all polynomial identities of degree $2n+2$ for $M_n(K)$ are consequences of the standard identity $s_{2n}=0$ of degree $2n$.*

Chapter 8

Conclusions and Open Problems

We have exploited the intrinsic combinatorial, algebraic or physical structure of several problems and have obtained algorithms that are faster and/or more accurate. Using these algorithms we proved theoretical results in areas ranging from astrophysics to numerical linear algebra to algebras with polynomial identities.

We derived algorithms for accurate computation of the SVD of polynomial Vandermonde and M-matrices, which compute even the tiniest singular values accurately regardless of condition numbers.

We explained the phenomenon of the Björck-Pereyra methods for all totally positive matrices whose initial minors are computable accurately and efficiently. We presented a new method for computing the Schur function that is exponentially faster than previous methods and only linear in the number of variables. This in turn allowed us to derive Björck-Pereyra type methods for the class of generalized Vandermonde matrices and proved that in many cases the methods are efficient and always accurate.

We classified a large number of structured matrices for which accurate computations and decompositions are possible. Further results appear in [25].

We exploited the properties of computer arithmetic to generalize Harry Diamond's theorem, and to raise important questions related to the consistent computer implementation of transcendental functions.

We derived structure-exploiting algorithms for the analysis of the cosmic microwave background radiation that will be used by astrophysicists and will make the analysis of future datasets possible.

We implemented a divide-and-conquer algorithm for computing complete symmetric polynomials in matrix algebras and partly proved a conjecture on the generators of the T-ideals in matrix algebras.

There are a number of open problems remaining, including the following:

1. Can we compute an accurate SVD in polynomial time for totally positive generalized Vandermonde matrices?

2. For which other classes of structured totally matrices do efficient methods exist for solving linear systems, inversion or computing SVDs to high relative accuracy?

3. Can we compute accurate SVDs of confluent polynomial Vandermonde matrices?

4. Which other entries of Table 4.1 can we fill in?

5. We conjecture that the true condition number of the singular values of a polynomial Vandermonde matrix, involving orthonormal polynomials, is the reciprocal of the smallest relative gap between the input data, i.e. the reciprocal of the distance to the nearest ill-posed problem.

6. More generally, we conjecture that having a condition number proportional to the reciprocal of the smallest relative gap between the input data is a *necessary condition* for the existence of an accurate algorithm in the sense of this thesis.

Bibliography

[1] M. Abramowitz and I. Stegun (editors). *Handbook of mathematical functions*, volume 55 of *NBS, Appl. Math. Ser.* U.S. Government Printing office, Washington, D.C., 1965.

[2] Attahiru Sule Alfa, Junggong Xue, and Qiang Ye. Entrywise perturbation theory for diagonally dominant M-matrices with applications. *Numer. Math.*, 90:401–414, 2002.

[3] S. A. Amitsur and J. Levitzki. Minimal identities for algebras. In *Proc. Amer. Math. Soc.*, volume 1, pages 449–463, 1950.

[4] ANSI/IEEE, New York. *IEEE Standard for Binary Floating Point Arithmetic*, Std 754-1985 edition, 1985.

[5] Zhaojun Bai, Mark Fahey, and Gene Golub. Some large scale matrix computation problems. *J. Comput. Appl. Math.*, 74:71–89, 1996.

[6] F. Benanti, J. Demmel, V. Drensky, and P. Koev. Computer approach to polynomial identities for algebras, 2002. in preparation.

[7] F. Benanti and V. Drensky. On the consequences of the standard polynomial. *Commun. Algebra*, 26:4243–4275, 1998.

[8] Å Björck and V. Pereyra. Solution of Vandermonde systems of equations. *Math. Comp.*, 24(112):893–903, 1970.

[9] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, 1997.

[10] L. S. Blackford, A. Cleary, J. Demmel, I. Dhillon, J. J. Dongarra, S. Hammarling, A. Petitet, H. Ren, K. Stanley, and R. C. Whaley. Practical experience in the dangers

of heterogeneous computing. LAPACK Working Note No.112. Technical Report CS-96-330, Department of Computer Science, University of Tennessee, 107 Ayres Hall, Knoxville, TN 37996-1301, USA, 1996. `www.netlib.org/lapack/lawns/lawn112.ps`.

[11] L.S. Blackford, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, A. Petitet, H. Ren, K. Stanley, and R.C. Whaley. Practical experience in the dangers of heterogeneous computing. In J. Waśniewski, J. Dongarra, K. Madsen, and D. Olesen, editors, *Applied Parallel Computing, Industrial Computation and Optimization*, pages 57–64, Berlin Heidelberg New York, 1996. Third International Workshop, PARA '96, Springer-Verlag. Lecture Notes in Computer Science, Vol. 1184.

