THE KRONECKER PRODUCT IN APPROXIMATION AND FAST TRANSFORM GENERATION

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Nikos P. Pitsianis

January 1997

THE KRONECKER PRODUCT IN APPROXIMATION AND FAST TRANSFORM GENERATION

Nikos P. Pitsianis, Ph.D.

Cornell University 1997

Two aspects of the Kronecker product are studied. First, we start with the Kronecker product approximation problem. Given a matrix A and a factorization of its dimensions, we find matrices B and C of the corresponding dimensions whose Kronecker product is as close as possible to A in Frobenius norm. Various constrained Kronecker product approximation problems are also considered along with examples.

Second, we develop a source-to-source compiler that processes matrix factorization formulae to generate efficient implementations of fast transformation algorithms. Our goal is to automate the implementation of efficient FFT and other fast transforms that can be characterized using Kronecker product factorizations. The Kronecker product notation plays a crucial role, allowing the simplification of expression of fast transform algorithms (Walsh-Hadamard, Haar, Slant, Hartley) and capitalizing on the experience from the FFT field. The compiler is based on a set of term rewriting rules that translate high level matrix descriptions into loops and assignment statements in any imperative programming language (parallel or sequential). We study the Haar and Daubechies wavelet algorithms with an effort to express them using matrix language. We provide back-end translators for FORTRAN, C and MATLAB.

Biographical Sketch

Nikos Pitsianis was born in 1964 at the village of Pydna in northern Greece, next to the battleground where in 168 b.c. the Roman general Lucius Aemilius Paullus defeated Perseus, the son of Philip V, king of Macedonia.

Nikos studied Mathematics at the Aristotle University of Thessaloniki. He began graduate studies in Statistical Physics but the fall of 1988 he was lured to the United States as a visiting scientist with I.B.M.'s *Scientific and Engineering Computations* department in Kingston, NY for two years. Recognizing his limited knowledge and foreseeing the demise of mainframe computers, he applied to graduate schools of fine Universities. In the fall of 1990 he was admitted in the graduate program of the department of Computer Science at Cornell University, under the supervision of Prof. Charles Van Loan.

To my family, teachers and friends who believed and supported me all those long Ithaca years.

Acknowledgements

I am grateful to Bob Plemmons who read an early draft of this thesis and provided insightful comments. I thank Paul Jackson and Bob Constable for their positive feedback on the Transformations chapter. Finally, I would like to express my deep appreciation for the help, support and guidance of my advisors Eva Tardos, Rich Zippel and most especially, Charles Van Loan. This work was supported in part by the Cornell Theory Center which receives funding from members of its Corporate Research Institute, the National Science Foundation, the Advanced Research Projects Agency, the National Institutes of Health, New York State, and IBM corporation.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Kronecker Product

The *Kronecker product* of two matrices B and C of sizes $m_B \times n_B$ and $m_C \times n_C$ respectively, is a $(m_B m_C) \times (n_B n_C)$ matrix defined by

$$B \otimes C = \begin{bmatrix} b_{11}C & b_{12}C & \cdots & b_{1n_B}C \\ b_{21}C & b_{22}C & \cdots & b_{2n_B}C \\ \vdots & \vdots & \ddots & \vdots \\ b_{m_B 1}C & b_{m_B 2}C & \cdots & b_{m_B n_B}C \end{bmatrix}. \tag{1.1}$$

This means that every (i, j) block of A of size $m_C \times n_C$ is a multiple of the corresponding b_{ij} element of matrix B and matrix C. Some authors define the right-hand side of (1.1) as $C \otimes B$, e.g. [RM89]. The Kronecker product is also known as the *tensor product* and *direct matrix product*.

Kronecker product notation arises in many different areas of science and engineering. Very important to us is that the Kronecker product simplifies the expression of many fast transform algorithms. For example, different fast Fourier transform (FFT) algorithms correspond to different sparse matrix factorizations of the Discrete Fourier Transform (DFT) whose factors involve Kronecker products [VL92]. The Kronecker product is used extensively in digital image processing. All separable image transforms like Fourier, Hadamard, Haar and Slant can be factored using Kronecker products [GW87,Jai89,Pra91].

In matrix-geometric solutions to stochastic models, linear systems with matrices of the form

$$I_m \otimes S_n + T_m \otimes I_n$$

are very common [Neu81]. Similarly structured matrices come up in studies of matrix methods for queuing problems in [Kau83]. The same type of matrix also appears in applications in inventory-control models [Zip88]. The solution of linear systems of the form

$$(A_1 \otimes A_2 \cdots \otimes A_k)x = b$$

is studied in [PS73] and efficient algorithms are produced in the area of approximation of multidimensional functionals. Two algorithms for the least squares solution to $(A \otimes B)x = t$ based on the QR and SVD factorizations of the Kronecker factors are presented in [FF94]. The Kronecker product is used in *array algebra*, a generalization of the vector, matrix and tensor algebra in [Rau80] for applications to fast transforms.

The properties of the Kronecker product are used extensively in [QD88] to determine the stability robustness of state space models. A least squares problem with a matrix consisting of Kronecker product blocks is solved to estimate the unknown parameters of nonlinear systems [CS88].

In an application of restoring noisy images, the matrix-vector product

$$(R_1 \otimes R_2)(R_1 \otimes R_2 + \alpha I)^{-1} g$$

has to be calculated, where matrices R_1 and R_2 are Toeplitz [LCCM89].

The Kronecker product is often used as an elementary operation to define more complex ones. The *strong Kronecker product* is defined in the area of combinatorial theory as a new multiplication tool for orthogonal matrices [DLS94]. Given two matrices A and B written as n_1-by-n_2 and n_2-by-n_3 block matrices respectively, the n_1-by-n_3 block matrix C is said to be the strong Kronecker product of A and B if and only if, each block of C can be written as

$$C_{ij} = \sum_{k=1}^{n_2} (A_{ik} \otimes B_{kj}). \tag{1.2}$$

Note that the operation is well defined only after the parameters n_1, n_2 and n_3 are set. The strong Kronecker product preserves orthogonality and is associative.

Another generalization of the Kronecker product is formulated in order to study large-scale systems [HC89].

For other applications of the Kronecker product, see [Ste91]. A history of the Kronecker product is presented in [HPS83].

1.2 Kronecker Products and Matrix Factorizations

By way of introduction, we state some important properties of the Kronecker product. We do not present any proofs because either these proofs are straightforward or have appeared elsewhere. For an extensive study of Kronecker products of matrices and their applications we refer the reader to [Ste91]. In order to avoid repetition of matrix constraints such as matrix size so that an operation is valid, we assume that identities are valid only when all matrices have appropriate sizes so all operations are well defined.

The Kronecker product is associative:

$$(A \otimes B) \otimes C = A \otimes (B \otimes C). \tag{1.3}$$

If A, B, C and D are matrices and AC and BD are defined, then

$$(AC) \otimes (BD) = (A \otimes B)(C \otimes D). \tag{1.4}$$

The Kronecker product distributes over addition:

$$(A + B) \otimes (C + D) = A \otimes C + A \otimes D + B \otimes C + B \otimes D. \tag{1.5}$$

The product of a Kronecker product times a vector is a matrix product:

$$(A \otimes B)x = \mathrm{vec}(BXA^T) \tag{1.6}$$

with $x = \mathrm{vec}(X)$. Here, the operator $\mathrm{vec}(X)$ creates a vector by stringing together, one-by-one, the columns of matrix $X \in \mathbb{R}^{m \times n}$:

$$\mathrm{vec}(X) = \begin{bmatrix} X(:,1) \\ \vdots \\ X(:,n) \end{bmatrix}. \tag{1.7}$$

A survey of vec operator properties can be found in [HS81].

The transpose, conjugate, and inverse of a Kronecker product is the Kronecker product of the transposes, conjugates, and inverses respectively of the factors:

$$\begin{aligned} (A \otimes B)^T &= A^T \otimes B^T \\ \overline{(A \otimes B)} &= \overline{A} \otimes \overline{B} \\ (A \otimes B)^{-1} &= A^{-1} \otimes B^{-1}. \end{aligned}$$

As a consequence to the above properties we can see that the Kronecker product preserves the properties of the factors. For example, if $A = A^T$ and $B = B^T$, then

$(B\otimes C)^T = B\otimes C$. Likewise, the Kronecker product of positive definite, non-negative, orthogonal and permutation matrices retain the corresponding property of their factors. Kronecker products of sparse matrices are also sparse at the block level.

From (1.4) it follows that the well known factorizations LU, QR, Cholesky, SVD, and Schur have nice Kronecker product extensions:

$$\left.\begin{array}{rcl} A &=& G_A G_A^T \\ B &=& G_B G_B^T \end{array}\right\} \Rightarrow \quad A\otimes B = (G_A\otimes G_B)(G_A\otimes G_B)^T$$

$$\left.\begin{array}{r} P_A A = L_A U_A \\ P_B B = L_B U_B^T \end{array}\right\} \Rightarrow \quad (P_A\otimes P_B)(A\otimes B) = (L_A\otimes L_B)(U_A\otimes U_B)$$

$$\left.\begin{array}{r} A = Q_A R_A \\ B = Q_B R_B \end{array}\right\} \Rightarrow \quad A\otimes B = (Q_A\otimes Q_B)(R_A\otimes R_B)$$

$$\left.\begin{array}{r} A = U_A\Sigma_A V_A^T \\ B = U_B\Sigma_B V_B^T \end{array}\right\} \Rightarrow \quad A\otimes B = (U_A\otimes U_B)(\Sigma_A\otimes\Sigma_B)(V_A\otimes V_B)^T$$

$$\left.\begin{array}{r} Q_A^H A Q_A = T_A \\ Q_B^H B Q_B = T_B \end{array}\right\} \Rightarrow \quad (Q_A\otimes Q_B)^H(A\otimes B)(Q_A\otimes Q_B) = (T_A\otimes T_B)$$

Some factorizations that are not quite preserved are the real Schur decomposition and the rank revealing QR.

The above identities bring up an important point. Suppose we have to solve the system $(A\otimes B)x = c$ with A, $B\in\mathbb{R}^{n\times n}$. If we ignore the fact that the system matrix is a Kronecker product, it takes $O(n^6)$ flops to solve it. If instead we use the fact that the system $(A\otimes B)x = c$ is equivalent to

$$BXA^T = C$$

with $x = \mathrm{vec}(X)$ and $c = \mathrm{vec}(C)$ then we only need to solve two multiple right-hand-side systems, and this can be done in only $O(n^3)$ flops:

- Solve $BY = C$ for Y in $O(n^3)$ flops using the LU factorization of B.
- Solve $XA^T = Y$ for X in $O(n^3)$ flops using the LU factorization of A.

Similarly, suppose we have to calculate the matrix vector product $(A \otimes B)x$, where $A, B \in \mathbb{R}^{n\times n}$. Without considering the Kronecker product structure, it takes $O(n^4)$ flops to perform the matrix-vector product, while the calculation of the equivalent expression BXA^T requires only $O(n^3)$ flops.

The rank of a matrix that is a Kronecker product is equal to the product of the ranks of its factors

$$\text{rank}(A \otimes B) = \text{rank}(A)\text{rank}(B).$$

If $A \in \mathbb{R}^{m\times m}$ and $B \in \mathbb{R}^{n\times n}$ then

$$\text{tr}(A \otimes B) = \text{tr}(A)\text{tr}(B)$$

$$\det(A \otimes B) = \det(A)^n \det(B)^m$$

where tr is the matrix trace and det is the determinant.

1.3 Kronecker Products and Permutations

A zero-one square matrix P is a *permutation* matrix if

$$PP^T = P^TP = I.$$

Note that the Kronecker product of two permutation matrices is a permutation matrix.

A very important permutation matrix to us is the *stride permutation* $P_{n,p}$ where $n = pm$. For example,

$$P_{8,2}x = [x_1, x_3, x_5, x_7, x_2, x_4, x_6, x_8]^T$$

and

$$P_{8,4}x = [x_1, x_5, x_2, x_6, x_3, x_7, x_4, x_8]^T.$$

That is, the application of $P_{n,p}$ on a vector x of length n reshuffles the elements of the vector as follows:

$$[x_1, x_{1+p}, x_{1+2p}, \cdots, x_2, x_{2+p}, x_{2+2p}, \cdots, x_p, x_{2p}, x_{3p}, \cdots]^T.$$

Obviously

$$P_{n,1} = P_{n,n} = I_n.$$

When p divides n, then

$$P_{n,p}^T = P_{n,\frac{n}{p}}.$$

If n is not a multiple of p, then $P_{n,p}^T$ is not a stride permutation.

Another way to visualize the effect of stride permutation is to consider its relationship with matrix transposition. If $x = \text{vec}(X)$, with $X \in \mathbb{R}^{p\times m}$, then

$$P_{pm,p}x = \text{vec}(X^T). \tag{1.8}$$

If we consider a linear memory where a matrix of size p-by-q is stored in column-major order, then the action of accessing this matrix in row-major order is equivalent in permuting the memory vector with $P_{pq,p}$.

The stride permutation permits the formal description of indexing operations arising in fast transform applications. For example, the bit reversal operation required in the Cooley-Tukey FFT is a permutation matrix that is equal to a product of Kronecker products involving identity and stride permutation matrices. Here is the length 16 bit reversal

$$R_{16,2} = P_{16,8}(I_2 \otimes P_{8,4})(I_4 \otimes P_{4,2}) = \prod_{q=1}^{4} (I_{2^{q-1}} \otimes P_{8,2^{4-q}}).$$

It is possible to relate $A \otimes B$ to $B \otimes A$ using stride permutations. It is the *commutative property* of the Kronecker product. for any matrices $A \in \mathbb{R}^{m_A \times n_A}$ and $B \in \mathbb{R}^{m_B \times n_B}$, then

$$A \otimes B = P_{m_A m_B, m_A}(B \otimes A) P_{n_A n_B, n_B}. \tag{1.9}$$

This commutative property of the Kronecker product is very useful in the development of fast transform algorithms and in the implementation of these algorithms on sequential and parallel computers, because the permutation matrices formally express memory access patterns and the communication required.

Matrix $P_{n,p}$ is also known as the vec-permutation matrix [HS81].

1.4 Contributions and Outline

This thesis is a study of various aspects of the Kronecker product and its application in expressing, simplifying, and implementing fast transform algorithms like fast Fourier and wavelet transforms. We start by introducing the Kronecker product approximation problem. In this problem we are given a matrix A and a factorization of its dimensions and find two matrices B and C of the corresponding dimensions, so that their Kronecker product is the closest to A in the Frobenius norm. We have seen that it is highly advantageous to exploit the structure present in a Kronecker product and the nearest Kronecker product makes this possible in certain applications through preconditioning.

In Chapter 2 we present three different methods for solving the nearest Kronecker product problem. One relies on the explicit singular value decomposition (SVD), one method is based on the Lanczos SVD, and one takes a separable least squares approach using the power method. We also show that the strong Kronecker product can be obtained by reordering the elements of a sum of Kronecker products, thus relating it with the generalized Kronecker products that arise in fast wavelet transforms. An application with preconditioning is given.

In Chapter 3 we examine the constrained Kronecker product approximation problem $A \approx B \otimes C$, where the factors B and C are required to have some structural property. For example, we may require B and C to be banded, symmetric, Toeplitz, Hankel, circulant, stochastic, or orthogonal. Several methods for solving these problems are developed and tested. An application from radar signal processing that requires a constrained Kronecker product approximation is given.

The Kronecker product is a great tool for the formal description of fast transform algorithms (Fourier, Hadamard, wavelet etc). Many different transform algorithms in signal processing can be expressed as matrix-vector products and different algorithms correspond to different factorizations of the transform matrix. The use of matrix notation to describe these algorithms unifies the literature and simplifies implementation. During the last few years, there has been a tremendous activity in the field of wavelets. However, inconsistent algorithmic notation and low level description of algorithms as multiple sums of subscripted scalars have been retarding clean implementations and lead to duplication of results.

The fast transform algorithms that come up in these areas, particularly in wavelets, are having an increasingly important role to play and require a great variety of code to be developed. With the Kronecker product notation, it is possible to automate the production of such code and thus be able to prove and guarantee program correctness. Given a transform-matrix factorization formula, we show that there is a "mechanical" way to process it and generate efficient implementations of loop nests to be executed in sequential and parallel computers. In Chapter 4 we present a methodology for automating the implementation of efficient Kronecker product code (with emphasis on FFT), using symbolic manipulation and code transformations. We construct a set of rules that describe a source-to-source compiler from a high level matrix description language, to loops and assignment statements that can be translated to any imperative programming language. We provide such translators for FORTRAN, C and MATLAB . We run those generated codes on various workstations and verify that their performance is comparable to highly optimized hand-crafted code.

Three appendices include code for the Chapters 2 and 3 algorithms and the program generator of Chapter 4. We provide MATLAB code for the Kronecker product approximation and for the solution of least squares problems with equality and inequality linear constraints. We also provide the source code for the code generation system written in Maple and Mathematica. Samples of the generated source code in FORTRAN are also given. The software requirements for using the provided code are:

- MATLAB version 4.0 or later.
- Maple V Release 3 or later.
- Mathematica version 2.2.
- Any standard FORTRAN 77 compiler.
- Any standard ANSI C compiler.

No hardware constraints are imposed by our code other than the requirements of the above software packages. For detailed information on the correct use of the above software, see Appendix B. All code is available on-line from

`ftp://ftp.cs.cornell.edu/pub/nikos/`.

1.5 Related Work

The work presented in this thesis touches many different areas of computer and computational science.

1.5.1 Approximation with Kronecker Products

There has not been any work related directly to the Kronecker product approximation problem. Marginally related is the *constrained Procrustes* problem (CPP)[1], where the object is to find a matrix X that minimizes the Frobenius norm

$$\|AX - B\|_F$$

for given matrices A and B, where X belongs to a closed convex cone [Hig88, AE93]. The CPP becomes the Kronecker product approximation when A is the identity matrix and we require X to be the Kronecker product of two matrices of specific sizes.

If A is a m-by-m block matrix with n-by-n Toeplitz blocks, the minimizer of $\|A - X\|_F$ over all m-by-m block matrices with n-by-n circulant blocks has been proposed as a preconditioner for block systems of Toeplitz blocks in [CJ92]. This

[1]Procrustes, in Greek mythology, was a robber who lived near Eleusis in Attica. Originally named Damastes or Polypemon, he acquired the name Procrustes ("The Stretcher") because he tortured his victims by cutting them down to fit his bed if they were too tall, or hammering and stretching them if they were too short. He was captured by the hero Theseus, who inflicted upon Procrustes the same kind of torture that he had imposed upon his victims.

work is based on approximating a Toeplitz matrix by a circulant [Cha88]. Our work on Kronecker product preconditioning in §2.5 is similar in spirit.

1.5.2 FFT and the Kronecker Product Notation

The FFT algorithm was introduced by Cooley and Tukey who reduced the complexity of the transform from $O(n^2)$ to $O(n\log(n))$ [CT65]. A lively account of how the FFT gained acceptance is given in [Coo87]. Not surprisingly, the history of FFT goes back all the way to Gauss [HJB85].

The importance of the FFT promoted the development of highly optimized implementations, via the use of vector hardware [Swa82], multiprocessors [BHSO87], and vector processors [Heg95]. Hundreds of articles have been published about implementations for specific machines, see for example [LCT93], [AGZ94] and [JKFM89]. Special hardware designs have been defined for performing FFTs. The Kronecker product notation is playing an increasingly important role as the number of applications and architectures explodes. Various signal processing algorithms are described with generalized Kronecker products in [RM89]. In the monograph [VL92] it is shown how the different fast transform algorithms are very similar when seen from the matrix factorization point-of-view.

This thesis extends the work presented in [RM89] and [VL92] by the provision of an automated tool to process the Kronecker product notation in order to generate code. The same notation is used in [GCT92a,GCT92b,GCT91,JJRT90] as a mathematical programming language for modeling, designing and implementing FFT and convolution algorithms, however no practical way for the use of this notation is given.

Our contributions in this regard are close to work at Ohio State University, where the Kronecker product is used to derive multiple recursive algorithms in various areas. Applications include an iterative version of Strassen's matrix multiplication algorithm [HJJ92,HJJ90], a systolic implementations of matrix computation problems[HL87], modeling multistage and direct interconnection networks [KSH$^+$92a,KSH$^+$92b], generating data distributions [KSH$^+$92c], large matrix transposition [KHJ$^+$93], and disk-based algorithms [KHJS92]. The main difference between our work and the EXTENT system developed in [DGK$^+$94] is that we provide flexibility and extensibility via symbolic manipulation of matrix expressions. The *EXTENT* system is limited in recognizing a very specific form of matrices, namely the product of matrices that are the Kronecker product of three matrices, two of which have to be the identity matrices.

1.5.3 Program Transformations and Code Generation

A high-level interpreted matrix language like MATLAB [Mat90] is an excellent tool for quick prototyping and experimentation in numerical programs, however when high performance is an issue, you have to resort to compiled code like FORTRAN, F90, C and C++. Through the use of interactive and automatic transformations in [DRGGP95], high-level matrix expressions written in MATLAB are translated into F90 with directives for parallelism. The compilation of array expressions for efficient code generation is also used in [GKHS93]. Highly desirable are environments that provide an interactive user interface and can handle well a specified class of problems like for example PDE iterative solvers in [CHR94].

Automatic FORTRAN code generation via symbol manipulation in Mathematica [Wol88] is used for handling the FFT's required for the direct solution of Poisson's equation[Bra91]. With the assistance of the symbolic algebra environment MACSYMA, Berman generated optimized FORTRAN code for FFT for vector lengths not equal to a power of two [Ber85].

To reduce the complexity involved in the creation of scientific code, a program transformation approach based upon the *Weyl* computer algebra substrate is used [Zip93]. However we envision that much more complex systems, like theorem provers, will be needed to handle very general problems, using a combination of automation via tactics and interactivity with the knowledgeable user to achieve highly optimized implementations of algorithms. Such a theorem prover is *NuPRL*, see [Jac95]. Nuprl and Weyl are ongoing projects at Cornell.

A technique to synthesize systematically parallel implementations of the FFT algorithm via program transformations, starting from a high-level specification in a functional language is presented in [SC93]. However, in their effort, they ignore the mathematical definition of the FFT and the corresponding sparse factorizations. Instead they use partial evaluation of recursive functional programs in order to generate larger radices and opportunities for parallelization. This results in more expensive code because the constant folding cannot compete with the formulae derived via the trigonometry and algebra that underlies the matrix factorizations.

A similar approach to ours is described in a sequence of publications [FCK95, BCFH92,FHB92] where they derive efficient implementations of numerical algorithms based on arbitrary matrix-vector products using program transformation of a functional language implementation of the algorithm. Our approach has a much more specific domain of applications, i.e, fast transforms. This permits greater depths in exploring possible implementations by exploiting the common structure of those transforms.

Chapter 2

Unconstrained Kronecker Product Approximation

Given a matrix A of dimension $m_A \times n_A$ with $m_A = m_B m_C$ and $n_A = n_B n_C$, we want to find matrices B and C of sizes $m_B \times n_B$ and $m_C \times n_C$ respectively that minimize the function:

$$f_A(B, C) = \|A - B \otimes C\|_F. \tag{2.1}$$

Here $\|\cdot\|_F$ denotes the Frobenius norm of a matrix:

$$\|A\|_F = \sqrt{\sum_i \sum_j a_{i,j}^2} = \sqrt{\operatorname{tr}(A^T A)}.$$

We consider as part of the input to this problem not only the matrix A but also the size of the unknown matrix B (sometimes there is flexibility in choosing the size of B and C). Note that any scaling of the matrix B with the respective inverse scaling of C has no effect on the value of the function, that is $f_A(B, C) = f_A(\alpha B, 1/\alpha C)$ for all $\alpha \neq 0$.