[12] T. Boros, T. Kailath, and V. Olshevsky. The fast parallel Björck-Pereyra-type algorithm for parallel solution of Cauchy linear equations. *Linear Algebra Appl.*, 302/303:265–293, 1999.

[13] T. Boros, T. Kailath, and V. Olshevsky. Pivoting and backward stability of fast algorithms for solving Cauchy linear equations. *Lin. Alg. Appl.*, 343/344:63–99, 2002.

[14] Julian Borrill. MADCAP: The Microwave Anisotropy Dataset Computational Analysis Package. In *Proceeedings of the 5th European SGI/Cray MPP Workshop*, 1999.

[15] Christopher Cantalupo, 2002. private communication.

[16] P. de Bernardis *et al.* A flat universe from high-resolution maps of the cosmic microwave background radiation. *Nature*, 404:955–959, April 2000.

[17] Carl de Boor and Alan Pinkus. Backward error analysis for totally positive linear systems. *Numer. Math.*, 27:485–490, 1977.

[18] J. Demmel. On condition numbers and the distance to the nearest ill-posed problem. *Num. Math.*, 51(3):251–289, July 1987.

[19] J. Demmel. *Applied Numerical Linear Algebra.* SIAM, 1997.

[20] J. Demmel. Accurate SVDs of structured matrices. *SIAM J. Mat. Anal. Appl.*, 21(2):562–580, 1999.

[21] J. Demmel, J. J. Dongarra, S. Hammarling, S. Ostrouchov, and K. Stanley. The dangers of heterogeneous network computing: Heterogenous networks considered harmful. In

Proceedings Heterogeneous Computing Workshop '96, pages 64–71. IEEE Computer Society Press, 1996.

[22] J. Demmel, M. Gu, S. Eisenstat, I. Slapničar, K. Veselić, and Z. Drmač. Computing the singular value decomposition with high relative accuracy. *Lin. Alg. Appl.*, 299(1–3):21–80, 1999.

[23] J. Demmel and W. Kahan. Accurate singular values of bidiagonal matrices. *SIAM J. Sci. Stat. Comput.*, 11(5):873–912, September 1990.

[24] J. Demmel and P. Koev. Accurate SVDs of polynomial Vandermonde matrices, involving orthogonal polynomials. in preparation.

[25] J. Demmel and P. Koev. Necessary and sufficient conditions for accurate and efficient rational function evaluation and factorizations of rational matrices. In *Structured matrices in mathematics, computer science, and engineering. II (Boulder, CO, 1999)*, pages 117–143. Amer. Math. Soc., Providence, RI, 2001.

[26] H. Diamond. Stability of rounded off inverses under iteration. *Math. Comput.*, 32:227–232, 1978.

[27] S. Dong and K. Liu. Stochastic estimation with z_2 noise. *Phys. Lett. B*, 328:130–136, 1994.

[28] V. Drensky. A minimal basis for the identities of a second-order matrix algebra over a field of characteristic 0 (Russian). *Algebra i Logika*, 20:282–290, 1981. English translation: Algebra and Logic, 20:188–194, 1981.

[29] V. Drensky. *Free Algebras and PI-Algebras.* Springer-Verlag, Singapore, 1999.

[30] V. Drensky and G. M. Piacentini Cattaneo. A central polynomial of low degree for 4×4 matrices. *J. Algebra*, 168:469–478, 1994.

[31] V. Drensky and A. Kasparian. Polynomial identities of eighth degree for 3×3 matrices. In *Annuaire de l'Univ. de Sofia*, volume 77, pages 175–195. Fac. de Math. et Mecan., Livre 1, Math., 1983.

[32] V. Drensky and Ts. G. Rashkova. Weak polynomial identities for the matrix algebras. *Commun. Algebra*, 21:3779–3795, 1993.

[33] James R. Driscoll and Dennis M. Healy Jr. Computing Fourier transforms and convolutions on the 2-sphere. *Adv. in Appl. Math.*, 15:202–250, 1994.

[34] Alan Edelman, 2002. private communication.

[35] E. Formanek. *The polynomial Identities and Invariants of $n \times n$ Matrices*, volume 78 of *CBMS Regional Conf. Series in Math.* AMS, Providence RI, 1991.

[36] Matteo Frigo and Steven G. Johnson. Fastest Fourier Transform in the West. `http://www.fftw.org`.

[37] M. Gasca and J. M. Pena. On factorizations of totally positive matrices. In *Total Positivity and its Applications*. Kluwer Acad. Publ., Dordrecht, 1996.

[38] Walter Gautschi. Construction of Gauss-Christoffel quadrature formulas. *Math. Comp.*, 22:251–270, 1968.

[39] Walter Gautschi. The condition of Vandermonde-like matrices involving orthogonal polynomials. *Lin. Alg. Appl.*, 52/53:293–300, 1983.