2.1 The Tilde Problem

Using a small example, let us see how the minimization of

$$f_A(B,C) = \|A - B \otimes C\|_F$$

can be transformed into a very familiar problem. Suppose A is 4×6, B is 2×3, and C is 2×2. The norm to minimize initially looks like this:

$$\left\| \left[\begin{array}{cc|cc|cc} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} \end{array} \right] - \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix} \otimes \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \right\|_F$$

Note that the displayed block structure of A has the same shape as B. What if we "stretch" matrix B by stringing its elements in column-major order using the vec operation. Since we have to preserve the correspondence of the elements, we have to reorder the blocks of A in exactly the same way:

$$\left\| \left[\begin{array}{cc} a_{11} & a_{12} \\ a_{21} & a_{22} \\ \hline a_{31} & a_{32} \\ a_{41} & a_{42} \\ \hline a_{13} & a_{14} \\ a_{23} & a_{24} \\ \hline a_{33} & a_{34} \\ a_{43} & a_{44} \\ \hline a_{15} & a_{16} \\ a_{25} & a_{26} \\ \hline a_{35} & a_{36} \\ a_{45} & a_{46} \end{array} \right] - \begin{bmatrix} b_{11} \\ b_{21} \\ b_{12} \\ b_{22} \\ b_{13} \\ b_{23} \end{bmatrix} \otimes \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \right\|_F$$

Now we transform C into a vector by taking $\text{vec}(C)^T$. In the same way we have to transform the blocks of A_{ij} into $\text{vec}(A_{ij})^T$ and we obtain:

$$\left\| \begin{bmatrix} a_{11} & a_{21} & a_{12} & a_{22} \\ a_{31} & a_{41} & a_{32} & a_{42} \\ a_{13} & a_{23} & a_{14} & a_{24} \\ a_{33} & a_{43} & a_{34} & a_{44} \\ a_{15} & a_{25} & a_{16} & a_{26} \\ a_{35} & a_{45} & a_{36} & a_{46} \end{bmatrix} - \begin{bmatrix} b_{11} \\ b_{21} \\ b_{12} \\ b_{22} \\ b_{13} \\ b_{23} \end{bmatrix} \otimes \begin{bmatrix} c_{11} & c_{21} & c_{12} & c_{22} \end{bmatrix} \right\|_F$$

Note that the Kronecker product of the two vectors is equivalent to their outer product.

In general, we have

Theorem 2.1.1 *If $B \in I\!R^{m_B \times n_B}$, $C \in I\!R^{m_C \times n_C}$, and*

$$A = \begin{bmatrix} A_{1,1} & \cdots & A_{1,n_B} \\ \vdots & & \vdots \\ A_{m_B,1} & \cdots & A_{m_B,n_B} \end{bmatrix}, \; A_{i,j} \in \mathbb{R}^{m_C \times n_C}$$

then

$$\|A - B \otimes C\|_F = \|\tilde{A} - bc^T\|_F$$

where $b = vec(B)$, $c = vec(c)$, and

$$\tilde{A} = \begin{bmatrix} \text{vec}(A_{1,1})^T \\ \vdots \\ \text{vec}(A_{m_B,1})^T \\ \vdots \\ \text{vec}(A_{1,n_B})^T \\ \vdots \\ \text{vec}(A_{m_B,n_B})^T \end{bmatrix} \in \mathbb{R}^{(m_B n_B) \times (m_C n_C)}.$$

Proof: See [VLP92] □

Thus the nearest Kronecker product problem is equivalent to a rank-1 matrix problem.

The following Matlab function forms matrix $\tilde{A}$:

```
function [T] = tilde(A,mb,nb)
% Set up the tilde matrix T from A subject to the dimension of B
% The dimensions of A must be divisible by the correponding
% dimensions of B

[m,n] = size(A);
mc = m / mb; nc = n / nb;
T = zeros(mb*nb,mc*nc);
x = zeros(1,mc*nc);       % as a block stretcher

for ib=1:mb
  for jb=1:nb
    x(:) = A((ib-1)*mc+1:ib*mc,(jb-1)*nc+1:jb*nc); % stretch block
    T((jb-1)*mb+ib,:) = x;
  end
end
return
```

Note that $A((i-1)*m_C+1 : i*m_C, (j-1)*n_C+1 : j*n_C)$ is the (i,j) block of A of size $m_C \times n_C$.

2.2 Three Methods

To solve the rank-1 approximation problem we need the following lemma:

Lemma 2.2.1 *If $A \in I\!R^{m\times n}$ then there exist orthogonal matrices $U \in I\!R^{m\times m}$ and $V \in I\!R^{n\times n}$ such that*

$$U^T AV = diag([\sigma_1, \sigma_2, \cdots, \sigma_p]) \in I\!R^{m\times n}$$

where $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p \geq 0$ and $p = \min(m,n)$. Moreover

$$\min_{x\in I\!R^m,\, y\in I\!R^n} \|A - xy^T\|_2 = \|A - \sigma_1 u_1 v_1^T\|_2 = \sigma_2$$

with u_1 and v_1 the first columns of matrices U and V.

Proof: See [GVL89], page 71. $\square$

Lemma 2.2.2 *If $A \in I\!R^{m \times n}$ then the minimizer of*

$$\min_{x \in I\!R^m, y \in I\!R^n} \|A - xy^T\|_2$$

is also the minimizer of

$$\min_{x \in I\!R^m, y \in I\!R^n} \|A - xy^T\|_F.$$

Proof: It is easy to see that

$$\|A - xy^T\|_F^2 = \operatorname{tr}\left((A - xy^T)^T(A - xy^T)\right) = \operatorname{tr}(A^T A) - 2y^T A^T x + x^T x y^T y. \quad (2.2)$$

With gradient equal to

$$\begin{bmatrix} -2Ay + 2y^T y x \\ -2A^T x + 2x^T x y \end{bmatrix} \quad (2.3)$$

any minimizer of (2.2) must be a nullifier of (2.3). Therefore x and y must be multiples of the singular vectors of A belonging to the same singular triplet $u_i^T A v_i = \sigma_i$, with u_i and v_i of unit length.

Let $x = \alpha u_i$ and $y = \beta v_i$, then expression (2.2) becomes

$$\|A - xy^T\|_F^2 = \operatorname{tr}(A^T A) - 2\alpha\beta\sigma_i + (\alpha\beta)^2.$$

Therefore the minimum is achieved when $i = 1$ and $\alpha\beta = \sigma_1$. $\square$

With the previous reduction and the help of the two lemmata, we can now state the following:

Theorem 2.2.3 *If the SVD of matrix $\tilde{A}$ is $\tilde{A} = U\Sigma V^T$ then the minimizers of the expression*

$$\|A - B \otimes C\|_F$$

are the matrices B^* and C^* with

$$vec(B^*) = \sigma_1 u_1,\ vec(C^*) = v_1$$

with u_1 and v_1 the first columns of matrices U and V respectively and σ_1 the largest singular value of $\tilde{A}$.

Proof: A direct application of Theorem (2.1.1) and Lemmata (2.2.1,2.2.2). $\square$

For an extensive presentation of the SVD, see [GVL89]. In the next subsections we show how to find the largest singular value and the corresponding singular vectors of a matrix using an explicit SVD method (§2.2.1), and the Lanczos method (§2.2.2). In §2.2.3, we present a new iterative method similar to the power method, that is also useful in proving the properties of the approximation.

2.2.1 Explicit SVD

As we have seen, the problem of finding the Kronecker product approximation to a matrix A, is equivalent to finding the largest singular value and the corresponding singular vectors of a scrambled version of A, we call $\tilde{A}$. When A is small, one can form explicitly $\tilde{A}$ using the MATLAB function `tilde` in §2.1.

With matrix $\tilde{A}$ formed explicitly, we can use the standard production SVD algorithms to get the factorization. The solution looks like this in MATLAB

```
function [B,C] = kpa_small(A,mb,nb)
% Find the Kronecker product approximation,
%       min(norm(A - kron(B,C),'fro'))
% A       : matrix to be approximated
% mb, nb : dimension of B
% The dimension of A must be divisible by
% the correponding dimension of B

[m,n] = size(A);
mc = m / mb; nc = n / nb;
```

```
[U,S,V] = svd(tilde(A,mb,nb));
B = zeros(mb,nb); C = zeros(mc,nc);

ss = sqrt(S(1,1));
B(:) = ss*U(:,1);
C(:) = ss*V(:,1);
return
```

We will refer to this method as the *direct SVD* method.

2.2.2 Lanczos

However when A is a large sparse matrix, finding the complete SVD of $\tilde{A}$ is prohibitively expensive due to fill-in. Besides, we only need the largest singular value and the corresponding left and right singular vectors. A more appropriate choice of an algorithm can be the Lanczos SVD [GLO81]. The Lanczos method for finding the largest singular value of a matrix $\tilde{A}$, first reduces this matrix into a bi-diagonal matrix that has the same singular values as the initial matrix. A simple version of Lanczos SVD algorithm in MATLAB is

```
function [sigma,u,v] = lanczos(A)
% lanczos method
% returns the largest singular value and its singular vectors
% u'*A*v = sigma

[m,n] = size(A);

transp = 0;
if m<n, transp = 1; A = A'; [m,n] = swap(m,n); end

V = randn(n,1); V = V/norm(V);
p = V(:,1); beta(0 +1) = 1; j = 0; U = zeros(m,1);

while beta(j +1) > 1e-12
  V(:,j+1) = p / beta(j +1);
  j = j+1;
  r = A * V(:,j) - beta(j-1 +1)*U(:,j-1 +1);
  alpha(j) = norm(r,2);
  U(:,j +1) = r/alpha(j);
  p = A'*U(:,j +1) - alpha(j)*V(:,j);
  beta(j +1) = norm(p,2);
end

B = diag(alpha);
for i=1:j-1
  B(i,i+1) = beta(i +1);
end

[Ub,S,Vb] = svd(B);

sigma = S(1,1);
u=U(:,2:j +1)*Ub(:,1);
v=V*Vb(:,1);

if transp == 1, [u,v] = swap(u,v); end

return
```

Note that Lanczos SVD only requires the provision of functions that perform the matrix-vector products $\tilde{A}x$ and $\tilde{A}^T x$ without the explicit formation of $\tilde{A}$, as it has been shown in [VLP92]. More specifically, here is MATLAB code that calculates $\tilde{A}x$ using a routine that calculates Ax.

```
function y = TildeTimes(AtimesX,x,mb,nb,mc,nc)
% calculate tilde(A,mb,nb) * x
% when you can only use A*x

y = zeros(mb*nb,1);
for i=1:nb
  rows = (i-1)*mb+1:i*mb;
  for j=1:nc
    Z = zeros(mc,mb);
    e = zeros(mb*mc,1); e((i-1)*nc+j) = 1;
    Z(:) = feval(AtimesX, e);
    y(rows) = y(rows) + Z'*x((j-1)*mc+1:j*mc);
  end
end
```

The corresponding $\tilde{A}^T x$ is

```
function y = TildeTranspTimes(AtimesX,x,mb,nb,mc,nc)
% calculate tilde(A,mb,nb)'* x
% when you can only use A*x

y = zeros(mc*nc,1);
for i=1:nc
  rows = (i-1)*mc+1:i*mc;
  for j=1:nb
    Z = zeros(mc,mb);
    e = zeros(mb*mc,1); e((j-1)*nc+i) = 1;
    Z(:) = feval(AtimesX, e);
    y(rows) = y(rows) + Z*x((j-1)*mb+1:j*mb);
  end
end
```

Therefore, the solution to our initial minimization problem is expressed via the following MATLAB function:

```
  function [B,C] = KA(Ax, ma, na, mb, nb)
  % Compute B (mb-by-nb) and C with compatible size so that
  % norm(A-kron(B,C),'fro') is minimum.
  % Matrix A is passed implicitly, via its size ma-by-na
  % and the function name Ax
  % to calculate A*x

  mc = ma/mb; nc = na/nb;
  B = zeros(mb,nb);  C = zeros(mc,nc);

  % Compute the largest singular triple using Lanczos
  [sigma,u,v] = lanczos(Ax,mb,nb,mc,nc);

  ss = sqrt(sigma);
  B(:) = ss*u;
  C(:) = ss*v;
  return
```

Variable `Ax` contains the string of the name of the function performing the matrix-vector product Ax that is passed to Lanczos together with the sizes $m_B n_B \times m_C n_C$.

Implementations of the Lanczos algorithm to solve large problems, must address the issue of loss of orthogonality of the Lanczos vectors. There are two different approaches to deal with the orthogonality problem. One approach attempts total reorthogonalization of each new Lanczos vector with respect to all previous ones, but it is very expensive numerically and it requires storage of all the Lanczos vectors. The second method is based on selective reorthogonalization. For a Lanczos implementation with selective reorthogonalization, see [Ber80].

2.2.3 Power Method

As we have already seen, the problem of finding the Kronecker product approximation to a matrix is equivalent to finding the rank-1 approximation to the tilde matrix. We give an alternate way to find such approximation. Let us first consider an easier problem. Suppose we are given two vectors a and v and we want to scale v by σ so that the length of $a - \sigma v$ is minimum, or written as a function, try to minimize $f(\sigma) = \|a - \sigma v\|_2$. As it can be seen in Figure 2.1, the solution is to scale v so as it becomes equal to the length of the projection of a onto v. Since $a^T v$ is the product of the length of v times the length of the projection of a onto the direction of v, we need to divide the quantity $a^T v$ by the square of the length of v. Therefore the optimal is achieved when

$$\sigma = \frac{a^T v}{v^T v}. \tag{2.4}$$

The same formula can be derived via calculus, without resorting to geometry. The minimizer of $\|a - \sigma v\|_2$ is equal to the minimizer of

$$f(\sigma) = \|a - \sigma v\|_2^2 = (a - \sigma v)^T (a - \sigma v)$$

which has gradient

$$g(\sigma) = 2\sigma v^T v - 2v^T a.$$

The nullifier of g is indeed the value given in (2.4).

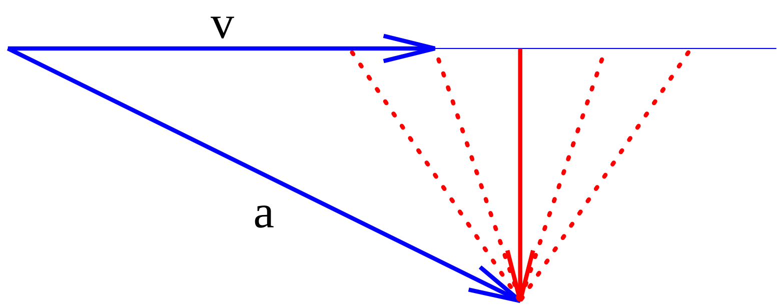

Figure 2.1: Solving graphically $\min \|a - \sigma v\|_2$ for given vectors a and v

Suppose now that we are given matrix $A \in \mathbb{R}^{m \times n}$, and we seek two vectors $u \in \mathbb{R}^m$ and $v \in R^n$ that minimize the expression

$$\|A - uv^T\|_F. \tag{2.5}$$

If we make a guess for v, we get m independent $\|a_i - \sigma_i v\|_2$ problems to solve, where a_i is the i^{th} row of A. Then use the σ's as u and improve v. This "Flip-Flop" leads us to the following algorithm:

```
function [u,sigma,v] = VecFlipFlop(A,v0)
% find the largest singular value sigma and corresponding
% left and right singular vectors u, v
% of matrix A
% v0 is an optional initial guess for v
epsilon = 1e-8;
[m,n] = size(A);
if nargin==2
  v = v0;
else
  v = randn(n,1);
end

done = 0;
while ~done
  u = A*v; u = u / norm(u);
  v = A'*u;
  sigma = v'*v; v = v/sigma;
  done = abs(u'*A*v - sigma) <= epsilon;
end
```

The above iteration is a variant of the *power method*. It will converge to the eigenvector corresponding to the largest eigenvalue of $A^T A$. The rate of convergence depends upon the ratio $|\lambda_2|/|\lambda_1|$ of the largest two eigenvalues of $A^T A$, the smaller the ratio the faster the convergence. The method will fail to converge when the ratio is close to one. These subtleties are discussed by J. H. Wilkinson in [Wil88]. The advantage of the power method is that it only requires two functions that provide the matrix-vector products Av and $A^T u$ for arbitrary vectors v and u without explicitly forming and storing matrix A. Moreover, the method can take advantage of an *à priori* estimate of vector v and reduce the iterations required to converge. When no such estimate is known, the vector of ones should be used as the initial guess for $v^{(0)}$, this will result to $u^{(1)} = Av^{(0)}$ to be a scaled average of the columns of A.

In implementing this algorithm we must be careful in recognizing convergence. To avoid the comparison of the residuals of two consecutive iterations that is expensive, we can assume that v must be of unit length, and thus check the norm of change of v .

```
function [u,v] = flip_flop(fun_tAx, fun_tATx, m, n, epsilon, v0)
% Rank-1 approximation of matrix A using the power method.
% Compute vectors u and v of lengths m and n respectively
% that minimize norm(A-u*v','fro').
% Matrix A is passed implicitly, via its size m-by-n
% and the function names fun_tAx and fun_tATx that
% calculate tilde(A)*x and tilde(A)'*x
% epsilon : termination criterion
% v0      : optional initial value for v
% if there is an initial value
if nargin==6
```

```
  v = v0;
else
  v = ones(n,1);
end

v = v/sqrt(v'*v); % normalize v
converge = 0; iter = 0;
while ~converge & iter <= 500
  iter = iter + 1;
  pv = v;
  v = eval([fun_tATx '(' fun_tAx '(v))']);     % v = A'*(A*v);
  v = v/sqrt(v'*v);                            % normalize v
  converge = (norm(pv - v) < epsilon);
end
u = eval([fun_tAx '(v)']);                     % u = A*v;
return
end
```

The work required is $O(mn)$ per iteration, if we assume that matrix A is dense.

As an aside we can note that if we define as

$$f_A(u,v) = \|A - uv^T\|_F^2 = \operatorname{tr}((A - uv^T)^T(A - uv^T))$$

then the gradient of f is,

$$g(u,v) = \nabla f = \begin{bmatrix} -2Av + 2v^Tvu \\ -2A^Tu + 2u^Tuv \end{bmatrix} \tag{2.6}$$

therefore the fixed point of Algorithm 0 is indeed the minimizer of $f_A(u,v)$.

We can now substitute vectors u and v in *Algorithm 0* with the vector-matrices vec(B) and vec(C) and matrix A with the transformed matrix $\tilde{A}$ and get the following algorithm expressed in MATLAB

```
function [B,C,iterations] = KPA(A,mb,nb,epsilon);
% kronecker product matrix approximation
%
% A       : the array to be approximated
% mb,nb   : the size of B
% epsilon : termination criterion
% returns arrays B,C such that norm(A - kron(B,C),'fro') is minimum

[ma,na] = size(A);
mc = ma/mb; nc = na/nb;
C = randn(mc,nc); B = randn(mb,nb);
conv = 0;
iterations = 0;

while ~conv
  iterations = iterations + 1;
  X = [];
  for i = 1:mb
    for j = 1:nb
      X(i,j) = trace(A((i-1)*mc+1:i*mc,(j-1)*nc+1:j*nc)'*C);
    end
  end
  X = X/trace(C'*C);

  nr1 = norm(X-B,'fro');
  conv = (nr1 <= epsilon);
  B = X;

  X = [];
  for i = 1:mc
    for j = 1:nc
      X(i,j) = trace(A(([1:mb]-1)*mc+i,([1:nb]-1)*nc+j)'*B);
    end
  end
  X = X/trace(B'*B);

  nr2 = norm(X-C,'fro');

  conv = conv & (nr2 <= epsilon);
  C = X;
end

return
```

This algorithm solves the nearest Kronecker product since in [VLP92] it was shown that the minimizer of

$$f_{A,C}(B) = \|A - B \otimes C\|_F$$

satisfies

$$b_{ij} = \frac{\mathrm{tr}(A_{ij}^T C)}{\mathrm{tr}(C^T C)}.$$

Likewise, the minimizer of

$$f_{A,B}(C) = \|A - B \otimes C\|_F$$

is satisfying

$$c_{ij} = \frac{\mathrm{tr}(\hat{A}_{ij}^T B)}{\mathrm{tr}(B^T B)}$$

where A is a block matrix as in Theorem 2.1.1 and $\hat{A}_{ij}$ is the matrix that consists of the (i, j) elements of all the blocks of A.

2.3 Properties and Perturbation

We state for reference some important attributes of the Kronecker product approximant.

2.3.1 Inheritance of Structure

The Kronecker product preserves properties of the original matrices in most cases, as we have seen in Chapter 1. It is of great interest to see whether the Kronecker product approximation preserves the properties of the approximated matrix. In [VLP92] it is shown that the Kronecker product approximation matrices of a symmetric positive definite matrix are indeed symmetric positive definite. Similarly for banded matrices and non-negative matrices.

However, the Kronecker product approximation matrices of a symmetric matrix are not necessarily symmetric (they can be skew symmetric). For example consider the following symmetric matrix that is the Kronecker product of a skew symmetric matrix

$$A = \left(\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \otimes \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \right) = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

The Kronecker product approximation matrices of a stochastic matrix (a square non-negative matrix with column-sums equal to one) are not stochastic, as it can be seen from the counter example

$$A = \begin{bmatrix} 0.1 & 0.0 & 0.6 & 0.2 \\ 0.2 & 0.2 & 0.1 & 0.1 \\ 0.3 & 0.3 & 0.2 & 0.4 \\ 0.4 & 0.5 & 0.1 & 0.3 \end{bmatrix}$$

and the closest pair

$$\begin{bmatrix} 0.2367 & 0.5167 \\ 0.7159 & 0.4861 \end{bmatrix}, \begin{bmatrix} 0.6024 & 0.4782 \\ 0.4049 & 0.5624 \end{bmatrix}.$$

The Kronecker product approximation matrices of an orthogonal matrix are not necessarily orthogonal. We will show how to approximate a matrix with a pair of matrices with specific structure, or properties in the next chapter.

2.3.2 Perturbation Theory

What happens to the Kronecker approximation when we perturb the input matrix? Suppose we change A into $A + E$ where $\|E\|_F$ is very small, how do B^* and C^*, the minimizers of $\|A - B \otimes C\|_F$ compare to $\bar{B}^*$ and $\bar{C}^*$ the minimizers of $\|A + E - B \otimes C\|_F$? We study the effect of the perturbation on the equivalent rank-1 approximation problem because there has been an extensive study of the effects of perturbation on the singular value decomposition of a matrix. It is valid to do so because the perturbation is analogous in the tilde space,

$$(A \tilde{+} E) = \tilde{A} + \tilde{E},$$

that means that a small change in A corresponds to a small change in $\tilde{A}$. We need the following two lemmata from [GVL89]

Lemma 2.3.1 *For all matrices A and E it holds*

$$|\sigma_1(A+E) - \sigma_1(A)| \leq \sigma_1(E) = \|E\|_2.$$

and

Lemma 2.3.2 *Let matrices A and E of the same size and the singular value decomposition of A:*

$$A = U\Sigma V^T \textit{ with } U = [u_1, U_2] \textit{ and } V = [v_1, V_2].$$

Let also

$$U^TAV = \begin{bmatrix} \sigma_1 & 0 \\ 0 & \Sigma_2 \end{bmatrix},$$

$$U^TEV = \begin{bmatrix} \epsilon_{11} & e_{12}^T \\ e_{21} & E_{22} \end{bmatrix}$$

so that we can define

$$0 < \delta = \sigma_1 - \sigma_2 - |\epsilon_{11}| - \|E_{22}\|_2 \textit{ and } \epsilon = \|[\epsilon_{11},\ e_{12}^T]\|_2.$$

If $\frac{\epsilon}{\delta} \leq \frac{1}{2}$ then there exist vectors p and q satisfying

$$\|[p^T,\ q^T]\|_2 \leq 2\frac{\epsilon}{\delta}$$

such that Range$(u_1 + U_2p)$ and Range$(v_1 + V_2q)$ are a pair of singular subspaces for $A+E$.

Recall subspaces $\mathcal{S}$ and $\mathcal{T}$ form a *singular subspace pair* for matrix A if

$$x \in \mathcal{S} \Rightarrow Ax \in \mathcal{T},\; y \in \mathcal{T} \Rightarrow A^T y \in \mathcal{S}.$$

Now we can state

Theorem 2.3.3 *If B^* and C^* are the minimizers of*

$$\|A - B \otimes C\|_F$$

and $\bar{B}^*$ and $\bar{C}^*$ are the minimizers of

$$\|(A+E) - B \otimes C\|_F$$

then

$$\|B^* \otimes C^* - \bar{B}^* \otimes \bar{C}^*\|_F^2 \;\leq\; 2\|A\|_2 \left(1 - \frac{1}{1 + 4\frac{\epsilon^2}{\delta^2}}\right)(\|A\|_2 + \|E\|_2) + \|E\|_2^2$$

where δ and ϵ are defined in lemma 2.3.2

Proof: Let the singular value decomposition of $\tilde{A}$ be $\tilde{A} = U\Sigma V^T$, with $\Sigma = \mathrm{diag}(\sigma_i)$ the singular values of $\tilde{A}$. We know that $\mathrm{vec}(B^*) = \sigma * u_1$ and $\mathrm{vec}(C^*) = v_1$, where σ the largest singular value. Let also be τ the largest singular value of $(A \tilde{+} E)$ and x and y the corresponing singular vectors. However, there exist vectors p and q such that $x \in \mathrm{Range}(u_1 + U_2 p)$ and $y \in \mathrm{Range}(v_1 + V_2 q)$ with U_2 and V_2 the rest of the columns of U and V, $\|[p^T q^T]\|_2 \leq 2\frac{\epsilon}{\delta}$ and δ, ϵ determined by lemma 2.3.1. But since $\mathrm{Range}(u_1 + U_2 p)$ and $\mathrm{Range}(v_1 + V_2 q)$ are one-dimensional spaces, we can consider

$$x = \frac{u_1 + U_2 p}{\|u_1 + U_2 p\|_2},\; y = \pm\frac{v_1 + V_2 q}{\|v_1 + V_2 q\|_2}$$

or simpler

$$x = \frac{u_1 + U_2 p}{\sqrt{1 + p^T p}},\; y = \pm\frac{v_1 + V_2 q}{\sqrt{1 + q^T q}}.$$

Then

$$\|B^* \otimes C^* - \bar{B}^* \otimes \bar{C}^*\|_F^2 = \|\sigma u_1 v_1^T - \tau x y^T\|_F^2 =$$

$$\sigma^2 + \tau^2 - 2\sigma\tau(u_1^T x)(v_1^T y) = \sigma^2 + \tau^2 - 2\sigma\tau\frac{1}{\sqrt{1+p^T p}\sqrt{1+q^T q}}$$

$$\leq \sigma^2 + \tau^2 - 2\sigma\tau\frac{1}{1+p^T p + q^T q}.$$

But we know that

$$\sqrt{p^T p + q^T q} \leq 2\frac{\epsilon}{\delta}$$

from lemma 2.3.2 therefore the last quantity is

$$\leq \sigma^2 + \tau^2 - 2\sigma\tau\frac{1}{1+4\frac{\epsilon^2}{\delta^2}}.$$

Also from the fact that $(\sigma - \tau)^2 \leq \|E\|_2^2$ from lemma 2.3.1 we deduce that for a given σ

$$\sigma - \|E\|_2 \leq \tau \leq \sigma + \|E\|_2.$$

Since the expression we are studying defines a parabola with upward concavity, its maximum, and therefore an upper bound to the range of its values is achieved at one of the ends of the interval, that after comparing them we find it is less than or equal to

$$2\sigma\left(1 - \frac{1}{1+4\frac{\epsilon^2}{\delta^2}}\right)(\sigma + \|E\|_2) + \|E\|_2^2.$$

□

From this theorem we see that the Kronecker product approximation is sensitive to perturbations that can change the singular subspaces of the largest singular value of $\tilde{A}$. This situation can arise when $\tilde{A}$ has the two largest singular values close together. However, when the largest singular value is isolated, any small perturbation on A will induce very small change in the Kronecker product approximation factors because the quantity

$$1 - \frac{1}{1 + 4\frac{\epsilon^2}{\delta^2}}$$

will be close to zero and therefore the change in the approximation factors will be bounded

$$\|B^* \otimes C^* - \bar{B}^* \otimes \bar{C}^*\|_F^2 \leq \|E\|_2^2.$$

2.4 Some Extensions

We show that any block matrix can be written as a finite sum of Kronecker products and how to compute the nearest strong Kronecker product to a given block matrix.

2.4.1 Nearest Sum of r Kronecker Products

The nearest Kronecker product to a matrix A corresponds to the nearest rank-one matrix of the scrambled matrix $\tilde{A}$. We can express $\tilde{A}$ as a summation of rank-one matrices defined by its singular value decomposition,

$$\tilde{A} = U\Sigma V^T = \sum_{i=1}^{r} \sigma_i u_i v_i^T$$

where u_i and v_i are the colums of orthogonal matrices U and V, Σ is a diagonal matrix with elements $\sigma_i \geq 0$ in descending order, and r is the rank of matrix $\tilde{A}$. In the unscrambled space, this fact means that any matrix A with composite (not prime) dimensions can be written as a summation of exactly r Kronecker products

$$A = \sum_{i=1}^{r} (B_i \otimes C_i)$$

with $\text{vec}(B_i) = \sigma_i u_i$ and $\text{vec}(C_i) = v_i$. Thus, if we wanted to approximate A with a sum of $k \leq r$ Kronecker products, then we simply take the first k terms of this summation.

We study the special case of approximating the summation $(I \otimes F) + (G \otimes I)$ in the next section. Sums of Kronecker products arise in queuing theory, see [Kau83].

2.4.2 The Strong Kronecker Product

The *strong Kronecker product* is defined in the area of combinatorial theory as a multiplication tool for orthogonal matrices [DLS94]. Given two matrices B and C written as n_1-by-n_2 and n_2-by-n_3 block matrices respectively, the n_1-by-n_3 block matrix A is said to be the strong Kronecker product of B and C if and only if, each block of A can be written as

$$A_{ij} = \sum_{k=1}^{n_2} (B_{ik} \otimes C_{kj}).$$

We denote the strong kronecker product of B and C with $B \circ C$. The operation is well defined only after the parameters n_1, n_2 and n_3 are set. The strong Kronecker product preserves orthogonality and is associative.

Now, consider matrix $\hat{A}$, a reordering of the elements of matrix A where each A_{ij} block has been substituted with the $\tilde{A}_{ij}$. From the definition of strong Kronecker product, we know that matrix $\tilde{A}_{ij}$ is of rank n_2, let us write it as

$$\tilde{A}_{ij} = X_i Y_j^T$$

where matrices X and Y contain n_2 orthogonal columns, the number of rows are equal to the block sizes of matrices B and C respectively. That means $\hat{A} = XY^T$ is a rank-n_2 matrix, the product of n_1-by-1 block matrix X and n_3-by-1 block matrix Y. Such a matrix corresponds to a sum of n_2 Kronecker products.

Let us clarify the above discussion with an example. Suppose B is an 2-by-3 block matrix with m_B-by-n_B blocks B_{ij} and C is an 3-by-4 block matrix with m_C-by-n_C blocks C_{ij} and $A = B \circ C$. That is

$$B = \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \end{bmatrix} \text{ and } C = \begin{bmatrix} C_{11} & C_{12} & C_{13} & C_{14} \\ C_{21} & C_{22} & C_{23} & C_{24} \\ C_{31} & C_{32} & C_{33} & C_{34} \end{bmatrix}$$

therefore A is a 2-by-4 block matrix with $m_B m_C$-by-$n_B n_C$ blocks A_{ij} with

$$A_{ij} = B_{i1} \otimes C_{1j} + B_{i2} \otimes C_{2j} + B_{i3} \otimes C_{3j}.$$

From the above we deduce that $\tilde{A}_{ij} = X_i Y_j^T$ for some matrices X_i and Y_j with 3 columns. If we consider

$$\hat{A} = \begin{bmatrix} \tilde{A}_{11} & \tilde{A}_{12} & \tilde{A}_{13} & \tilde{A}_{14} \\ \tilde{A}_{21} & \tilde{A}_{22} & \tilde{A}_{23} & \tilde{A}_{24} \end{bmatrix}$$

then obviously

$$\hat{A} = \begin{bmatrix} X_1 \\ X_2 \end{bmatrix} \begin{bmatrix} Y_1^T & Y_2^T & Y_3^T & Y_4^T \end{bmatrix} = XY^T$$

that is a rank-3 matrix, or equivalently that A can be obtained by reordering the sum of 3 Kronecker products.

We formalize the above discussion with the following

Theorem 2.4.1 *Let A be a n_1-by-n_3 block matrix with blocks A_{ij}, B be a n_1-by-n_2 block matrix, and C be a n_2-by-n_3 block matrix. We denote vectors $b_{ij} = vec(B_{ij})$ and $c_{ij} = vec(C_{ij})$. Then the following holds:*

$$\|A - B \circ C\|_F^2 = \sum_{i,j} \|\tilde{A}_{ij} - \sum_{k=1}^{n_2} b_{ik} c_{kj}^T\|_F^2$$

Proof:

$$\|A - B \circ C\|_F^2 =$$

$$\sum_{i,j} \|A_{ij} - \sum_{k=1}^{n_2} B_{ik} C_{kj}^T\|_F^2 =$$

$$\sum_{i,j} \|\tilde{A}_{ij} - \sum_{k=1}^{n_2} b_{ik} c_{kj}^T\|_F^2$$

because of Theorems 2.1.1 and 2.4.1. □

To find an optimal pair of n_1-by-n_2 block matrix B and n_2-by-n_3 block matrix C, that minimize

$$\min_{B,C} \|A - B \circ C\|_F$$

for a given n_1-by-n_3 block matrix A we:

- Form matrix $\hat{A}$, a reordering of matrix A where each block A_{ij} is substituted by $\tilde{A}_{ij}$.
- Compute the SVD of $\hat{A}$.
- Use the first n_2 terms of the SVD expansion to form B and C according to Theorem 2.4.1.

In Appendix A we provide MATLAB code that calculates the strong Kronecker product as well as approximations with it.

2.5 Preconditioning Applications

We will use the Kronecker product approximation as a preconditioner for the conjugate gradient (CG) method to solve the linear systems of equations that arise from the discretization of partial differential equations. Standard finite-difference discretizations yield certain symmetric positive definite block tridiagonal linear systems of the form $Ax = b$. In general, matrix A can be written as

$$A = I_{n_1} \otimes (a_1 I_{n_2} + a_2 J_{n_2}) + J_{n_1} \otimes (a_2 I_{n_2} + a_3 J_{n_2}) \tag{2.7}$$

where I_n is the identity matrix of size n by n and J_n the n by n symmetric tridiagonal with zeros on the diagonal and ones on the first super and sub-diagonal.

We say that matrix A corresponds to stencil

$$\begin{array}{|c|c|c|}\hline a_3 & a_2 & a_3 \\ \hline a_2 & a_1 & a_2 \\ \hline a_3 & a_2 & a_3 \\ \hline \end{array} . \tag{2.8}$$

For example, the Dirichlet stencil has coefficients $a_1 = 4$, $a_2 = -1$ and $a_3 = 0$ while the Laplace stencil has $a_1 = 20$, $a_2 = -4$ and $a_3 = -1$ from computer vision [Hor86].

The Dirichlet matrix is used as the model problem for the comparison of different preconditioners in [CGM85]. The preconditioning matrix M, should be an approximation to matrix A that is easier to solve with. It is easy to show that if M is the Kronecker product approximation of a stencil matrix, then $M = B \otimes C$ where the two factors B and C are symmetric Toeplitz tridiagonal matrices of the form $B = b_1 I_{n_1} + b_2 J_{n_1}$, $C = c_1 I_{n_2} + c_2 J_{n_2}$. The coefficients b_1, b_2, c_1 and c_2 can be found explicitly.

It is known that the conjugate gradient method converges rapidly if the condition number $\kappa(M^{-1}A)$ is small or if the eigenvalues are clustered. In Figure 2.2 we see the clustering of the eigenvalues of $M^{-1}A$ around 1, although it is not definitive enough to suggest a proof about the convergence of the preconditioned conjugate gradient method. We also compare the residuals at each iteration of the Kronecker product approximation preconditioner, versus the residuals of the diagonal and the incomplete Cholesky (IC) preconditioners in figure 2.3. We observe that Kronecker product approximation is much better than the diagonal preconditioner but not as good as IC. The table for the iterations needed to achieve residual r such that $r^T Ar \leq 10^{-6}$ for the Kronecker product approximation and the IC preconditioners leaves the same impressions.

The bottleneck for each iteration in preconditioned CG, is solving the linear

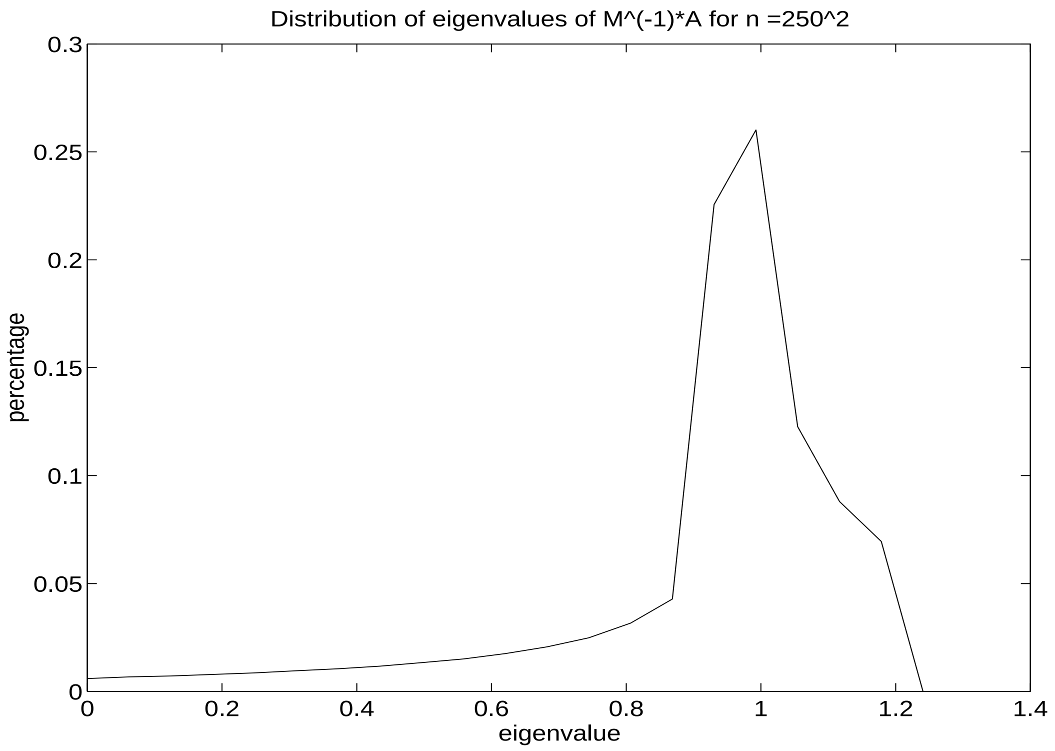

Figure 2.2: The distribution of the eigenvalues of $M^{-1}A$ for $n = n_1 n_2 = 62500$ with $n_1 = n_2 = 250$ shows the clustering of eigenvalues around 1.0.

Table 2.1: Comparison of iterations for the Kronecker product approximation and the IC preconditioners.

Problem Size	IC Iterations	KPA Iterations
4^2	7	7
8^2	9	11
16^2	17	19
32^2	23	33
64^2	39	56
128^2	51	74
256^2	66	93

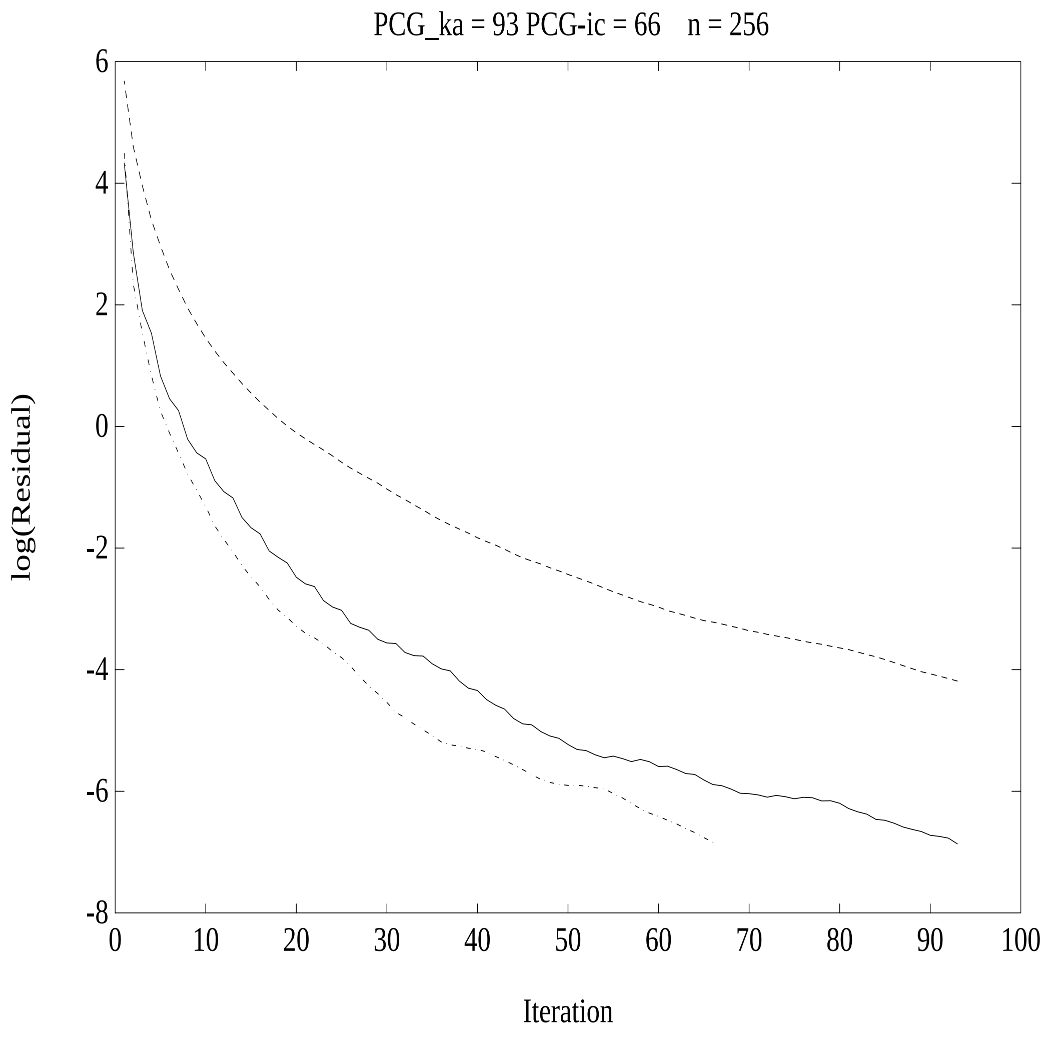

Figure 2.3: A comparison of the reduction of the residual for three different preconditioners. The dashed line is for the diagonal preconditioner, the solid for the Kronecker product approximation and the dash-dotted line for the incomplete Cholesky. The size of A is $n = n_1 n_2 = 65536$ with $n_1 = n_2 = 256$.

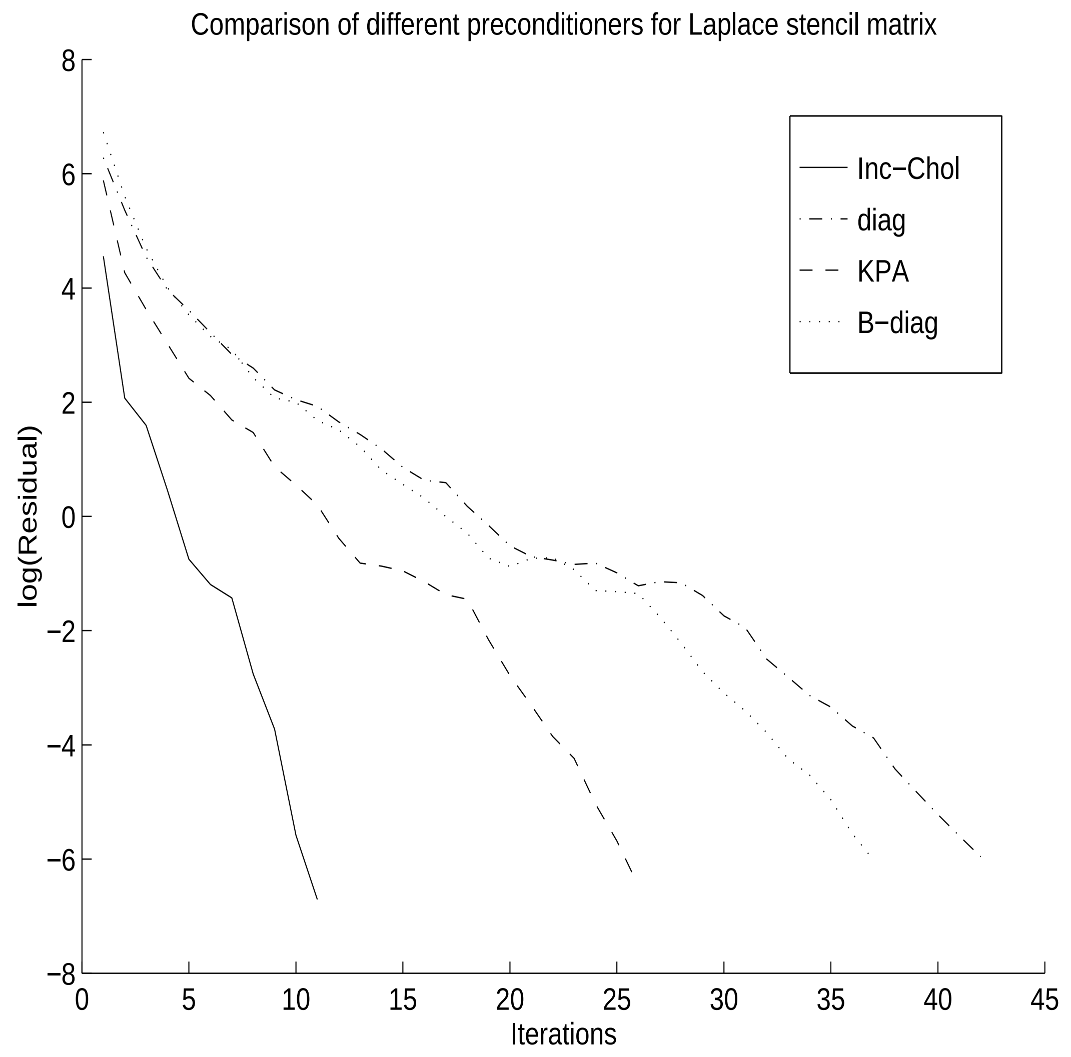

Figure 2.4: A comparison of the reduction of the residual for four different preconditioners. The solid line corresponds to Incomplete Cholesky, the dashed line is for the Kronecker product approximation, the dotted for the block-diagonal preconditioner and the dash-dotted line for the diagonal preconditioner respectively. The size of the Laplace stencil matrix A is $n = n_1 n_2 = 400$ with $n_1 = n_2 = 20$.

system $Mz = r$ for z. Using the IC as preconditioner for the model problem, solving $Mz = r$ corresponds to solving the system $GG^Tz = r$ where G is the incomplete Cholesky decomposition of A. In contrast, using the Kronecker product approximation as a preconditioner, a Kronecker product has to be solved at each iteration. That is $Mz = r$ corresponds to $(B \otimes C)z = r$ or equivalently to solving $CZB^T = R$ for Z, where Z is a n_2-by-n_1 matrix such that $\text{vec}(Z) = z$ and R is a m_2-by-m_1 matrix such that $\text{vec}(R) = r$.