[40] Walter Gautschi. Is the recurrence relation for orthogonal polynomials always stable? *BIT*, 33:277–284, 1993.

[41] Walter Gautschi. Orthogonal polynomials: applications and computation. *Acta Numerica*, pages 45–119, 1996.

[42] G. Golub and C. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, 3rd edition, 1996.

[43] N. J. Higham. Stability analysis of algorithms for solving confluent Vandermonde-like systems. *SIAM J. Mat. Anal. Appl.*, 11(1):23–41, 1990.

[44] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, PA, 1996.

[45] M. Hutchinson. A stochastic estimator of the trace of the influence matrix for Laplacian smoothing splines. *Commun. Stat. Simula.*, 18:1059–1076, 1989.

[46] N. Jacobson. *PI-algebras: An Introduction*, volume 441 of *Lecture Notes in Math.* Springer Verlag, Berlin-New York, 1979.

[47] G. James and A. Kerber. *Representation Theory of the Symmetric Group*, volume 16 of *Encyclopedia of mathematics and its applications*. Addison-Wesley, 1981.

[48] William Kahan. Mathematics written in sand. In *Proc. Joint Statistical Mtg. of the American Statistical Association*, pages 12–26, 1983. `http://www.cs.berkeley.edu/~wkahan/MathSand.pdf`.

[49] A. R. Kemer. Finite basis property of identities of associative algebras (Russian). *Algebra i Logika*, 26:597–641, 1987. Translation: Algebra and Logic, 26:362-397, 1988.

[50] A. R. Kemer. *Ideals of Identities of Associative Algebras*, volume 87 of *Translations of Math. Monographs*. AMS, Providence, RI, 1991.

[51] A. Kerber and A. Kohnert. Symmetrica 1.0, University of Bayreuth, Germany, 1992. `http://www.mathe2.uni-bayreuth.de/axel/symneu_engl.html`.

[52] Vincent Lefèvre and Jean-Michel Muller. Worst cases for correct rounding of the elementary functions in double precision. Research report no.2000-35, Laboratoire de l'Informatique du Parallélisme, École Normale Supérieure de Lyon, 2000.

[53] U. Leron. Multilinear identities of the matrix ring. *Trans. Amer. Math. Soc.*, 183:175–202, 1973.

[54] I. G. MacDonald. *Symmetric functions and Hall polynomials*. Oxford University Press, 2nd edition, 1995.

[55] The MathWorks, Inc., Natick, MA. *MATLAB Reference Guide*, 1992.

[56] David W. Matula. The base conversion theorem. *Proc. Amer. Math. Soc.*, 19:716–723, 1968.

[57] Leonard Gaines Monk. *Elementary-recursive decision procedures*. Ph.D. dissertation, University of California, Berkeley, 1975.

[58] C. O'Cinneide. Relative-error for the LU decomposition via GTH algorithm. *Numer. Math.*, 73:507–519, 1996.

[59] Siang Peng Oh, David N. Spergel, and Gary Hinshaw. An efficient technique to determine the power spectrum from cosmic microwave background sky maps, 1998. `http://lanl.arXiv.org/ps/astro-ph/9805339`.

[60] D. Priest. Algorithms for arbitrary precision floating point arithmetic. In P. Kornerup and D. Matula, editors, *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 132–145, Grenoble, France, June 26-28 1991. IEEE Computer Society Press.

[61] C. Procesi. *Rings with Polynomial Identities.* Marcel Dekker, New York, 1973.

[62] L. Reichel and L. N. Trefethen. Eigenvalues and pseudo-eigenvalues of Toeplitz matrices. *Lin. Alg. Appl.*, 162–164:153–185, 1992.

[63] J. Reiser and D. Knuth. Evading the drift in floating point addition. *Information Processing Letters*, 3:84–87, 1975.

[64] L. H. Rowen. *Polynomial Identities in Ring Theory.* Acad. Press, New York, 1980.

[65] A. Schönhage and V. Strassen. Schnelle Multiplikation grosser Zahlen. *Computing (Arch. Elektron. Rechnen)*, 7:281–292, 1971.

[66] George Smoot. The Cosmic Microwave Background Spectrum, 1997. `http://lanl.arXiv.org/ps/astro-ph/9705101`.

[67] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, 1996.

[68] W. Specht. Gesetze in Ringen I. *Math. Z.*, 52:557–589, 1950.

[69] Richard Stanley. Some combinatorial properties of Jack symmetric functions. *Adv. Math.*, 77:76–115, 1989.

[70] G. Szegö. *Orthogonal Polynomials.* American Mathematical Society, 1967.

[71] H. Van de Vel. Numerical treatment of a generalized Vandermonde system of equations. *Linear Algebra and Its Appl.*, 17:149–179, 1977.

[72] S. Wolfram. *Mathematica: A system for doing mathematics by computer.* Addison-Wesley, 1988.