For the model problem of size n and $m_1 \approx n_1 \approx \sqrt{n}$, it can be shown that G is banded with only 3 sub-diagonals non zero and solving the system requires $9n$ flops. For the Kronecker product approximation preconditioner, B and C are tridiagonal and $10n$ flops are required, except that a large percentage of those flops can be executed in parallel, because we solve for multiple right hand sides.

The CM-5 experience

The Connection Machine Scientific Software Library (CMSSL) [Thi93] provides a toolkit for iterative methods together with customized solvers when the linear systems are structured. The most common operation for the CG is the matrix-vector product Ax. Since A in the model problem is a stencil, we distributed x on a n_2 by n_1 grid so that the application Ax requires only nearest neighbor communication. This is the corresponding Connection Machine Fortran (CMF) [Thi91] routine that calculates $y = Ax$:

```
      subroutine dirichletnn(n,x,y)
      integer n,i,j
      real x(n,n), y(n,n)
cmf$layout x(:news,:news)
cmf$layout y(:news,:news)

      y = 4*x - (eoshift(x,1, 1,0.0) +
     &           eoshift(x,1,-1,0.0) +
     &           eoshift(x,2, 1,0.0) +
     &           eoshift(x,2,-1,0.0))

      return
      end
```

For solving $CZB^T = R$ for Z, we take the LDLT decomposition of the symmetric tridiagonal matrices B and C that are $B = L_B D_B L_B^T$ and $C = L_C D_C L_C^T$ respectively. Back and forward substitution are used for solving the unit triangular (bi-diagonal) systems. Here is the subroutine that solves $LDL^T Z = R$ for Z. The right hand side R is overwritten by Z. Matrix D is represented only by its diagonal `d` while unit lower bi-diagonal L is represented by its first sub-diagonal `c`. The rhs is `r`.

```
c solve using LDL'
      subroutine tridiag_sym_solve(n,d,c,r)
      implicit none
      integer n
      real c(n-1),d(n)
      real r(n,n)

      integer i,j

cmf$layout d(:news)
cmf$align c(i) with d(i)
cmf$layout r(:news,:news)

      do j=1,n-1
        r(j+1,:) = r(j+1,:) - c(j) * r(j,:)
      end do

      do i=1,n
        r(i,:) = r(i,:) / d(i)
      end do

      do i=n,2,-1
        r(i-1,:) = r(i-1,:) - c(i-1) * r(i,:)
      end do
```

```
    return
    end
```

Implementing the IC preconditioner in a parallel environment is a very difficult task, as it is demonstrated in [GKS92]. Since there was no good implementation for the IC preconditioner, we compared the Kronecker product approximation with the diagonal and the block diagonal preconditioners using the implementations found in CMSSL. The speed of CM-5 allowed us to solve very large instances of the model problem. The Kronecker product approximation preconditioner needed fewer iterations than the other preconditioners, but it took more time due to communication overhead. These results from a 32-node CM5 with vector units can be seen in figure 2.5. We observe that the simplest preconditioner is faster than the more complex ones, despite the greater number of iterations needed. This is due to computation and communication costs of the complex preconditioners.

2.6 Open Issues

As we have seen, although the Kronecker product approximation requires fewer iterations as a preconditioner, for the conjugate gradient to converge, it required more time than the simpler diagonal preconditioner. It is therefore of interest to study data alignment issues that are favorable in solving $CZB^T = R$, that is the linear system solved at each iteration of the conjugate gradient.

Another issue is the implementation of the Kronecker product approximation algorithm on a parallel computer. The separable least squares approach for calculating the Kronecker product approximation is especially attractive for implementation on p processors connected in a mesh. For simplicity assume that $m_1 = n_1 = m_2 = n_2 = \sqrt{p}$ and the processors are arranged as a $\sqrt{p}$-by-$\sqrt{p}$ array

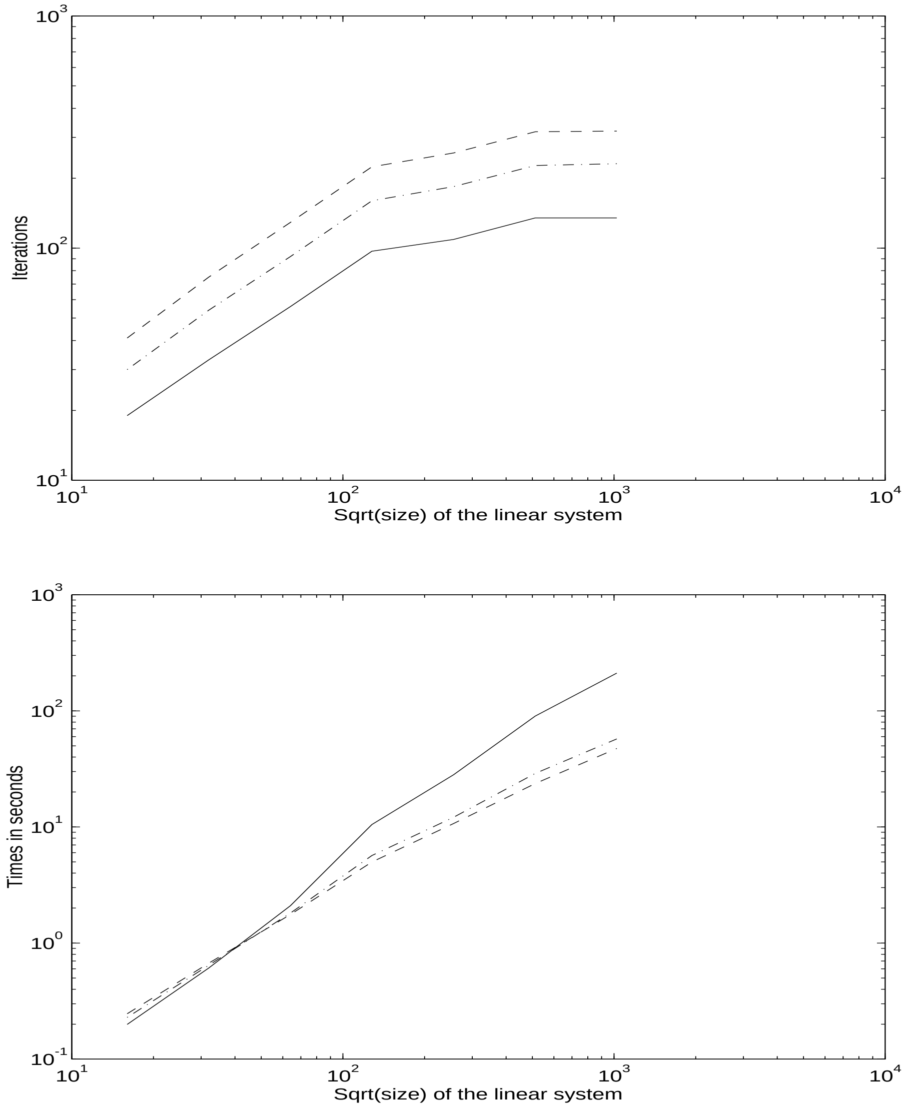

Figure 2.5: A comparison of the iterations and time to converge for three different preconditioners on the CM-5 with 32 nodes. The dashed line is for the diagonal preconditioner, the solid for the Kronecker product approximation and the dash-dotted line for the block diagonal.

with connections that allow communications to be completed in at most $\mathrm{O}(\log p)$ messages. Processor ij will calculate the elements b_{ij} and c_{ij} and has in its local memory the blocks A_{ij} and $\hat{A}_{ij}$ together with a copy of the current B and C matrices. An iteration of the separable least squares approach can be done as follows:

1 All processors collectively calculate $\mathrm{tr}(C^TC)$ and propagate the current C matrix in $\mathrm{O}(\log p)$ time steps.

2 Each processor in parallel calculates its own b_{ij} in $\mathrm{O}(p)$ time steps.

3 All processors collectively calculate $\mathrm{tr}(B^TB)$ and propagate the current B matrix in $\mathrm{O}(\log p)$ time steps.

4 Each processor in parallel calculates its own c_{ij} in $\mathrm{O}(p)$ time steps.

There is the promise of the Kronecker product approximation to be a good preconditioner for the stencil matrices for 3-D cases. Figure 2.6 shows that our preconditioner provides a very competitive picture with respect to the preconditioner based on the incomplete Choleski factorization. Of course there are many issues that need research here, like the way to choose the size of the approximation, how to handle the lower dimensions, the issue of the stride and so on.

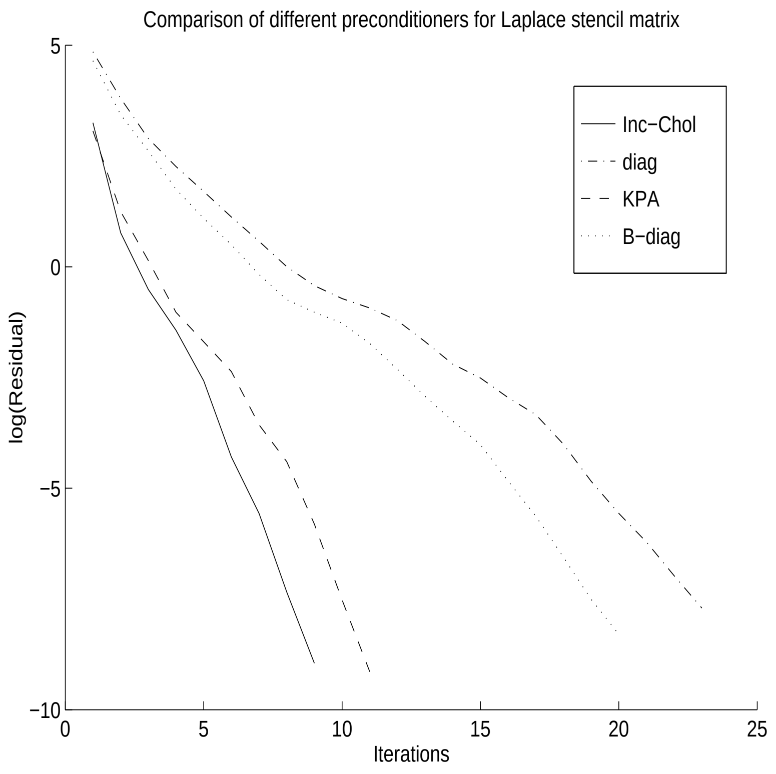

Figure 2.6: A comparison of the reduction of the residual for four different preconditioners. The solid line corresponds to Incomplete Choleski, the dashed line is for the Kronecker product approximation, the dotted for the block-diagonal preconditioner and the dash-dotted line for the diagonal preconditioner respectively. The size of the 3 dimensional square Laplace stencil matrix A is $n = n_1 n_2 n_3 = 343$ with $n_1 = n_2 = n_3 = 7$.

Chapter 3

Constrained and Modified Problems

In this chapter we study the constrained Kronecker product approximation of finding two matrices B and C of predetermined sizes that their Kronecker product is closest to a given matrix A. We require the matrices B and C to have a special structure, like being symmetric or Toeplitz, or being Markov or orthogonal. We also present some indicative applications of such approximations.

The idea of a constrained approximation of a matrix with another more structured matrix is not new. It is a recent idea to approximate a Toeplitz matrix with a circulant matrix in order to reduce computational cost of solving a linear system when the matrix is Toeplitz [Nag95b,Nag95a]. The use of such an approximation as a preconditioner for the conjugate gradient method was first proposed by Strang [Str86]. Chan proposed as a preconditioner of Toeplitz systems, the closest circulant to the Toeplitz matrix in the Frobenius norm [Cha88] and many more followed. It is therefore, only natural for us to show how to expand the

Kronecker product approximation with constraints so that the factors have special properties. Actually, as it turns out, the structural constraints (homogeneous linear constraints), when we require certain matrix elements to be equal or zero, for example, for symmetric, or banded, or Toeplitz, or circulant matrices, are very easy to solve. For non-homogeneous linear constraints and non-linear constraints, the problem is difficult to solve, however we do provide algorithms to get a solution.

Consider the constrained Kronecker product approximation problem

$$\min_{G(B)\leq g, H(C)\leq h} \|A - B \otimes C\|_F \tag{3.1}$$

where $G(B) \leq g$ and $H(C) \leq h$ are constraints involving the unknown matrices B and C. In the following subsections we will show how to handle linear equality constraints and some interesting properties of the constrained matrices that allow us to solve some difficult constrained problems.

3.1 Linear Equality Constraints

When the constraints are linear equalities, the constrained optimization problem can be written as:

$$\min_{S_B^T \mathrm{vec}(B)=g, S_C^T \mathrm{vec}(C)=h} \|A - B \otimes C\|_F. \tag{3.2}$$

There are two ways to solve such an optimization problem, the method of *Lagrange multipliers* and the *Null Space* method. We will use the latter. As we know, the equivalent problem is

$$\min_{S_B^T b=g, S_C^T c=h} \|\tilde{A} - bc^T\|_F \tag{3.3}$$

where $b = \mathrm{vec}(B)$ and $c = \mathrm{vec}(C)$. Let us assume that the matrices S_B and S_C are full rank and their corresponding QR factorizations are

$$\mathrm{S_B} = \mathrm{Q_B} \begin{bmatrix} R_B \\ 0 \end{bmatrix} \text{ and } \mathrm{S_C} = \mathrm{Q_C} \begin{bmatrix} R_C \\ 0 \end{bmatrix}. \tag{3.4}$$

Then the constraint equations can be written as

$$\begin{bmatrix} R_B^T & 0 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \mathrm{g} \text{ and } \begin{bmatrix} R_C^T & 0 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \mathrm{h} \tag{3.5}$$

where

$$\mathrm{Q_B^T b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \text{ and } \mathrm{Q_C^T c} = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}. \tag{3.6}$$

This means that b_1 and c_1 have to be determined by solving the two independent lower triangular systems $R_B^T b_1 = g$ and $R_C^T c_1 = h$ while b_2 and c_2 are free from the constraints (unconstrained). Therefore, the minimization problem is equivalent to the unconstrained problem

$$\min \left\| \mathrm{Q_B^T \tilde{A} Q_C} - \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \begin{bmatrix} c_1^T & c_2^T \end{bmatrix} \right\|_{\mathrm{F}} \tag{3.7}$$

with $R_B^T b_1 = g$, $R_C^T c_1 = h$, that can be partitioned in a conformable way into

$$\min \left\| \begin{bmatrix} \hat{A}_{11} & \hat{A}_{12} \\ \hat{A}_{21} & \hat{A}_{22} \end{bmatrix} - \begin{bmatrix} b_1 c_1^T & b_1 c_2^T \\ b_2 c_1^T & b_2 c_2^T \end{bmatrix} \right\|_{\mathrm{F}} \tag{3.8}$$

where $R_B^T b_1 = g$, $R_C^T c_1 = h$ and $\hat{A} = Q_B^T \tilde{A} Q_C$.

3.1.1 Homogeneous Case

When the linear constraints are homogeneous, that is vectors g and h are equal to zero; the constraints set b_1 and c_1 to zero and thus (3.8) reduces to

$$\min \left\| \hat{A}_{22} - b_2 c_2^T \right\|_F \tag{3.9}$$

that we already know how to solve, using either Lanczos or the totally separable least squares method (Flip-Flop) back in Chapter 2. The initial unknowns, matrices B and C are filled up with the values of the vectors b and c that are given by $b = Q_B[b_1; b_2]$ and $c = Q_C[c_1; c_2]$. A Matlab function implementation is

```
function [B,C] = constr_kpa_h(A,mb,nb,Sb,Sc)
% Constrained Kronecker Product Approximation
% (Homogeneous Linear Constraints)
% Given A, mb, nb, Sb, Sc,
% (assuming that Sb and Sc have full row rank)
% Find B of size mb*nb and C the optimizers of
% min(A-kron(B,C),'fro') s.t. Sb'*vec(B) = 0, Sc'*vec(C) = 0

[ma,na] = size(A); mc = ma/mb; nc = na/nb;

rb = rank(Sb); [Qb,Rb] = qr(Sb); Qb2 = Qb(:,rb+1:mb*nb);
rc = rank(Sc); [Qc,Rc] = qr(Sc); Qc2 = Qc(:,rc+1:mc*nc);

[sigma,u,v] = lanczos(Qb2'*tilde(A,mb,nb)*Qc2);

B = zeros(mb,nb); C = zeros(mc,nc);

B(:) = Qb2*(sigma*u);
C(:) = Qc2*v;

return
```

With the solver of homogeneous linear constraints we can solve problems that require a specific structure on matrices B and C like certain matrix elements to be zero, or equal to each-other. Such constraints cover matrices to be symmetric, skew symmetric, banded, Toeplitz, or Hankel. The constraint matrices consist of zeroes, ones and minus ones, moreover they are so well structured that their QR factorization can be constructed rather than calculated. That means that very efficient algorithms resembling signal processing transformations can be developed for specific problems. We study such a problem from digital image processing in more detail later in §3.1.2.

For example, suppose we require matrices B and C to be *Toeplitz*, matrices that are constant along each of their diagonal. That is, if B is 3×4, it should look like

$$\mathrm{B} = \begin{bmatrix} \alpha & \beta & \gamma & \delta \\ \epsilon & \alpha & \beta & \gamma \\ \zeta & \epsilon & \alpha & \beta \end{bmatrix}. \tag{3.10}$$

Let $b = \mathrm{vec}(B)$, we will generate a $\{0, 1, -1\}$ matrix G to describe the constraints for B to be Toeplitz, such that $Gb = 0$. After observing that we have $n_B - 1$ successive pairs of columns and the first $m_B - 1$ elements of the first column of the pair should be equal to the last $m_B - 1$ elements of the second column of the pair, we claim that

$$G_{(m,n)} = M_{n-1} \otimes M_{m-1} - N_{n-1} \otimes N_{m-1}$$

$$M_n = [I_n \; 0] \text{ and } N_n = [0 \; I_n]$$

where M_n is the n-by-$n+1$ matrix consisting of the identity matrix of size n and a column of zeros, N_n is the n-by-$n+1$ matrix consisting of a column of zeros and the identity matrix of size n, and m, n are the dimensions of the matrix that has to be Toeplitz. For the above example, the constraint equations can be written as:

$$G_{(3,4)}\text{vec}(B) = 0.$$

where

$$G_{(3,4)} = \left[\begin{array}{ccc|ccc|ccc|ccc} 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \end{array}\right].$$

In a similar way we can construct constraint matrices for other structured matrices like symmetric, banded, Hankel, circulant, or arbitrary sparsity patterns. The constraints for orthogonal matrices are non-linear, while the Markov constraints are linear but also require non-negativity.

3.1.2 An Image Processing Application

This section is a case study for constrained Kronecker product approximation where the constraints are describing a symmetric Toeplitz matrix. The application comes from Image Processing, more precisely from the area of the statistical characterization of discrete images. The matrix A to be approximated is supposed to be the Kronecker product of two symmetric Toeplitz matrices, however numerical noise in the calculation of A makes necessary the constrained approximation. This application comes from a space-time adaptive signal processing problem for airborn radar [War94]. The main interest in this case study is to show that the null space of a structural constraint matrix such us the one describing a symmetric Toeplitz matrix can be constructed instead of calculated, and its application to a vector to be expressed as a fast transform.

The problem is to minimize

$$\min_{G_B \operatorname{vec}(B)=0, G_C \operatorname{vec}(C)=0} \|A - B \otimes C\|_F$$

where matrices G_B and G_C implement the constraints for B and C to be symmetric Toeplitz matrices. The matrix G is a $\{0, 1, -1\}$ matrix and has a regular structure. We do not really care to deal with G because we only need an orthonormal basis of its null space.

Consider the matrices

$$E_1 = \frac{1}{2}\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad E_2 = \frac{1}{\sqrt{6}}\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$
$$E_3 = \frac{1}{2}\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \quad E_4 = \frac{1}{\sqrt{2}}\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

that are scaled so that $\|E_i\|_F = 1$ for $i = 1, \cdots, 4$. It is easy to see that all 4-by-4 symmetric Toeplitz matrices T can be written as

$$T = x_1 E_1 + x_2 E_2 + x_3 E_3 + x_4 E_4$$

or equivalently

$$\text{vec}(T) = [\text{vec}(E_1), \text{vec}(E_2), \text{vec}(E_3), \text{vec}(E_4)]x$$

for the right choice of $x = [x_1, x_2, x_3, x_4]^T$. Note that, matrices E_i seen as column vectors, have unit length and they are orthogonal to each-other. That means that matrix

$$Q = [\text{vec}(E_1), \text{vec}(E_2), \text{vec}(E_3), \text{vec}(E_4)]$$

is an orthonormal basis of vec(T) the subspace of vectors that when expressed as 4-by-4 matrices, they are symmetric Toeplitz. Therefore Q is an orthonomal basis of the null space of constraint matrix G. In general, the orthonormal basis matrix Q of the subspace of symmetric Toeplitz matrices of size n, is of size n^2-by-n defined by the code segment:

```
function Q = null_sym_toep(n)
% find an orthonormal basis of the subspace of
% symmetric Toeplitz matrices of size n

v=zeros(n,1);v(1) = 1;
Q(:,1) = 1/sqrt(n)*vec(toeplitz(v));
for i=2:n
  v = zeros(n,1);
  v(i) = 1;
  Q(:,i) = 1/sqrt(2*(n-i+1))*vec(toeplitz(v));
end
return
```

However, we need not form matrix Q explicitly. The matrix-vector product Qx can be performed in linear time, all columns of Q access different elements of x.

3.1.3 Non-Homogeneous Linear Case

On the other hand, when the linear equality constraints are not homogeneous, the problem cannot be reduced to the unconstrained rank-one approximation. Unlike the homogeneous linear equality constraints, the off-diagonal blocks of the matrix function to minimize in (3.9) contain unknowns since vectors b_1 and c_1 are not necessarily zero.

However, if we look at the gradient of the function presented in (3.8), the formula for the gradient is given in (2.6), we get

$$-2\begin{bmatrix} \hat{A}_{11}c_1 + \hat{A}_{12}c_2 - (c_1^Tc_1 + c_2^Tc_2)b_1 \\ \hat{A}_{21}c_1 + \hat{A}_{22}c_2 - (c_1^Tc_1 + c_2^Tc_2)b_2 \\ \hat{A}_{11}^Tb_1 + \hat{A}_{21}^Tb_2 - (b_1^Tb_1 + b_2^Tb_2)c_1 \\ \hat{A}_{12}^Tb_1 + \hat{A}_{22}^Tb_2 - (b_1^Tb_1 + b_2^Tb_2)c_2 \end{bmatrix}. \tag{3.11}$$

The gradient this time consists of only the second and the fourth rows of (3.11)

$$g(b_2, c_2) = \begin{bmatrix} \nabla_{b_2} f \\ \nabla_{c_2} f \end{bmatrix} = -2\begin{bmatrix} \hat{A}_{21}c_1 + \hat{A}_{22}c_2 - (c_1^Tc_1 + c_2^Tc_2)b_2 \\ \hat{A}_{12}^Tb_1 + \hat{A}_{22}^Tb_2 - (b_1^Tb_1 + b_2^Tb_2)c_2 \end{bmatrix} \tag{3.12}$$

The values of b_2 and c_2 that minimize (3.8), also zero the above expressions. By setting the right-hand-side of (3.12) to zero, we notice that the unknowns b_2 and c_2 are totally separable.

$$b_2 = \frac{\hat{A}_{21}c_1 + \hat{A}_{22}c_2}{c_1^Tc_1 + c_2^Tc_2} \tag{3.13}$$

$$c_2 = \frac{\hat{A}_{12^T}b_1 + \hat{A}_{22}^Tb_2}{b_1^Tb_1 + b_2^Tb_2} \tag{3.14}$$

This allows us to use the flip flop method for totally separable least squares. Here is a MATLAB implementation

```
function [B,C] = constr_kpa_nh(A,mb,nb,Sb,Sc,g,h)
% Constrained Kronecker Product Approximation
% (Non-Homogeneous L Constraints)
% Given A, mb, nb, Sb, Sc, g, h
% (assuming that Sb and Sc have full row rank)
% Find B of size mb*nb and C the optimizers of
% min(A-kron(B,C),'fro') s.t. Sb'*vec(B) = g, Sc'*vec(C) = h
```

```
[ma,na] = size(A); mc = ma/mb; nc = na/nb;

rb = rank(Sb); [Qb,Rb] = qr(Sb); b1 = Rb(1:rb,:)'\g;
rc = rank(Sc); [Qc,Rc] = qr(Sc); c1 = Rc(1:rc,:)'\h;

Ah = Qb'*tilde(A,mb,nb)*Qc;
Ah11 = Ah(1:rb,1:rc);       Ah12 = Ah(1:rb,rc+1:mc*nc);
Ah21 = Ah(rb+1:mb*nb,1:rc); Ah22 = Ah(rb+1:mb*nb,rc+1:mc*nc);

c2 = randn(mc*nc-rc,1);
conv = 0; iter = 0;
while ~conv
  iter = iter+1;
  b2 = (Ah21*c1 + Ah22*c2)/(c1'*c1 + c2'*c2);
  pc2 = c2;
  c2 = (Ah12'*b1 + Ah22'*b2)/(b1'*b1 + b2'*b2);
  conv = norm(pc2 - c2) < 1e-12;
end

B = zeros(mb,nb); C = zeros(mc,nc);
B(:) = Qb*[b1; b2];
C(:) = Qc*[c1; c2];

return
```

This approach is similar to the power method however part of the vectors have to remain constant. We have no knowledge of its convergence properties. We come back and study this problem in its most general form, the rank-one approximation with linear equality constraints, in section 3.4. The above procedure performs very well for homogeneous constraints because the flip-flop formulae are identical. The vectors and scalars `Ah12'*b1`, `b1'*b1`, `Ah21*c1` and `c1'*c1` become zero.

3.2 Special Constraints

Constraining B and C to be stochastic or orthogonal leads to some interesting computational challenges. The constraints are nonlinear and require special handling.

3.2.1 Stochastic Matrices

It is often possible to represent the behavior of a physical system by describing all the different states which it can occupy and by indicating how it moves from one state to another in time. If the future evolution of the system depends only on its current state, the system may be represented by a Markov process. When the state space is discrete, the term "Markov Chain" is employed. The matrix of probabilities for moving from state j to state i is a stochastic matrix [Ste95][1]. Such a matrix is non-negative, with column sums equal to 1, for example:

$$M = \begin{bmatrix} 0.110 & 0.078 & 0.089 & 0.044 & 0.168 & 0.038 & 0.070 & 0.152 & 0.005 \\ 0.114 & 0.040 & 0.191 & 0.132 & 0.180 & 0.051 & 0.126 & 0.123 & 0.202 \\ 0.026 & 0.234 & 0.176 & 0.080 & 0.060 & 0.133 & 0.208 & 0.115 & 0.031 \\ 0.196 & 0.208 & 0.163 & 0.175 & 0.056 & 0.242 & 0.068 & 0.071 & 0.242 \\ 0.145 & 0.065 & 0.143 & 0.114 & 0.036 & 0.067 & 0.056 & 0.181 & 0.177 \\ 0.060 & 0.273 & 0.086 & 0.067 & 0.113 & 0.136 & 0.074 & 0.039 & 0.093 \\ 0.084 & 0.007 & 0.039 & 0.086 & 0.016 & 0.010 & 0.100 & 0.023 & 0.103 \\ 0.085 & 0.034 & 0.072 & 0.138 & 0.212 & 0.290 & 0.055 & 0.158 & 0.058 \\ 0.180 & 0.060 & 0.041 & 0.164 & 0.159 & 0.033 & 0.244 & 0.139 & 0.089 \end{bmatrix}$$

Although you can very easily prove that the Kronecker product of two stochastic matrices is stochastic, for instance,

$$\begin{bmatrix} 0.019 & 0.666 \\ 0.981 & 0.334 \end{bmatrix} \otimes \begin{bmatrix} 0.467 & 0.871 \\ 0.533 & 0.129 \end{bmatrix} = \begin{bmatrix} 0.009 & 0.017 & 0.311 & 0.581 \\ 0.010 & 0.002 & 0.355 & 0.086 \\ 0.459 & 0.855 & 0.156 & 0.291 \\ 0.522 & 0.126 & 0.178 & 0.043 \end{bmatrix}$$

however, as it was shown in [VLP92], the Kronecker product approximation of a stochastic matrix does not consist necessarily of stochastic matrices, for example the closest Kronecker product to M above is:

[1]Often in literature, the stochastic matrix is defined as a non-negative matrix with *row* sums equal to 1. Non-negative matrices with both row and column sums equal to one are called *doubly-stochastic*.

$$B = \begin{bmatrix} 0.358 & 0.275 & 0.350 \\ 0.442 & 0.318 & 0.327 \\ 0.202 & 0.395 & 0.321 \end{bmatrix}, \; C = \begin{bmatrix} 0.314 & 0.268 & 0.308 \\ 0.320 & 0.337 & 0.430 \\ 0.346 & 0.417 & 0.265 \end{bmatrix}$$

with column sums

$$M = \begin{bmatrix} 1.001 & 0.989 & 0.998 & 0.979 & 1.022 & 1.003 \end{bmatrix}.$$

We are interested in solving the constrained Kronecker product problem where the matrices B and C are required to be stochastic. The meaning of such an approximation is the following: if A is a matrix of probabilities for moving from state j to state i, then the nearest Kronecker product of two stochastic matrices B and C is equivalent to considering a two dimensional system (or a pair of independent Markov systems) where there are n_B the size of B "super-nodes" of identical structure, consisting of n_C nodes each. B holds the probabilities moving across "super-nodes" and C the probabilities moving within a super-node. Such a reduction can be considered as a form of *lumping*.

The general form of the problem is

$$\min_{e_{n_B}^T B = e_{n_B}, B \geq 0, e_{n_C}^T C = e_{n_C}, C \geq 0} \|A - B \otimes C\|_F. \tag{3.15}$$

where e is the vector of all ones. Such problems involving linear equality and inequality constraints can be solved using the active set family of methods or a penalty function approach in order to handle the inequality constraints, for an extended analysis see Fletcher, page 264ff [Fle90].

Another idea is to use the usual flip flop algorithm but transform each step into a linear inequality constrained least squares problem (LSI). This is done as follows: Consider that c is known and we are trying to solve the problem

$$\min_{e_{n_B}^T B = e_{n_B}, b \geq 0} \|\hat{A} - bc^T\|_F \tag{3.16}$$

for b. This is equivalent to the following least squares (LS) problem

$$\min_{e_{n_B}^T B = e_{n_B}, b \geq 0} \|(I_{m_B n_B} \otimes c)b - vec(\hat{A}^T)\|_F \tag{3.17}$$

that we know how to solve [LH74]. A similar formulation exists for fixing b and solving for c

$$\min_{e_{n_C}^T C = e_{n_C}, c \geq 0} \|(I_{m_C n_C} \otimes b)c - vec(\hat{A})\|_F. \tag{3.18}$$

However, such an approach is very expensive and is presented here only for the sake of completeness. Algorithms for solving the LSI problem are presented in [LH74]. We provide a complete suite of routines for solving

- non-negative least squares problem (NNLS), that is find vector x that

  $$\min_{x \geq 0} \|Ax - b\|_2,$$

- least distance programming (LDP)

  $$min_{Gx \geq h} \|x\|_2,$$

  via a reduction to NNLS,

- least squares with linear inequality constraints (LSI) via a reduction to LDP, and

- the linear equality and inequality constrained least squares problem (LSIE), via a reduction to LSI,

in Matlab , in appendix A.

We have also seen, after running several experiments that when matrix A is stochastic then the equality constraints requiring the column sums of B and C to be ones are also producing non-negative matrices, thus stochastic. Actually we can even relax the requirement for the matrices to be square.

Of course, we can also substitute the constraints into the main expression and then solve the unconstrained optimization problem. Here is the iterative algorithm that comes from the substitution of the constraints. It is of the same run-time complexity but runs much faster in practice, than the general linear equality constraint minimizer. With this we want to show that although the general rank-one approximation problem with constraints is a hard one, we can come up with algorithms that perform very well for specific problem instances. The advantages come from the exploitation of the structure of the specific problem.

Suppose we are given a stochastic matrix A, we have a guess for stochastic matrix C and we want to find a stochastic matrix B that minimizes the norm $\|A - B \otimes C\|_F^2$. Since the values of the columns of matrix B are independent with each other, without loss of generality, let us assume that B consists of exactly one column. With regard to the column-sum one constraint for B we can rewrite the problem as

$$\left\| \begin{bmatrix} A_1 \\ \vdots \\ A_n \end{bmatrix} - \begin{bmatrix} b \\ 1 - \sum b \end{bmatrix} \otimes \mathrm{C} \right\|_{\mathrm{F}} \tag{3.19}$$

where A_i are blocks of A congruent to C and $n = m_B$. The above expression has the same minimizer with the function

$$\mathrm{f(b)} = \left\| \begin{bmatrix} A_1 \\ \vdots \\ A_{n-1} \end{bmatrix} - \mathrm{b} \otimes \mathrm{C} \right\|_{\mathrm{F}}^2 + \left\| \mathrm{A_n} - (1 - \sum \mathrm{b})\mathrm{C} \right\|_{\mathrm{F}}^2 \tag{3.20}$$

which has gradient given by the formula

$$\frac{\partial f}{\partial b_i} = -2\mathrm{tr}(C^T A_i) + 2b_i \mathrm{tr}(C^T C) + 2\mathrm{tr}(C^T A_n) - 2(1 - \sum b)\mathrm{tr}(C^T C). \tag{3.21}$$

When we set the gradient to zero and do some simplifications using the additive and multiplicative properties of trace, we get the relation

$$b_i + \sum b = \frac{\mathrm{tr}(C^T(A_i - A_n + C))}{\mathrm{tr}(C^T C)}. \tag{3.22}$$

For general matrix B, the above relation becomes

$$b_{ij} + \sum_{k=1}^{m_B - 1} b_{ik} = \frac{\mathrm{tr}(C^T(A_{ij} - A_{m_B j} + C))}{\mathrm{tr}(C^T C)}. \tag{3.23}$$

For a given C we can find b and thus B by solving a multiple right hand side system of linear equations whose matrix G_n of size n has a very special structure, all elements are equal to one except the diagonal, that are equal to two. For example, here is

$$G_8 = \begin{bmatrix} 2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 2 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 2 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 2 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 2 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 2 \end{bmatrix}$$

Solving a system with such a matrix is very easy, it can be done in linear time. Let us see how. $G_n = I_n + 1_n$ where 1_n is the matrix of ones of size n, `ones(n)` in Matlab . It is easy to see that

$$G_n^{-1} = I_n - \frac{1}{n+1} 1_n$$

because

$$\begin{aligned} (I_n + 1_n)(I_n - \frac{1}{n+1} 1_n) &= \\ I_n - \frac{1}{n+1} 1_n + 1_n - \frac{n}{n+1} 1_n &= \\ I_n + (-\frac{1}{n+1} + 1 - \frac{n}{n+1}) 1_n &= I_n. \end{aligned}$$

Thus solving the linear system $G_n b = d$ for b takes only $2n$ flops because

$$b = G_n^{-1} d = d - \frac{1}{n+1} \sum d.$$

In the same way we can solve for C, knowing B, after we exchange them by using the commutative property of the Kronecker product defined in (1.9).

```
function [B,C] = kpa_markov(A,mb,nb)
% Kronecker Product Approximation with stochastic matrices
% Given A: the matrix to approximate (has to be stochastic
% mb, nb: the size of matrix B

[ma,na] = size(A);
mc = ma/mb; nc = na/nb;

At = P(mb*mc,mb)'*A*P(nb*nc,nc)';
B = rand(mb,nb); s = sum(B); for j=1:nb, B(:,j) = B(:,j)/s(j); end
C = rand(mc,nc); s = sum(C); for j=1:nc, C(:,j) = C(:,j)/s(j); end

iter = 0; conv = 0; nrm = Inf;

while ~conv
  iter = iter + 1;
  R = [];
  % fix C, solve for B
  cc = sum(sum(C.*C));
  for j=1:nb
    for i=1:mb-1
      Ai = A((i-1)*mc+1:i*mc,(j-1)*nc+1:j*nc);
      An = A((mb-1)*mc+1:mb*mc,(j-1)*nc+1:j*nc);
      R(i,j) = sum(sum(C .*(C+Ai-An)))/cc;
    end
  end

  B(1:mb-1,:) = R - kron(ones(mb-1,1),1/mb*sum(R));
  B(mb,:) = ones(1,nb) - sum(B(1:mb-1,:));

  R = [];
  % fix B, solve for C
  bb = sum(sum(B.*B));
  for j=1:nc
```

```
    for i=1:mc-1
      Ai = At((i-1)*mb+1:i*mb,(j-1)*nb+1:j*nb);
      An = At((mc-1)*mb+1:mc*mb,(j-1)*nb+1:j*nb);
      R(i,j) = sum(sum(B.*(B+Ai-An)))/bb;
    end
  end

  C(1:mc-1,:) = R - kron(ones(mc-1,1),1/mc*sum(R));
  C(mc,:) = ones(1,nc) - sum(C(1:mc-1,:));
  pn = nrm;  nrm = norm(A - kron(B,C),'fro');
  conv = pn - nrm < 1e-8;
end
```

Note that the above code works for non-square matrices too. Assuming that A is of size n^2-by-n^2, this algorithm requires $6n^4$ flops per iteration with some precomputation. Going back to our random stochastic matrix M, we can find the closest Kronecker product of stochastic matrices to be

$$B = \begin{bmatrix} 0.356 & 0.278 & 0.351 \\ 0.440 & 0.323 & 0.326 \\ 0.203 & 0.399 & 0.322 \end{bmatrix}, \; C = \begin{bmatrix} 0.320 & 0.260 & 0.307 \\ 0.327 & 0.330 & 0.429 \\ 0.353 & 0.409 & 0.264 \end{bmatrix}.$$

As you can see, the matrices are very close to the uncostrained Kronecker product approximation, only this time they are stochastic.

In order to claim that we are lumping or approximating a Markov system with two independent Markov systems (a 2-dimensional system), we actually need to solve

$$\min \|PAP^T - B \otimes C\|_F \tag{3.24}$$

for matrices P, B and C, where P is a permutation matrix and B and C are stochastic. The permutation P allows us to "rename" states and thus lump them in any order in groups other than the sequential $1, \cdots, m_c$, $m_c+1, \cdots, 2m_c$, and so on that we currently achieve with the constrained Kronecker product approximation.

However, even the simpler problem of finding

$$\min \|PAP^T - B\|_F$$

for permutation matrix P, where A and B are square matrices, is NP-hard; it corresponds to the Bandwidth Reduction problem from [GJ79]. It is equivalent to trying to minimize the bandwidth of a symmetric matrix by simultaneous row and column permutations. We must note here that the problem of minimizing either

$$\min \|PA - B\|_F \text{ or } \min \|AP^T - B\|_F$$

is solvable in polynomial time, actually it is a weighted bipartite matching problem.

3.2.2 Orthogonal Matrices

Also it is of interest to find, given a square matrix A, a pair of orthogonal matrices Q_B and Q_C such that their Kronecker product is the closest to A according to the Frobenius norm. This is a Kronecker product approximation with non-linear equality constraints. Again, the Kronecker product of two orthogonal matrices is orthogonal due to the matrix multiplication and transposition properties of the Kronecker product, however, the closest Kronecker product to an orthogonal matrix

$$M = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

is not necessarily orthogonal as it can be seen from the two closest Kronecker products that are $\sqrt{2}$ away from M:

$$B_1 = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}, \; C_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

and

$$B_2 = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \; C_2 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

A possible approach is to solve the unconstrained problem $\|A - B \otimes C\|_F$ and then take the closest orthogonal matrices Q_B and Q_C to B and C respectively. The *closest orthogonal matrix* Q_A of a square matrix A is defined to be the minimizer of

$$\min \|A - Q\|_F \text{ for all orthogonal matrices } Q$$

and can be shown that it is the product of the matrices whose columns are the left and right singular vectors of A, that is, if $A = U\Sigma V^T$ is the Singular Value Decomposition of A, then $Q_A = UV^T$. But Q_B and Q_C is not necessarily the pair that solves this optimization problem with non-linear constraints. For the above pair of matrices B and C, the closest orthogonal matrix to Bs is the identity matrix, the C matrices are already orthogonal.

Below, we give a method for iteratively improving the approximation with the Kronecker product of two orthogonal matrices. Let A, a square matrix of size $n_B n_C$, that we want to approximate with the Kronecker product $B \otimes C$ where B and C are orthogonal and of sizes n_B and n_C respectively. Suppose we have somehow acquired an orthogonal B and we want to solve the following for C

$$\min_{C^T C = I_{n_C},\, CC^T = I_{n_C}} \|A - B \otimes C\|_F^2. \tag{3.25}$$

The above expression is equivalent to

$$\min_{C^T C = I_{n_C},\, CC^T = I_{n_C}} \|A(B^T \otimes I_{n_C}) - I_{n_B} \otimes C\|_F^2. \tag{3.26}$$

Let $Y = A(B^T \otimes I_{n_C})$, then we can reduce the above to

$$\min_{C^TC=I_{n_C},\, CC^T=I_{n_C}} \sum_{i=1}^{n_B} \|Y_{ii} - C\|_F^2 \tag{3.27}$$

where Y_{ii} is the i^{th} diagonal block of Y of size n_C-by-n_C. Subsequently we can transform the above expression

$$\begin{aligned}
&= \min_{C^TC=I_{n_C},\, CC^T=I_{n_C}} \sum_{i=1}^{n_B} \operatorname{tr}((Y_{ii} - C)^T(Y_{ii} - C)) \\
&= \min_{C^TC=I_{n_C},\, CC^T=I_{n_C}} \sum_{i=1}^{n_B} \operatorname{tr}(Y_{ii}^TY_{ii} + C^TC - Y_{ii}^TC - C^TY_{ii}) \\
&\equiv \min_{C^TC=I_{n_C},\, CC^T=I_{n_C}} \sum_{i=1}^{n_B} \left(n_C - 2\operatorname{tr}(C^TY_{ii})\right) \\
&= \min_{C^TC=I_{n_C},\, CC^T=I_{n_C}} \left(n_Bn_C - 2\sum_{i=1}^{n_B} \operatorname{tr}(C^TY_{ii})\right) \\
&= \min_{C^TC=I_{n_C},\, CC^T=I_{n_C}} \left(n_Bn_C - 2\operatorname{tr}(C^T\sum_{i=1}^{n_B} Y_{ii})\right) \\
&= \min_{C^TC=I_{n_C},\, CC^T=I_{n_C}} \left(n_Bn_C - 2\operatorname{tr}(C^TZ)\right) \qquad (3.28)
\end{aligned}$$

using the properties of trace and the fact that $\operatorname{tr}(C^TC) = n_C$. In the last expression we also substituted $Z = \sum_{i=1}^{n_B} Y_{ii}$. Therefore, we need to maximize the expression $\operatorname{tr}(C^TZ)$ for all orthogonal matrices C. In general, the problem

$$\max_{Q \text{ orthogonal}} \operatorname{tr}(Q^TA)$$

is equivalent to the closest orthogonal matrix problem because the minimizer of $\|A - Q\|_F$ is

$$\begin{aligned}
&= \operatorname{argmin} \left(\|A - Q\|_F^2\right) \\
&= \operatorname{argmin} \operatorname{tr}\left((A - Q)^T(A - Q)\right) \\
&= \operatorname{argmin} \left(\operatorname{tr}(A^TA) - 2\operatorname{tr}(Q^TA) + \operatorname{tr}(Q^TQ)\right) \\
&= \operatorname{argmax} \operatorname{tr}(Q^TA).
\end{aligned}$$

Thus the minimizer C of (3.28) is the closest orthogonal matrix to Z. In a similar way, now that we know C we can solve for B. This is how it goes:

$$\begin{aligned}
& \min_{B^TB=I_{n_B},\, BB^T=I_{n_B}} \|A - B\otimes C\|_F^2 \\
= & \min_{B^TB=I_{n_B},\, BB^T=I_{n_B}} \|A(I_{n_B}\otimes C^T) - B\otimes I_{n_C}\|_F^2 \\
= & \min_{B^TB=I_{n_B},\, BB^T=I_{n_B}} \|P^T_{n_Bn_C,n_B} A(I_{n_B}\otimes C^T)P^T_{n_Bn_C,n_C} - I_{n_C}\otimes B\|_F^2 \qquad (3.29)
\end{aligned}$$

where P is the stride permutation matrix from the commutative property of the Kronecker product. The last expression of (3.29) is identical to the expression presented in (3.26) so the solution proceeds in the same way. The following MATLAB function shows how.

```
function [Qb,Qc,nrm] = kpa_orth(A,Qb,Qc)
% find two orthogonal matrices Qb and Qc whose Kronecker product
% is closest to A
% Qb,Qc: Initial guess

  [nA,nA] = size(A); [nB,nB] = size(Qb); [nC,nC] = size(Qc);

  done = 0;
  while ~done
    done = 1;
    nrm = norm(A - kron(Qb,Qc),'fro');
    Y = A * kron(Qb',eye(nC));
    Z = zeros(nC);
    for i=1:nB
      p = (i-1)*nC+1:i*nC;
      Z = Z + Y(p,p);
    end

    [U,S,V] = svd(Z);
    newQc = U*V';
    newnrm = norm(A - kron(Qb,newQc),'fro');
    if newnrm < nrm, Qc = newQc; nrm = newnrm; done = 0; end

    Y = A * kron(eye(nB),Qc'); Y = P(n,nC)*Y*P(n,nB);
    Z = zeros(nB);
    for i=1:nC
```

```
      p = (i-1)*nB+1:i*nB;
      Z = Z + Y(p,p);
    end

    [U,S,V] = svd(Z);
    newQb = U*V';
    newnrm = norm(A - kron(newQb,Qc),'fro');
    if newnrm < nrm, Qb = newQb; nrm = newnrm; done = 0; end
  end

return
```

Each iteration involves the calculation of two SVDs of sizes n_B-by-n_B and n_C-by-n_C in addition to the flops required for the calculation of the matrices Y and Z. However the algorithm can be implemented without calculating the whole Y and never forming matrices $Q_B^T \otimes I$ and $I \otimes Q_C^T$.

Optimizations with non-linear constraints have more than one local minima, moreover convergence to a minimum close to the global minimum depends heavily on the starting point. For this problem, we found that a good starting point is the pair of the closest orthogonal matrices to the Kronecker product approximation of A. Figure 3.1 displays the plots of the residuals that resulted from 100 runs from random initial orthogonal matrices for 16 different problems. Circles display the residual resulting from starting from the closest orthogonal matrices to the Kronecker product approximation of A while crosses starting from the identity matrices. Matrix A is a random orthogonal matrix of size 16 and is approximated by two size-4 orthogonal matrices. We see that in more than half of the different approximation problems displayed, starting from the closest orthogonal matrices to the Kronecker product approximation resulted in converging to the group of the lowest (may be the optimal) residuals.

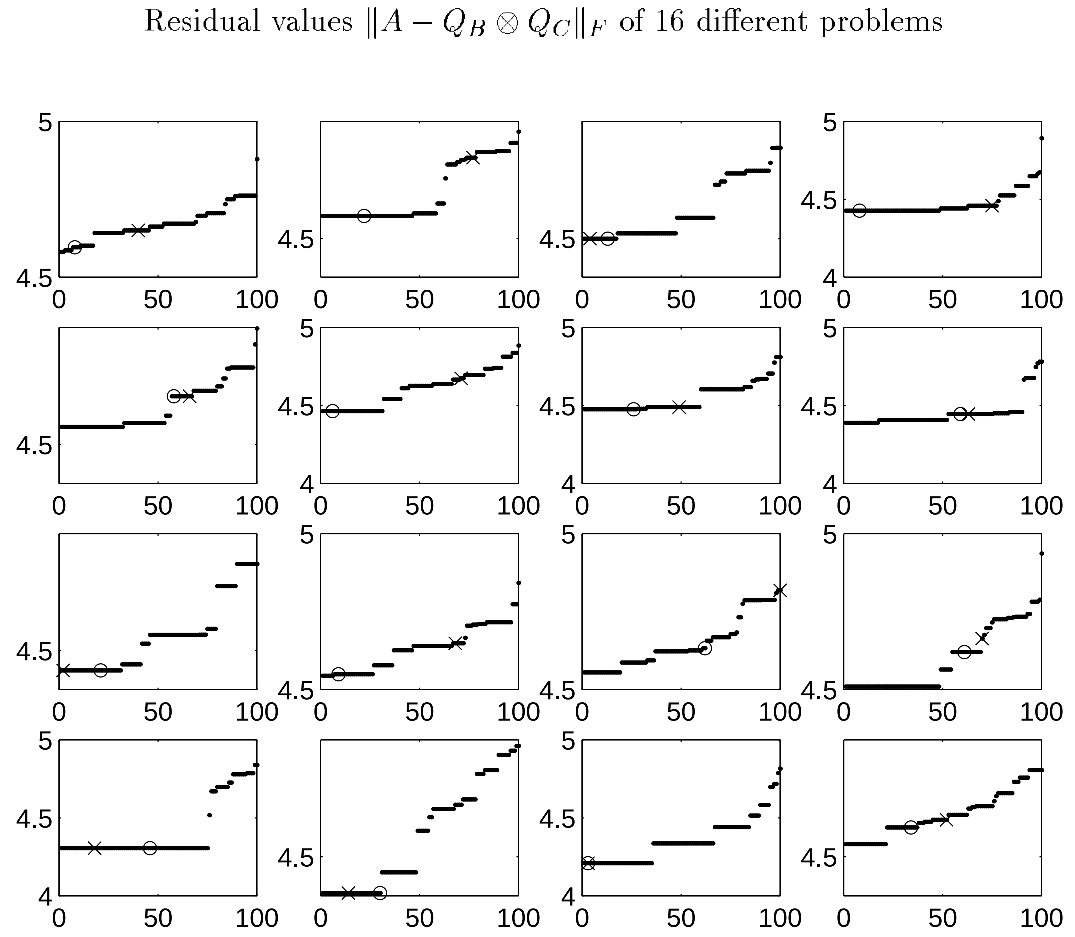

Figure 3.1: Residuals of 16 different problems that resulted from 100 runs from random initial orthogonal matrices. Circles display the residual resulting from starting from the closest orthogonal matrices to the Kronecker product approximation of A while crosses denote the residuals achieved by starting from the identity matrices. Matrix A is a random orthogonal matrix of size 16 and is approximated by two size-4 orthogonal matrices. We see that in more than half of the different approximation problems displayed, starting from the closest orthogonal matrices to the Kronecker product approximation resulted in converging to the group of the lowest (may be the optimal) residuals.

3.3 A Unified MATLAB Front-End

Although building the constraint matrices S_B and S_C is an easy task, it is not trivial, especially when there is a need for experimentation and a frequent change of problem size and constraint type. For these reasons we offer a MATLAB function that provides a common interface for all constrained and unconstrained algorithms, moreover this function generates the constraint matrices automatically. The general call is

```
[B,C] = kpa(A,mB,nB,eps,constrB,constrC,SB,SC,g,h)
```

where all but the first three arguments are optional. `A` is the matrix to be approximated, `mB, nB` are the dimensions of the first Kronecker factor, the matrix `B`. Optional variables are the convergence criterion `eps`, the string arrays [2] `constrB` and `constrC` are containing keywords describing the constraints. The recognized keywords are: `none`, for the unconstrained case, `symmetric`, `skewsymmetric`, `tril`, `triu` for lower and upper triangular matrix constraints, `diagonal`, `tridiagonal`, `toeplitz`, `circulant`, `hankel`, `markov`, `special` and `orthogonal`. When the `special` keyword is used for either arguments, the user should provide the final four arguments of the function call, that are used to create the constraints:

$$S_B^T \mathrm{vec}(B) = g,\ S_C^T \mathrm{vec}(C) = h.$$

For example in order to define that B must be symmetric, tridiagonal and Toeplitz and C to be lower triangular, arguments `constrB` and `constrC` must be declared as

```
cB = ['trid';'symm';'toep']
cC = ['tril']
[B,C] = kpa(A,mb,nb,1e-6,cB,cC)
```

[2] MATLAB requires all elements of a string array to have the same length.

Note that abbreviations are used for the constraint names, however all abbreviations must be of the same length so that the string array is valid in MATLAB .

Our MATLAB function implementation provides no search for conflicting constraints. It is the responsibility of the user to provide meaningful constraints.

Due to its size, the MATLAB code for function `kpa` is listed in the Appendix A.

3.4 Rank-One Approximation with Linear Equality Constraints

As we have seen so far, all linear constrained approximations (except the Markov) are rank-one approximations with linear equality constraints. In this section we provide some proofs that reveal the difficulty of the problem and underline our approach to use iterative methods that converge to a solution.

We can prove the following:

Lemma 3.4.1 *Solving*

$$\min_{S^T u=g\ T^T v=h} \|A - uv^T\|_F^2 \tag{3.30}$$

where S and T are full-rank matrices, is equivalent to solving

$$\min \left\| \mathrm{Q}^{\mathrm{T}}\mathrm{AZ} - \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \begin{bmatrix} v_1^T & v_2^T \end{bmatrix} \right\|_{\mathrm{F}}^2 \tag{3.31}$$

where u_2 and v_2 are free variables and

$$S = Q\begin{bmatrix} R_S \\ 0 \end{bmatrix}, \qquad T = Z\begin{bmatrix} R_T \\ 0 \end{bmatrix} \tag{3.32}$$

$$R_S u_1 = g, \qquad R_T v_1 = h \tag{3.33}$$

$$u = Q\begin{bmatrix} u_1 \\ u_2 \end{bmatrix}, \qquad v = Z\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \tag{3.34}$$

Proof: Straight forward application of the null space basis of the constraint matrices. □

Lemma 3.4.2 *The solution of*

$$\min \left\| \mathrm{A} - \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \begin{bmatrix} v_1^T & v_2^T \end{bmatrix} \right\|_{\mathrm{F}}^2 \tag{3.35}$$

where u_1 and v_1 are constant vectors, is equivalent to solving the system

$$\begin{aligned} A_{22}v_2 &= (v_1^T v_1 + v_2^T v_2)u_2 + A_{21}v_1 \\ A_{22}^T u_2 &= (u_1^T u_1 + u_2^T u_2)v_2 + A_{12}^T u_1 \end{aligned}$$

where A_{ij} are the congruent blocks of matrix A.

Proof: Via expansion of the objective function and zeroing of the gradient. □

Lemma 3.4.3 *The solution of the non-linear system of equations*

$$\begin{aligned} Ax &= (\alpha + x^T x)y + a \\ A^T y &= (\beta + y^T y)x + b \end{aligned} \tag{3.36}$$

for a m-by-n matrix A with $m \geq n$, non-negative reals α and β and vectors a and b, is equivalent to solving a system of n multi-varied polynomials of n variables of degree 5.

Proof: We solve the first of the equations in (3.36) for y

$$y = \frac{Ax - a}{\alpha + x^T x}$$

given that the scalar $\alpha + x^T x$ is non-zero. After substituting y to the second equation of (3.36)

$$\beta(\alpha + x^T x)^2 x + (\alpha + x^T x)^2 b + (Ax - a)^T (Ax - a)x - (\alpha + x^T x)A^T(Ax - a) = 0.$$

From the above we conclude that we have to solve a system of n multivariate polynomials $p_i(x_1, x_2, \cdots, x_n)$ for $i = 1, \cdots, n$ of degree 5. In order to find the set of common (real) zeros of a set of n polynomials of n variables we can employ various methods like the multivariate resultant and Gröbner Bases. For more information the reader is directed to the collection of articles in [KL92]. $\square$

Chapter 4

Automatic Implementations of Kronecker Product Formulae

In this chapter we present a system for automating the implementation of efficient FFTs and a range of fast transforms using symbolic manipulation and program transformations. We do so in an abstract way so that the resulting system can handle other fast transforms that can be expressed as a sparse matrix factorization with Kronecker products. The Kronecker product plays a very important role in this work because most of our program transformations are based on Kronecker product identities.

The first generation of FFTs were described at the scalar level with very complicated index expressions. The situation is more amenable to automation when the FFT is related to a factorization of the DFT matrix with factors that are Kronecker products of diagonal matrices, or permutation matrices, or smaller Fourier transform matrices.

High performance FFTs typically contain at least a few "hand-crafted" *butter-*

flies, a term inherited from signal flow graphs that describes the algorithm to join a sequence of small FFT's to build a larger one. Central to the butterfly computation is the computation of a small FFT whose size is called the *radix* of the butterfly. The construction of these small FFT algorithms is an art, since there has not been presented a formal/constructive way for generating them. Authors typically list the algorithms as a sequence of assignments that achieve the desired result and report the flop count required [VL92,Nus82]. We are able to automate this task.

The implementation of a radix-p FFT can be split into two separate tasks.

1. Building the nested loops that define the order the vector elements are accessed.

2. Building a sequence of assignment statements that perform the calculation of a single transform of length p.

We provide programs that automatically generate each part. The automatic generation of FFT implementations permits the generation of large-radix butterflies. This is attractive because high radix algorithms have the advantages of reduced flop count and better memory traffic patterns. The automation supports the construction of FFT butterflies that are just the right size for a given problem and computer architecture combination. Although we produce output that is either a FORTRAN or FORTRAN 90 subroutine, or a MATLAB function, we can easily extend our system to other iterative programming languages or hardware specification languages.

Our system is most closely related to the work presented in a series of publications [GKH+94,KSH+92a,HJJ92,DGK+94] where direct translation of Kronecker product formulae is used to obtain programs. More specifically, the Kronecker product identities together with stride permutation matrices from § 1.3 are used to determine loop limits and data distributions for a predefined loop structure. For that, they provide an expert environment called EXTENT to compile, execute, and benchmark the resulting code, so as to decide which combination of algorithms and factorizations are the best for a given problem and computer architecture. Similar to our system, EXTENT starts with a factorization of the DFT matrix. However in [DGK+94], it is required that the factorization be expressed in the generic form

$$\prod_j (I_{f(j)} \otimes A_{g(j)} \otimes I_{h(j)})$$

with A a butterfly or stride permutation matrix. The authors also require the knowledge of the actual problem sizes and depend on hand coding the butterflies of the algorithms.

Our work involves algebraic formula manipulation and code transformations at a symbolic level. Since the actual matrices are never built their size need not be known. We use the Kronecker product notation together with other linear algebra operators and code transformation techniques from compiler theory to generate the loop nests required. We do so at a purely symbolic level. Moreover we are able to automatically generate butterflies of any radix size using a technique similar to loop unrolling.

4.1 The Language of Factorizations

For us, having a fast transform for the n-by-n matrix vector product Mx means that we have a factorization

$$M = M_t M_{t-1} \cdots M_1 \; t \ll n$$

with the property that each update $x := M_q x$ involves $O(n)$ flops. Therefore the loop

```
for $q = 1 : t$
    $x = M_q x$
end
```

overwrites x with Mx in $O(tn)$ flops.

We introduce this point of view by discussing the digit-reversal transformation. This is followed by several FFT examples and then a sampling of other fast transforms that arise in signal processing.

4.1.1 Digit Reversal

The digit reversal transformation is a permutation. We say *digit* and not *bit* reversal because in a radix-p algorithm, the permutation is equivalent to reordering the elements of a vector by reversing the order of digits of the indices written as base-p numbers. The digit reversal permutation matrix can be defined as

$$R_{n^t,n} = \prod_{q=t,t-1}^{1} \left(I_{n^{t-q}} \otimes P_{n^q,n^{q-1}}\right)$$

or the transposed equivalent

$$R_{n^t,n} = \prod_{q=1}^{t} \left(I_{n^{t-q}} \otimes P_{n^q,n}\right). \tag{4.1}$$

The equivalence follows because the reverse of the reverse ordering is the original ordering. Therefore the inverse of the digit reversal matrix is itself. The notation $\prod_{q=t,t-1}^{1} A_q$ denotes that the index variable q takes the values $t : -1 : 1$. One way to perform the computation is to execute:

```
for $q = t : -1 : 1$
    $x = \left(I_{n^{t-q}} \otimes P_{n^q,n}\right) x$
end
```

4.1.2 The DFT

The Discrete Fourier Transform (DFT) of a vector x of length n is defined as the matrix-vector product $F_n x$ where F_n is the highly structured n-by-n matrix:

$$(F_n)_{pq} = \omega_n^{pq} \; p, q = 0, \cdots, n-1 \tag{4.2}$$

where $\omega_n = \exp(-\frac{2\pi pqi}{n})$.

If $n = pm$ then

$$F_n = (F_p \otimes I_m) D_{p,m} (I_p \otimes F_m) P_{n,p} \tag{4.3}$$

where $D_{p,m}$ is a diagonal matrix of weights with

$$D_{p,m}(j,j) = w_{pm}^{(j \bmod m)\lfloor j/m \rfloor} \text{ for } j = 0, \cdots, pm-1 \tag{4.4}$$

and $P_{n,p}$ is the *stride permutation* matrix, see § 1.3.

Define matrix

$$B_{n,m} = (F_n \otimes I_m) D_{n,m}$$

the butterfly matrix. Using (4.3), well known FFT algorithms can be related to various factorizations of F_n.

The *Cooley-Tukey* version of the FFT corresponds to the factorization

$$F_{n^t} = \left(\prod_{q=t,t-1}^{1} (I_{n^{t-q}} \otimes B_{n,n^{q-1}}) \right) R_{n^t,n} \tag{4.5}$$

where $R_{n^t,n}$ is the digit reversal permutation matrix.

The *Stockham* FFT factorization corresponds to

$$F_{n^t} = \prod_{q=t,t-1}^{1} ((B_{n,n^{q-1}} P_{n^q,n}) \otimes I_{n^{t-q}}). \tag{4.6}$$

The *transposed Stockham* factorization, as the name reveals is the transpose of the above

$$F_{n^t} = \prod_{q=1}^{t} ((B_{n,n^{q-1}} P_{n^q,n})^T \otimes I_{n^{t-q}}). \tag{4.7}$$

The *Pease* factorization is

$$F_{n^t} = \left(\prod_{q=t,t-1}^{1} ((B_{n,n^{q-1}} \otimes I_{n^{t-q}})P_{n^t,n}) \right) R_{n^t,n}. \tag{4.8}$$

The *Gentleman-Sande* version of the Cooley-Tukey factorization corresponds to

$$F_{n^t} = R_{n^t,n} \left(\prod_{q=t,t-1}^{1} (I_{n^{t-q}} \otimes B_{n,n^{q-1}}) \right)^T \tag{4.9}$$

See [VL92] for derivations of these and other factorizations of F_n..

If p and m are relatively prime, then

$$F_n = \Gamma_{p,n}(F_p \otimes F_m)\Upsilon_{p,n}^T \tag{4.10}$$

where $\Gamma_{p,n}$ and $\Upsilon_{p,n}^T$ are n-by-n permutation matrices based on number-theoretic properties as defined in [VL92], page 189.

In the case that n is prime, there is the *Rader* factorization

$$\mathrm{F_n = Q_n(r)} \begin{bmatrix} 1 & e^T \\ e & C \end{bmatrix} \mathrm{Q_n^T(r^{-1})} \tag{4.11}$$

where r is a primitive root of the set of indices $\{2, \cdots, n-1\}$, $Q_n(r)$ is a number-theoretic permutation matrix, e is a vector of ones and C is a circulant matrix defined by the vector $v_j = \omega_n^{r^j}$ for $j = 0{:}n-2$. The circulant matrix-times-vector product can be calculated using the FFT of length $n-1$, thus a non-prime length FFT. For an extensive presentation, the reader is directed to [VL92] and references therein.

4.1.3 Other Fast Transforms

The *Walsh-Hadamard* transform is defined by $x := W_{2^t}x$ where

$$\mathrm{W_{2^t}} = \bigotimes_{\mathrm{q=1}}^{\mathrm{t}} \mathrm{W_2} \text{ and } \mathrm{W_2} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{4.12}$$

The notation $\otimes_{q=1}^{t} A_q$ is a shorthand for $A_1 \otimes \cdots \otimes A_t$. It can be shown that

$$W_{2^t} = \prod_{q=1}^{t} (I_{2^{q-1}} \otimes W_2 \otimes I_{2^{t-q}}). \tag{4.13}$$

The *Haar* transform is defined by $x := H_{2^t} x$ where

$$\mathrm{H}_{2^{\mathrm{t}}} = \prod_{\mathrm{q}=1}^{\mathrm{t}} \left[\begin{array}{cc} (H_2 \otimes I_{2^{q-1}}) P_{2^q,2} & O \\ O^T & I_{2^t - 2^q} \end{array} \right] \tag{4.14}$$

with $H_2 = W_2 = F_2$. The factorization formula follows the multiresolution algorithm presented in [Mal89]. We present a derivation of this factorization as well as a generalization to cover the Daubechies wavelets from [Dau92] in §4.6.

The *Slant* transform is defined by $x := S_{2^t} x$ where

$$\mathrm{S_n} = \mathrm{M_n}(\mathrm{I_2} \otimes \mathrm{S_{n/2}}) \text{ and } \mathrm{S_2} = \frac{1}{\sqrt{2}} \left[\begin{array}{cc} 1 & 1 \\ 1 & -1 \end{array} \right]. \tag{4.15}$$

Matrix M is sparse. A compact way to define M is to consider it as two submatrices:

$$M_n(u,u) = \frac{1}{\sqrt{2}} \left[\begin{array}{cccc} 1 & 0 & 1 & 0 \\ a_n & b_n & -a_n & b_n \\ 0 & 1 & 0 & -1 \\ -b_n & a_n & b_n & a_n \end{array} \right]$$

where $u = [1, 2, n/2+1, n/2+2]$, and

$$M_n(u,u) = S_2 \otimes I_{n/2-2}.$$

for $u = [3 : n/2, n/2+3 : n]$. The values a_n and b_n are given by

$$a_{2n} = \sqrt{\frac{3n^2}{4n^2-1}}, \; b_{2n} = \sqrt{\frac{n^2-1}{4n^2-1}}.$$

A factorization of the Slant transform matrix of size 2^t is

$$S_{2^t} = \prod_{q=1}^{t} (I_{2^{q-1}} \otimes M_{2^{t-q+1}}). \tag{4.16}$$

The Slant transform is used in image processing [Pra91].

The *Hartley* transform is defined by $x := H_n x$ where

$$H_n(j,k) = \cos\left(\frac{2\pi jk}{n}\right) + \sin\left(\frac{2\pi jk}{n}\right) \text{ with } j,k = 0,\cdots,n-1.$$

Due to the similarity to the Fourier transform, it has a very similar factorization

$$H_{pm} = (H_p \otimes I_m) D_{p,pm} (I_p \otimes H_m)$$

where $D_{p,pm}$ is a block diagonal matrix with diagonal blocks that are sparse. The diagonal blocks have at most two non-zeros per row and column, (on the diagonal and anti-diagonal). The same factorizations known for the FFT also apply here. For example, the Cooley-Tukey factorization for the discrete Hartley transform is

$$H_{n^t} = \prod_{q=t,t-1}^{1} \left(I_{n^{t-q}} \otimes \left((H_n \otimes I_{n^{q-1}}) D_{n,n^{q-1}}\right)\right) R_{n^t,n}. \tag{4.17}$$

Other variants result by following the other known factorizations like Stockham and Pease. The Hartley transform can be regarded as a real analog of the Fourier transform [VL92].

In the preceding, we only listed here one dimensional transform algorithms. Multidimensional transforms are Kronecker products of the above formulae. For example, suppose that the 2-dimensional FFT of the m-by-n matrix X is needed. If $x = \text{vec}(X)$ then

$$\text{vec}(F_m X F_n^T) = (F_n \otimes F_m)x.$$

For more on high dimesional transforms see, § 4.5.1.

4.2 Automatic Loops Nest Generation

In order to generate the loop nest, we define a language that describes the different matrix types found in fast transform algorithms and then construct a term rewriting system based in pattern matching. We provide a working experimental system in Mathematica that implements the grammar and transformation rules. We also provide back-ends to translate the resulting simplified expressions of the matrix grammar into Fortran, Fortran 90, C and MATLAB functions, that we execute in various architectures.

4.2.1 A Grammar for Matrix Formulae

We need a simple language to describe matrix formulae and simple programming constructs like assignments and loops. A computer language like MATLAB or Fortran 90 is perfect because they allow expressions with matrix arguments and the colon notation for indices. But here we will consider a constructor-based language to make our life easier in recognizing the various constructs without having to write a complicated parser. Figure 4.1 presents the context free grammar that describes this language, words in capitals are keywords (literals). $Indexname$ and $Variablename$ denote strings of characters, $AExpression$ is an algebraic expression involving Indexnames and integers. The meaning of those keywords is pretty direct. There are only four types of statements:

- $ASSIGN$ denotes an assignment statement of a vector,
- $LOOP$ denotes a sequential loop
- $PARALLEL$ denotes a parallel loop structure.
- SEQ denotes a sequence of exactly two statements.

Both loop constructs accept a sequence of statements. The sequential and parallel loops are defined by the triplet

$$(Indexname, Index, Statement)$$

where $Indexname$ is a symbol that names the loop-index variable, $Index$ stands for the range of the loop index, and $Statement$ stands for the statements in the

$$\begin{array}{rcl}
Statement & \rightarrow & SEQ(Statement, Statement) \\
 & | & ASSIGN(Vector, Expression) \\
 & | & LOOP(Indexname, Index, Statement) \\
 & | & PARALLEL(Indexname, Index, Statement) \\
Vector & \rightarrow & VEC(Variablename, Index) \\
Index & \rightarrow & IDX(AExpression, AExpression, AExpression) \\
Expression & \rightarrow & Vector \\
 & | & TIMES(Matrix, Vector) \\
 & | & DOTSTAR(Vector, Vector) \\
Matrix & \rightarrow & F(AExpressionn) \\
 & | & P(AExpression, AExpression) \\
 & | & I(AExpression) \\
 & | & D(AExpression, AExpression) \\
 & | & TRANSP(Matrix) \\
 & | & TIMES(Matrix, Matrix) \\
 & | & KRON(Matrix, Matrix) \\
 & | & PROD(Indexname, Index, Matrix)
\end{array}$$

Figure 4.1: A grammar for the simplified matrix language used to transform a matrix expression into a sequence of loops and assignments.

loop nest.

An index is defined by the keyword IDX and a triplet of arithmetic and symbolic expressions. The first expression defines the initial value of the index, the second is the step or stride of the index and the third one is the final value of the index. We require that the final value be exact and not an upper bound because otherwise we cannot define an index algebra to manipulate symbolic indices. For example the index $IDX(1,2,10)$ is illegal because 10 is not a feasible value for this index.

A vector is defined by the VEC keyword and the pair $(Variablename, Index)$ that names the vector, and an index that specifies the corresponding positions of the vector. The Matlab vector expression `v(2:2:100)` is equivalent to

$$VEC(v, IDX(2,2,100)).$$

For simplicity, the assignment statement only assigns values to vectors. Thus an expression is only a matrix-vector product defined by the prefix $TIMES$ or the Hadamard product of two vectors defined by $DOTSTAR$. The assignment statement `x(1:2:63) = y(0:2:62)` is written as

$$ASSIGN(VEC(x, IDX(1,2,63)), VEC(y, IDX(0,2,62))).$$

Other vector operations can be defined as matrix-vector products where the matrices have a special structure. However, no other operation is needed in the domain of FFT algorithm development. Even the Hadamard product of two vectors can be written as the product of a diagonal matrix with the elements of the first vector times the second. We added the construct to the grammar anyway in order to have only diagonal matrices of twiddle factors.

The following keywords denote various matrices and operations:

- $TRANSP$ denotes the transpose of a matrix.
- $TIMES$ denotes matrix multiplication.
- $PROD$ is a generalization of matrix multiplication, and corresponds to the well known symbol $\prod$. It requires the triplet $(Indexname, Index, Matrix)$ where $Indexname$ is a symbol that names the index, $Index$ defines the range of the index variable and $Matrix$ stands for the matrix expression.
- $KRON$ denotes the Kronecker product of two matrices.
- $I(n)$ stands for the identity matrix I_n.
- $F(n)$ stands for the DFT matrix F_n.
- $P(n,p)$ is the stride permutation matrix $P_{n,p}$ from (1.8).
- $D(p,m)$ is the diagonal matrix of weights from (4.4).

Matrices are not computed in this language, rather they are seen as specifications of how to compute with them. For example, in the expression

$$ASSIGN(x, TIMES(TIMES(A,B),y))$$

that defines the statement $x = (AB)y$, we do not intend to ever multiply the two matrices together. Instead we transform this expression to $x = A(Bx)$ and then to the sequence $x = By;\ x = Ax$. Finally the applications of matrices B and A have to be translated into the appropriate loops and vector operations.

To illustrate the grammar, let us express the combine phase of the Cooley-Tukey factorization (4.5):

$$x = \left(\prod_{q=t,t-1}^{1} (I_{n^{t-q}} \otimes B_{n,n^{q-1}}) \right) x$$

where B, the butterfly matrix, is defined as

$$B_{n,L_*} = (F_n \otimes I_{L_*}) D_{n,L_*}.$$

At the highest level we are assigning an expression to a vector

```
ASSIGN(
    $Vector$,
    $Expression$,
    )
```

The vector in question is called x and its index ranges from 0 to $n^t - 1$ in steps of 1:

```
VEC(
    $x$,
    IDX(0,1,$n^t - 1$),
    )
```

The *Expression* is a matrix-vector product:

```
TIMES(
    $Matrix$,
    $Vector$,
    )
```

The *Vector* here is the same x. Taking stock of the development so far, we substitute the *Vector* and *Expression* in the initial assignment and get:

```
ASSIGN(
    VEC(
        $x$,
        IDX(0,1,$n^t - 1$),
        ),
    TIMES(
        $Matrix$,
        VEC(
            $x$,
            IDX(0,1,$n^t - 1$),
            ),
        ),
    )
```

The matrix here is a product:

```
PROD(
    $Indexname$,
    $Index$,
        $Matrix$,
        )
```

The *Indexname* is q and steps from t on down to 1 in steps of -1 and so becomes

```
    PROD(
        $q$,
        IDX($t$,-1,1),
        $Matrix$,
        )
```

The *Matrix* here is a Kronecker product:

```
    KRON(I($n^{t-q}$), B($n$,$n^{q-1}$))
```

Therefore the complete assignment statement is

```
    ASSIGN(
        VEC(
            $x$,
            IDX(0,1,$n^t - 1$),
            ),
        TIMES(
            PROD(
                $q$,
                IDX($t$,-1,1),
                KRON(I($n^{t-q}$), B($n$,$n^{q-1}$)),
                ),
            VEC(
                $x$,
                IDX(0,1,$n^t - 1$),
                ),
            ),
        )
```

One more example is the digit reversal application. The assignment as matrix-vector product is

$$x = \left(\prod_{q=0}^{t-2} (I_{n^{t-2-q}} \otimes P_{n^{q+2},n}) \right) x$$

and corresponds to

$$\begin{aligned}&ASSIGN(VEC(x, IDX(0,1,n^t-1)),\\ &\qquad TIMES(PROD(q,0,1,t-2,KRON(I(n^{t-2-q}),P(n^{q+2},n))),\\ &\qquad\qquad VEC(x, IDX(0,1,n^t-1)))).\end{aligned}$$

Irreducible Form of Strings

The rewriting rules, as we will see next, are based on various linear algebra identities. However, the identities do not have a direction of application. When we say that

$$A_m \otimes B_n = (A_m \otimes I_n)(I_m \otimes B_n)$$

for square matrices A and B of sizes m and n respectively, we can define a rule that transforms all instances of the left-hand-side of the equality to the right-hand-side and vice-versa. Going from right to left in the above example, it might seem as a reduction or simplification. However going from left to right, it can also introduce new opportunities for further transformations.

In a term rewriting system, you have to define when the rewrites terminate, i.e., when the expression that is manipulated has reached an *irreducible form*. We say that a string is in irreducible form, if no changes can be introduced even if we keep applying the rewriting rules. In our case, a string from the matrix language is in irreducible form if the following three conditions hold:

1. there is no matrix expression other than $F(n)$ where n is the size of the butterfly.

2. All loops are nested.

3. No assignment sequence contains an unnecessary step.

It is possible for the same expression to have more than one irreducible form, since it depends upon the order in which the rewriting rules are applied. For example consider the jk and kj versions of the Stockham FFT algorithm in Figure 4.2

A Pattern-Matching Term Rewriter

The Mathematica programming environment is used to implement a source to source compiler that generates efficient code from the high-level matrix description of fast transforms.

The elementary tools for building such an environment are *transformation rules* based on *pattern matching*. A transformation rule is a rule that describes how to change expressions from one form to another. A rule has a left-hand side that contains patterns and a right-hand side that contains the directions of what to do with the expressions that matched the left-hand side. A pattern is a class of expressions that have the same structure.

Mathematica has many kinds of built-in transformation rules for arithmetic, simplifying algebraic expressions, and differentiation. Moreover it allows you to define your own rules for manipulating new functions and objects.

When Mathematica evaluates an expression, it goes on applying transformation rules until it can find no more rules to apply.

The manual of Mathematica [Wol88] is vague on the order or rule applications.

> When you give a sequence of transformation rules, Mathematica arranges them so that the most specific rules come first. Then, when it tries to apply the rules, it tests them in order. If a specific rule does apply, then it is used; only if none of the specific rules apply will the later, more general, rules be used. If Mathematica cannot decide whether a new rule is more or less specific than the ones it already knows, it puts the new rule at the end of the list. (...) The order in which Mathe-

```
function x = $jk$-Stockham($x$,$n$,$t$)
for $q = 1 : t$
    $L = n^q; r = n^{t-q}; L_* = L/n; r_* = nr; w = \mathrm{diag}(D_{n,L_*})$
    $y = x$
    forall $j = 0 : L_* - 1$
        forall $k = 0 : r - 1$
            $x(k + jr : n^{t-1} : k + jr + n^{t-1}(n-1)) = F_n \cdots$
            $(w(j : L_* : j + L_*(n-1)) \odot y(k + jr : n^{t-1} : k + jr + n^{t-1}(n-1))$
        end
    end
end

function x = $kj$-Stockham($x$,$n$,$t$)
for $q = 1 : t$
    $L = n^q; r = n^{t-q}; L_* = L/n; r_* = nr; w = \mathrm{diag}(D_{n,L_*})$
    forall $k = 0 : r - 1$
        $y = x(k : r : k + r(L-1))$
        forall $j = 0 : L_* - 1$
            $x(k + jr : n^{t-1} : k + r(L-1)) = F_n \cdots$
            $(w(j : L_* : j + L_*(n-1)) \odot y(jn : (j+1)n - 1))$
        end
    end
end
```

Figure 4.2: Versions jk and kj of the Stockham FFT algorithm.

matica puts the rules does not depend on the order in which you type [them].

A rule is expressed as a function. For example `f[x_] := 2 x -1` is the definition of a function that doubles the value of its single argument and then subtracts one from the product. The underscore after the name of the variable `x_` on the left-hand side of the rule declares `x_` as the free variable that can by substituted by anything. Variable `x_` is the simplest pattern possible that stands for everything. Actually a single underscore is the simplest pattern with no name. We name a pattern so we can refer to the expression that is matched at the right hand side of the rule, or in order to define a more complex pattern with constraints. For example, the pattern `f[x_, x_]`, is only matched with expressions with identical arguments like `f[n,n]` or `f[3,3]` but not with `f[a,b]` because if the first `x` gets instantiated with `a`, the second `x` cannot get instantiated with `b`. Abelson and Sussman offer some practical information on pattern matching and unification [AS85]. You can think of a pattern as a template where the "gaps" or "blanks" or "holes" can be filled with arbitrary expressions. However, there are provisions that allow you to restrict the possible substitutions of a pattern,

- `_` any expression,
- `x_` any expression to be named x,
- `x_ + y_` any summation of terms to be named: x the first one and y the rest of the terms,
- `x_ y_` any product of terms to be named: x the first one and y the rest of the terms,

or even define default values:

- `x_ + y_.` the default for `y` is 0
- `x_ y_.` the default for `y` is 1
- `x_ ^ y_.` the default for `y` is 1

We use the transformation rules to express mathematical relations in order to simplify algebraic expressions. For example, the rules that simplify transposed matrix expressions are:

```
TRANSP[F[n_]] := F[n]
TRANSP[P[m_,n_]] := P[m,m/n]
TRANSP[I[n_]] := I[n]
TRANSP[D[n_,p_]] := D[n,p]
TRANSP[TIMES[A_,B_]] := TIMES[TRANSP[B],TRANSP[A]]
TRANSP[PROD[v_, IDX[f_,s_,l_], M_]] := PROD[v, IDX[l,-s,f], TRANSP[M]]
TRANSP[KRON[A_,B_]] := KRON[TRANSP[A],TRANSP[B]]
```

The rules describe the well-known identities

$$\begin{aligned}
F_n^T &= F_n \\
P_{m,n}^T &= P_{m,m/n} \\
I_n^T &= I_n \\
D_{n,p}^T &= D_{n,p} \\
(AB)^T &= B^T A^T \\
\left(\prod_{q=1}^{t} A_q\right)^T &= \prod_{q=t,t-1}^{1} A_q^T \\
(A \otimes B)^T &= A^T \otimes B^T.
\end{aligned}$$

In general, rules come in the form

$$\text{pattern} := \text{replacement-value}$$

and these transformation rules are applied automatically. However, given the fuzziness of the priorities that Mathematica assigns to rules, it is more flexible to name a rule using the form

rule-name = pattern := replacement-value

and later-on apply the named rule at an expression via the form

expression /. rule-name

Since mathematical identities do not have a direction of application, the above scheme permits the definition of an identity as a pair of rules that one undoes the result of the other. By naming the rules and using them accordingly, we avoid falling into an infinite loop. For example, one might define the simplification

```
TIMES[KRON[I[n1_],F[n2_]], KRON[F[n1_],I[n2_]]] := KRON[F[n1],F[n2]]
```

On the other hand, in a multidimensional setting, the rule

```
KRON[F[n1],F[n2]] := TIMES[KRON[I[n1_],F[n2_]], KRON[F[n1_],I[n2_]]]
```

transforms a 2-dimensional FFT into a sequence of one dimensional FFTs. Such a definition introduces an infinite loop. By giving names to rules and thus controlling when they are applied resolves this problem.

We define the transformation rules using the form

rule-name = pattern :> replacement-value

because this is the way to set a delayed definition, that is, the right hand side of the rule is not evaluated until the moment of the application of the rule.

The scheme

expression //. rule-names

goes on to trying to apply the rules in order, until none of them apply any more. Thus by naming and grouping the rules, one can define "strategies" –sequences of rules– that can be applied to expressions to reach a desired result. With the named rules and controlled rule application we also avoid another situation. When an expression contains nested objects like `f[g[x]]`, Mathematica always tries to apply the rules associated with the innermost objects. However, for our approach, an outer to inner reduction or a mixed reduction is preferable.

Finally we can restrict the application of a transformation rule, except by the strict pattern specification, by adding a condition. The form is

$$\text{pattern} \;:>\; \text{replacement-value} \;;\; \text{condition}$$

that means "change pattern with the replacement-value whenever condition holds". For example the matrix transpose rule we gave before for the stride permutation matrix is safer as

```
TRANSP[P[m_,n_]] := P[m,m/n] \; Mod[m,n] == 0
```

4.2.2 Program Transformation Rules

Here are the major rules, based on matrix identities involving the Kronecker product, the stride permutation matrix and compiler techniques that transform a matrix expression into efficient code consisting only of loops and assignment statements.

R1 Matrix-Matrix-Vector Multiplication: Let A and B two square matrices, then the assignment $y = ABx$ can be written as a sequence of assingments

$$y = Bx$$
$$y = Ay$$

Here we assume that both matrices are square. The required pattern matching rule is given by:

$$ASSIGN[\mathbf{x}, TIMES[\mathbf{A}, TIMES[\mathbf{B}, \mathbf{y}]]] :=$$
$$SEQ[ASSIGN[\mathbf{x}, TIMES[\mathbf{B}, \mathbf{y}]],$$
$$ASSIGN[\mathbf{x}, TIMES[\mathbf{A}, \mathbf{x}]]]$$

where the items in bold face are pattern variables and the rest are literals.

R2 Π Product: The expression $x = (\prod_{j=j_1}^{j_2} A_j)x$ can be transformed to

```
for j=j_2:-1:j_1
    $x = A_j x$
endfor
```

Note the loop-index reversal, in order to associate the multiplications to the right and thus have a sequence of matrix-vector products. This transformation rule is a generalization of the Matrix-Matrix-Vector multiplication rule.

$$ASSIGN[\mathbf{x}, TIMES[PROD[\mathbf{q}, IDX[\mathbf{f}, \mathbf{s}, \mathbf{l}], \mathbf{M}], \mathbf{x}]] :=$$
$$LOOP[\mathbf{q}, IDX[\mathbf{l}, -\mathbf{s}, \mathbf{f}], ASSIGN[\mathbf{x}, TIMES[\mathbf{M}, \mathbf{x}]]]$$

R3 Reindexing: Suppose we have a vector $y = x(f_1 : s_1 : l_1)$ where f_1, s_1, l_1 are algebraic expressions denoting the first index f_1, the step s_1 and the last index l_1. Any subvector $z = y(f_2 : s_2 : l_2)$ can be expressed as a subvector of the original vector x as $z = x(f_1 + f_2 s_1 : s_1 s_2 : f_1 + l_2 s_1)$. The corresponding rule is:

$$REINDEX[VEC[\mathbf{x}, IDX[\mathbf{f1}, \mathbf{s1}, \mathbf{l1}]], IDX[\mathbf{f2}, \mathbf{s2}, \mathbf{l2}]] :=$$
$$VEC[\mathbf{x}, IDX[\mathbf{f1} + \mathbf{f2} * \mathbf{s1}, \mathbf{s1} * \mathbf{s2}, \mathbf{f1} + \mathbf{l2} * \mathbf{s1}]]$$

R4 $I \otimes A$: Let `A_m` be a square matrix of size m and two vectors x and y of length n with $n = pm$. Then the code that implements the matrix vector product $x = (I_p \otimes A_m)y$ where x, y are vectors of length n, is:

```
forall j=0:p-1
  x(j*m:(j+1)*m-1) = A_m * x(j*m:(j+1)*m-1)
endfor
```

Moreover, this loop can be executed in parallel. The corresponding pattern matching rule is:

$$\begin{aligned}
ASSIGN[\mathbf{x}, TIMES[KRON[I[\mathbf{p}], \mathbf{A}], \mathbf{y}]] :> \\
(v = newloopindex(); \\
sz = SIZE[\mathbf{A}]; \\
newidx = IDX[v * sz, 1, (v+1) * sz - 1]; \\
PARALLEL[v, IDX[0, 1, \mathbf{p} - 1], \\
ASSIGN[REINDEX[\mathbf{x}, newidx], \\
TIMES[\mathbf{A}, REINDEX[\mathbf{y}, newidx]]]])
\end{aligned}$$

R5 $A \otimes I$: Similarly, the code implementing $x = (A_p \otimes I_m)x$ is:

```
forall j=0:m-1
  x(j:m:j+m*(p-1)) = A_p * x(j:m:j+m*(p-1))
endfor
```

The pattern matching rule is

$$\begin{aligned}
ASSIGN[\mathbf{x}, TIMES[KRON[\mathbf{M}, I[\mathbf{m}]], VEC[\mathbf{y}, \mathbf{idx}]]] :> \\
(v = newloopindex(); \\
sz = SIZE[M];
\end{aligned}$$

$$newidx = IDX[v, m, v + m * (sz - 1)];$$
$$PARALLEL[v, IDX[0, 1, m - 1],$$
$$ASSIGN[REINDEX[x, newidx],$$
$$TIMES[M, REINDEX[VEC[y, idx], newidx]]]])$$

Both rules are direct consequences of the Kronecker Product property shown in (1.6).

Rules **R6**-**R10** require very complicated patterns and they are applied conditionally. In order to mentain a high level of presentation, we only provide a simple conceptual description of the rules here. For the exact implementation of a rule, see Appendix B where the complete source code of the rules is listed.

R6 Stride Permutation: Given that $n = pm$, the Stride Permutation $x = P_{n,p}x$ is equivalent to the following reindexing with the assistance of a loop:

```
y = x
forall j=0:p-1
  x(j*m:(j+1)*m-1) = y(j:p:j+(m-1)*p)
endfor
```

An alternate way to transform the stride permutation is the "dual" to the above loop:

```
y = x
forall j=0:m-1
  x(j:m:j+m*(p-1)) = y(j*p:(j+1)*p-1)
endfor
```

These loops can be executed in parallel. Note that the Stride Permutation rules are the only transformations that require temporary vector storage, the transformation cannot be done in place. In addition to the above two transformation rules, we have the general index transformation that is based on the following lemma: If $y = P_{n,p}x$ then

$$y(j) = x((j \bmod (n/p)) * p + \lfloor j/(n/p) \rfloor) \tag{4.18}$$

This formula is useful in a situation like the following where a stride permutation is adjacent to a parallel loop:

```
x = P(n,p) x
forall j=somerange
  an assignment with x(idx) in the right hand side
endfor
```

that can be transformed into

```
y = x
forall j=somerange
  x(idx) = y(idx2)
  an assignment with x(idx) in the right hand side
endfor
```

only when `idx2` can be determined using (4.18). This is a potentially difficult calculation to handle automatically when all operands are symbolic. In the current implementation of our system, we handle this problem using an identity look-up table. For the way this is implemented please see the source code in Appendix B.

R7 Weight Multiplication: The weight multiplication $x = Dx$ can be written as as an element-wise multiplication of the corresponding weights w (where $w = \mathrm{d}iag(D)$), that is $x = w.*x$. Similarly to the Stride Permutation transformation, the weight multiplication can be "pushed" into a parallell loop.

R8 Loop Interchange: Two nested parallel loops can be interchanged.

R9 Loop Join: Two parallel loops forming a sequence can be joined into one parallel loop iff they have the same range and their bodies assign to the same variable with the same index function i.e.:

```
forall j=somerange
  x(idx) = some-right-hand-side
endfor
forall j=somerange
  x(idx) = some-other-right-hand-side
endfor
```

can be transformed into

```
forall j=somerange
  x(idx) = some-right-hand-side
  x(idx) = some-other-right-hand-side
endfor
```

The index function `idx` has to be identical to both assignments. The right hand sides of the assignments cannot contain references to vector `x`.

R10 Join Assignments: This is the reverse rule to Matrix-Matrix-Vector multiplication rule. That is, the sequence of assignments:

```
x = z
y = f(x)
```

can be substituted by the single assignment `y = f(z)`.

These are the major term rewriting rules. Many more minor rules handle the automatic index and variable name generation, simplification of linear algebra expressions (for example transposition of a product of matrices), index operations and other secondary tasks.

The secondary rules are active all the time and apply whenever there is an instance that matches their patterns. For example, if at any moment an expression of the form IDX(f_1, s_1, l_1)+IDX(f_2, s_2, l_2) is generated, it will be substituted by IDX$(f_1 + f_2, s_1 + s_2, l_1 + l_2)$ without the need of interaction with the user. In the same way that all symbolic manipulation systems will substitute $x + x$ with $2x$.

On the other hand, the major term rewriting rules require the interaction of the user. For practicality, major rules are grouped in sets of rules that have similar scope and that can cooperate. Rules **R1**, **R2**, **R4** and **R5** are expanding rules. Rules **R6**, **R7**, **R9** and **R10** are contracting. While the loop interchange rule **R8** neither introduces new expressions nor reduces existing expressions of the algorithm under transformation.

The choice of the grouping of rules and the order of the application of the groups, define a transformation strategy. Preparing a transformation strategy is a delicate task even for an expert user. In general, the same transformation strategy can handle all problems of the same dimension, different transformation strategies are needed for multidimensional problems. A less experienced user can just use the ready-made transformation strategies we provide, or in interactive mode, via trial and error, apply one rule at a time until the desired outcome is achieved. The following is a transcript of such an interactive session that displays the term rewriting system in action.

An Example

Here we present step by step the transformation of the matrix factorization corresponding to the transposed Stockham algorithm into a loop nest. The notation we use is that of MATLAB . Actually if one defines the MATLAB functions implementing the matrices P, D, B and F one can execute each stage of the transformation in order to verify the preservation of correctness.

We start with the actual matrix factorization of the corresponding algorithm. Here we consider the factorization of Transposed Stockham. We express the initial statement as an assignment, because this is what we want the final program to achieve. Again remember that we assume that the length of the vector x is n^t.

$$x = (\prod_{q=t,t-1}^{1}(I_r \otimes B_{n,L*})(P^T_{r*,n} \otimes I_{L*}))x$$

where $L = n^q$, $r = n^{t-q}$, $L_* = L/n$, and $r_* = n * r$. The shorthand notation to shorten the display of expressions, is identical to the one used in [VL92].

The only rule that applies at this point is **R2** which transforms the product of matrices into a sequential loop.

```
for $q = 1 : t$
    $L = n^q; r = n^{t-q}; L_* = L/n; r_* = nr;$
    $x = (I_r \otimes B_{n,L*})(P^T_{r*,n} \otimes I_{L*})x$
end
```

The next rule that we can apply is **R1** which splits the assignment statement into two due to matrix-matrix-vector product.

```
for $q = 1 : t$
    $L = n^q; r = n^{t-q}; L_* = L/n; r_* = nr;$
    $x = (P^T_{r*,n} \otimes I_{L*})x$
    $x = (I_r \otimes B_{n,L*})x$
end
```

The two rules that transform a Kronecker product with the identity matrix into a parallel loop can be applied. There is no interference in their application, so we just perform the transformation by the rule **R5** on the first occurrence.

```
for $q = 1 : t$
    $L = n^q; r = n^{t-q}; L_* = L/n; r_* = nr;$
    forall $j = 0 : L_* - 1$
        $x(j : L_* : j + L_*(r_* - 1)) = P^T_{r*,n} x(j : L_* : j + L_*(r_* - 1))$
        end
        $x = (I_r \otimes B_{n,L_*})x$
    end
```

At this point we perform a linear algebra simplification that could have been done at any prior or subsequent point. We remove the transposition of matrix P using the identity involving stride permutations

$$P_{n,p}^T = P_{n,\frac{n}{p}}$$

since $\frac{r_*}{n} = r$.

```
    for $q = 1 : t$
        $L = n^q; r = n^{t-q}; L_* = L/n; r_* = nr;$
        forall $j = 0 : L_* - 1$
            $x(j : L_* : j + L_*(r_* - 1)) = P_{r_*,r}x(j : L_* : j + L_*(r_* - 1))$
        end
        $x = (I_r \otimes B_{n,L_*})x$
    end
```

We can reduce matrix B_{n,L_*} by its equivalent expression. Again, this substitution can be done at any point of this transformation. The choice to do it now is not the best, but it is possible that this can happen and it does happen in the automatic program transformation setting, where equivalence reductions are executed first.

```
    for $q = 1 : t$
        $L = n^q; r = n^{t-q}; L_* = L/n; r_* = nr;$
        forall $j = 0 : L_* - 1$
            $x(j : L_* : j + L_*(r_* - 1)) = P_{r_*,r}x(j : L_* : j + L_*(r_* - 1))$
        end
        $x = (I_r \otimes ((F_n \otimes I_{L_*})D_{n,L_*}))x$
    end
```

Another trivial substitution is the one of exchanging the diagonal matrix D_{n,L_*} with a vector w of its diagonal elements. The operator diag() when its single argument is a matrix, it returns a vector containing the diagonal elements of that matrix. When the argument is a vector, diag() returns a diagonal matrix with that vector on the diagonal. We do not address the issue of precalculation of twiddle factors, it is covered appropriately in [VL92].

```
for $q = 1 : t$
    $L = n^q; r = n^{t-q}; L_* = L/n; r_* = nr; w = \mathrm{diag}(D_{n,L_*})$
    forall $j = 0 : L_* - 1$
        $x(j : L_* : j + L_*(r_* - 1)) = P_{r_*,r}x(j : L_* : j + L_*(r_* - 1))$
    end
    $x = (I_r \otimes ((F_n \otimes I_{L_*})\mathrm{diag}(w)))x$
end
```

Next comes a non-trivial transformation resulting from the application of the **R4** rule. The result is a new parallel loop.

```
for $q = 1 : t$
    $L = n^q; r = n^{t-q}; L_* = L/n; r_* = nr; w = \mathrm{diag}(D_{n,L_*})$
    forall $j = 0 : L_* - 1$
        $x(j : L_* : j + L_*(r_* - 1)) = P_{r_*,r}x(j : L_* : j + L_*(r_* - 1))$
    end
    forall $k = 0 : r - 1$
        $x(k * L : (k + 1) * L - 1) = ((F_n \otimes I_{L_*})\mathrm{diag}(w))x(k * L : (k + 1) * L - 1)$
    end
end
```

The assignment statement in the last parallel loop matches the **R1** rule that splits the assignment into a sequence of two simpler assignments.

```
for $q = 1 : t$
    $L = n^q; r = n^{t-q}; L_* = L/n; r_* = nr; w = \mathrm{diag}(D_{n,L_*})$
    forall $j = 0 : L_* - 1$
        $x(j : L_* : j + L_*(r_* - 1)) = P_{r_*,r}x(j : L_* : j + L_*(r_* - 1))$
    end
    forall $k = 0 : r - 1$
        $x(k * L : (k + 1) * L - 1) = \mathrm{diag}(w)x(k * L : (k + 1) * L - 1)$
        $x(k * L : (k + 1) * L - 1) = (F_n \otimes I_{L_*})x(k * L : (k + 1) * L - 1)$
    end
end
```

The first of the two new assignments gets the product of a diagonal matrix times a vector, such a product can be reduced into the pointwise multiplication of two vectors.

```
for $q = 1 : t$
    $L = n^q; r = n^{t-q}; L_* = L/n; r_* = nr; w = \text{diag}(D_{n,L_*})$
    forall $j = 0 : L_* - 1$
        $x(j : L_* : j + L_*(r_* - 1)) = P_{r_*,r} x(j : L_* : j + L_*(r_* - 1))$
    end
    forall $k = 0 : r - 1$
        $x(k * L : (k+1) * L - 1) = w \odot x(k * L : (k+1) * L - 1)$
        $x(k * L : (k+1) * L - 1) = (F_n \otimes I_{L_*}) x(k * L : (k+1) * L - 1)$
    end
end
```

The second new assignment matches the **R5** rule, and another parallel loop is generated.

```
for $q = 1 : t$
    $L = n^q; r = n^{t-q}; L_* = L/n; r_* = n * r; w = \text{diag}(D_{n,L_*})$
    forall $j = 0 : L_* - 1$
        $x(j : L_* : j + L_*(r_* - 1)) = P_{r_*,r} x(j : L_* : j + L_*(r_* - 1))$
    end
    forall $k = 0 : r - 1$
        $x(k * L : (k+1) * L - 1) = w \odot x(k * L : (k+1) * L - 1)$
        forall $j = 0 : L_* - 1$
            $x(k * L + j : L_* : k * L + (L - 1)) = F_n \cdots$
            $x(k * L + j : L_* : k * L + (L - 1))$
        end
    end
end
```

The name of the loop index is chosen according to *tradition*. Indices with limits related to t get the name q, L or L_* get the name j, r or r_* get the name k, in order to strengthen the association with [VL92].

The next step is to "push" the twiddle factor multiplication into the parallel loop with the help of rule **R7**. It is done by reindexing the vectors in the twiddle factor multiplication assignment statement to agree with the next assignment.

```
for $q = 1 : t$
    $L = n^q; r = n^{t-q}; L_* = L/n; r_* = nr; w = \text{diag}(D_{n,L_*})$
    forall $j = 0 : L_* - 1$
        $x(j : L_* : j + L_*(r_* - 1)) = P_{r_*,r} x(j : L_* : j + L_*(r_* - 1))$
        end
        forall $k = 0 : r - 1$
            forall $j = 0 : L_* - 1$
                $x(k * L + j : L_* : k * L + (L - 1)) = w(j : L_* : j + L_*(n - 1)) \odot \cdots$
                $x(k * L + j : L_* : k * L + (L - 1))$
                $x(k * L + j : L_* : k * L + (L - 1)) = F_n \cdots$
                $x(k * L + j : L_* : k * L + (L - 1))$
            end
        end
    end
```

We are trying to get the code into an irreducible form, that is to get rid of the permutation matrix and make a single loop nest. At first we change the order of the two parallel loops to bring the two loops with the same index j next to each other (rule **R8**).

```
    for $q = 1 : t$
        $L = n^q; r = n^{t-q}; L_* = L/n; r_* = nr; w = \mathrm{diag}(D_{n,L_*})$
        forall $j = 0 : L_* - 1$
            $x(j : L_* : j + L_*(r_* - 1)) = P_{r_*,r}x(j : L_* : j + L_*(r_* - 1))$
        end
        forall $j = 0 : L_* - 1$
            forall $k = 0 : r - 1$
                $x(k * L + j : L_* : k * L + (L - 1)) = w(j : L_* : j + L_*(n - 1)) \odot \cdots$
                $x(k * L + j : L_* : k * L + (L - 1))$
                $x(k * L + j : L_* : k * L + (L - 1)) = F_n \cdots$
                $x(k * L + j : L_* : k * L + (L - 1))$
            end
        end
    end
```

The second parallel loop nest (the one defined by indices j and k operates on subvectors of the vector operated on the first parallel loop, we therefore are allowed to join the two parallel loops corresponding to the j loop index (rule **R9**).

```
    for $q = 1 : t$
        $L = n^q; r = n^{t-q}; L_* = L/n; r_* = nr; w = \mathrm{diag}(D_{n,L_*})$
        forall $j = 0 : L_* - 1$
            $x(j : L_* : j + L_*(r_* - 1)) = P_{r_*,r}x(j : L_* : j + L_*(r_* - 1))$
            forall $k = 0 : r - 1$
            $x(k*L+j:L_*:k*L+(L-1)) = w(j:L_*:j+L_*(n-1)) \odot \cdots$
            $x(k*L+j:L_*:k*L+(L-1))$
            $x(k*L+j:L_*:k*L+(L-1)) = F_n \cdots$
            $x(k*L+j:L_*:k*L+(L-1))$
        end
    end
end
```

The **R6** rule applies next to give:

```
for $q = 1 : t$
    $L = n^q; r = n^{t-q}; L_* = L/n; r_* = nr; w = \mathrm{diag}(D_{n,L_*})$
    forall $j = 0 : L_* - 1$
        $y = x(j:L_*:j+L_*(r_*-1))$
        forall $k = 0 : r - 1$
            $x(k*L+j:L_*:k*L+(L-1)) = y(k:r:k+r*(n-1))$
            $x(k*L+j:L_*:k*L+(L-1)) = w(j:L_*:j+L_*(n-1)) \odot \cdots$
            $x(k*L+j:L_*:k*L+(L-1))$
            $x(k*L+j:L_*:k*L+(L-1)) = F_n \cdots$
            $x(k*L+j:L_*:k*L+(L-1))$
        end
    end
end
```

Finally, the rule **R10** applies twice to generate the following code that is in irreducible form:

```
    for $q = 1 : t$
        $L = n^q; r = n^{t-q}; L_* = L/n; r_* = nr; w = \mathrm{diag}(D_{n,L_*})$
        forall $j = 0 : L_* - 1$
            $y = x(j:L_*:j+L_*(r_*-1))$
            forall $k = 0 : r - 1$
                $x(k*L+j:L_*:k*L+(L-1)) = F_n \cdots$
                $(w(j:L_*:j+L_*(n-1)) \odot y(k:r:k+r*(n-1)))$
            end
        end
    end
```

4.2.3 Generated Codes

We successfully transform all fast Fourier and other transform algorithms. Here we list the input specification and the corresponding output for all transform algorithms that we tried.

The Cooley-Tukey factorization (4.5), without the digit reversal phase can directly be translated to the assignment

```
    $ASSIGN(VEC(x, IDX(0, 1, n^t - 1)),$
                $TIMES(PROD(q, t, -1, 1, KRON(I(n^{t-q}), B(n, n^{q-1}))),$
                        $VEC(x, IDX(0, 1, n^t - 1))))$
```

After applying the transformations we get the following, expressed in MATLAB - F90 like syntax

```
for $q = 1 : t$
    $L = n^q; r = n^{t-q}; L_* = L/n; r_* = nr; w = \mathrm{diag}(D_{n,L_*})$
    forall $k = 0 : r - 1$
        forall $j = 0 : L_* - 1$
            $x(kL + j : L_* : kL + j + L_*(n-1)) = F_n \cdots$
            $(w(j : L_* : j + L_*(n-1)) \odot x(kL + j : L_* : kL + j + L_*(n-1)))$
        end
    end
end
```

The Stockham factorization (4.7) corresponds to the assignment statement in the prefix language

```
$ASSIGN(VEC(x, IDX(0, 1, n^t - 1)),$
    $TIMES(PROD(q, t, -1, 1,$
                $KRON(TIMES(B(n, n^{q-1}), P(n^q, n)), I(n^{t-q}))),$
        $VEC(x, IDX(0, 1, n^t - 1))))$
```

and the resulting code is

```
    for $q = 1 : t$
        $L = n^q; r = n^{t-q}; L_* = L/n; r_* = nr; w = \mathrm{diag}(D_{n,L_*})$
        forall $k = 0 : r - 1$
            $y = x(k : r : k + r(L-1))$
            forall $j = 0 : L_* - 1$
                $x(k + jr : n^{t-1} : k + r(L-1)) = F_n \cdots$
                $(w(j : L_* : j + L_*(n-1)) \odot y(jn : (j+1)n - 1))$
            end
        end
    end
```

The transposed Stockham code generation has been described step-by-step in the previous section. Here again is the resulting code.

```
for $q = 1 : t$
    $L = n^q; r = n^{t-q}; L_* = L/n; r_* = nr; w = \text{diag}(D_{n,L_*})$
    forall $j = 0 : L_* - 1$
        $y = x(j : L_* : j + L_*(r_* - 1))$
        forall $k = 0 : r - 1$
            $x(k * L + j : L_* : k * L + (L - 1)) = F_n \cdots$
            $(w(j : L_* : j + L_*(n - 1)) \odot y(k : r : k + r * (n - 1)))$
        end
    end
end
```

The Pease factorization (4.8) without the bit reversal phase corresponds to

$$
\begin{aligned}
&ASSIGN(VEC(x, IDX(0, 1, n^t - 1)),\\
&\quad TIMES(PROD(q, IDX(t, -1, 1),\\
&\qquad\qquad TIMES(KRON(B(n, n^{q-1}), I(n^{t-q})), P(n^t, n))),\\
&\qquad VEC(x, IDX(0, 1, n^t - 1))))
\end{aligned}
$$

that generates the code

```
for $q = 1 : t$
    $L = n^q; r = n^{t-q}; L_* = L/n; r_* = nr; w = \text{diag}(D_{n,L_*})$
    $y = x$
    forall $k = 0 : r - 1$
        forall $j = 0 : L_* - 1$
            $x(k + jr : rL_* : k + jr + rL_*(n - 1)) = F_n \cdots$
            $(w(j : L_* : j + L_*(n - 1)) \odot y(kn + jr_* : n(k + 1) + jr_* - 1))$
        end
    end
end
```

Apparently the two parallel loops for j and k can be collapsed into a single parallel loop because the only expression they are involved in is $k + jr$.

The Gentleman-Sande framework of the Cooley-Tuckey FFT (4.9) can be written as

$$
\begin{aligned}
&ASSIGN(VEC(x, IDX(0, 1, n^t - 1)),\\
&\quad TIMES(TRANSP(PROD(q, t, -1, 1, KRON(I(n^{t-q}), B(n, n^{q-1})))),\\
&\qquad VEC(x, IDX(0, 1, n^t - 1))))
\end{aligned}
$$

that translates into the quite different code:

```
for $q = t : -1 : 1$
    $L = n^q; r = n^{t-q}; L_* = L/n; r_* = nr; w = \mathrm{diag}(D_{n,L_*})$
    forall $k = 0 : r - 1$
        forall $j = 0 : L_* - 1$
            $x(j + kL : L_* : j + kL + L_*(n-1)) = w(j : L_* : j + L_*(n-1)) \cdots$
            $\odot(F_n x(j + kL : L_* : j + kL + L_*(n-1)))$
        end
    end
end
```

The Walsh-Hadamard transform (4.13) is written as

```
$ASSIGN(VEC(x, IDX(0, 1, 2^t - 1)),$
        $TIMES(PROD(q, IDX(1, 1, t), KRON(I(2^{q-1}), W(2), I(2^{t-q}))),$
                $VEC(x, IDX(0, 1, 2^t - 1))))$
```

that translates into a very familiar-looking:

```
for $q = t : -1 : 1$
    $L = 2^q; r = 2^{t-q}; L_* = L/2; r_* = 2r$
    forall $j = 0 : L_* - 1$
        forall $k = 0 : r - 1$
            $x(k + jr_* : r : k + r + jr_*) = W_2 x(k + jr_* : r : k + r + jr_*)$
        end
    end
end
```

that requires exactly $n \log_2 n$ additions.

The Haar tranform (4.14) is given in the form of a loop:

```
$LOOP(q, IDX(t, -1, 1),$
    $ASSIGN(VEC(x, IDX(0, 1, n^q - 1)),$
        $TIMES(TIMES(KRON(H(n), I(n^{q-1})), P(n^q, n)),$
            $VEC(x, IDX(0, 1, n^q - 1)))))$
```

The resulting code is

```
for $q = t : -1 : 1$
    $L = 2^q; r = 2^{t-q}; L_* = L/2; r_* = 2r$
    $y = x(0 : L - 1)$
```

```
        forall $j = 0 : L_* - 1$
            $x(j : L_* : j + L_*) = H_2 y(2j : 2j + 1)$
        end
    end
```

The Slant transform factorization (4.16) is written as the assignment statement:

```
    $ASSIGN(VEC(x, IDX(0, 1, 2^t - 1)),$
                $TIMES(PROD(q, IDX(1, 1, t), KRON(I(2^{q-1}), M(2^{t-q+1}))),$
                        $VEC(x, IDX(0, 1, 2^t - 1))))$
```

that transforms to

```
    for $q = t : -1 : 1$
        $L = 2^q; r = 2^{t-q}; L_* = L/2; r_* = 2r$
        forall $j = 0 : L_* - 1$
            $x(jr_* : (j + 1)r_* - 1) = M_{r_*} x(jr_* : (j + 1)r_* - 1)$
        end
    end
```

The above code requires 2^t flops (linear time) because the implementation of the matrix vector product $M_n x$ takes linear time as we can see from the definition of M.

The Hartley transform factorization (4.17) is written as

```
$ASSIGN(VEC(x, IDX(0, 1, n^t - 1)),$
            $TIMES(PROD(q, IDX(t, -1, 1),$
                        $KRON(I(n^{t-q}),$
                            $TIMES(KRON(H(n), I(n^{q-1})), D(n, n^{q-1})))),$
                    $VEC(x, IDX(0, 1, n^t - 1))))$
```

The resulting code is

```
    for $q = 1 : t$
        $L = n^q; r = n^{t-q}; L_* = L/n; r_* = nr$
        forall $k = 0 : r - 1$
            $x(kL : (k + 1)L - 1) = D_{n,L_*} x(kL : (k + 1)L - 1)$
            forall $j = 0 : L_* - 1$
                $x(kL + j : L_* : kL + j + L_*(n - 1)) = H_n \cdots$
                $x(kL + j : L_* : kL + j + L_*(n - 1))$
        end
    end
end
```

Other variants can be generated by following the other known factorizations like Stockham, Pease etc.

The digit reversal permutation (4.1) is written as the assignment

$$
\begin{aligned}
&ASSIGN(VEC(x, IDX(0,1,n^t-1)),\\
&\qquad TIMES(PROD(q, IDX(t,-1,1), KRON(I(n^{t-q}), P(n^q, n^{q-1}))),\\
&\qquad\qquad VEC(x, IDX(0,1,n^t-1))))
\end{aligned}
$$

that reduces to

```
for $q = 1 : t$
    $L = n^q; r = n^{t-q}; L_* = L/n; r_* = nr$
    forall $k = 0 : -1 + r$
        $y(0 : -1 + L) = x(kL : -1 + (1 + k)L)$
        forall $j = 0 : -1 + L_*$
            $x(kL + jn : -1 + kL + (1 + j)n) = y(j : L_* : j + L_*(-1 + n))$
        end
    end
end
```

This algorithm requires O(tn^t) operations, thus it is not optimal. A linear time algorithm to calculate the indices of the digit reversal follows, it is the generalization of the algorithm in [Els89].

```
function [idx] = idx_rev(n,p)
% find the index vector for the digit reversal permutation
% n: radix
% t: exponent

idx = zeros(n^t,1);
idx(1:n) = 0:n-1;
for q=2:t
  idx(1:n^(q-1)) = n*idx(1:n^(q-1));
  for j=1:n-1
    idx(n^(q-1)*j+1:n^(q-1)*(j+1)) = idx(1:n^(q-1))+j;
  end
end
```

```
return
```

This algorithm requires exactly $n(n^t - n)/(n-1)$ operations.

4.3 Automatic Derivation of a General Radix

The n-point FFT or any other radix computation implemented as a sequence of assignments, can be derived by unrolling the loops derived in the previous section. The unrolling is done by tracing the execution of the loop nest in a symbolic environment and collecting all the operations as assignments. We provide a symbolic system that performs the radix generation in Maple. For more information regarding the source code and usage, please see Appendix B.

We consider a global lookup table $\mathcal{T}$ where we can store the symbolic expressions that temporary variables are assigned to, using as lookup index the name of the variable. We should think of $\mathcal{T}$ as a relation (or partial function) that associates symbolic names to symbolic expressions. For example if at some point of an algorithm, the temporary variable t_0 is assigned the value $x_0 + x_1$ then we consider that under the relation $\mathcal{T}$, $t_0 \xrightarrow{\mathcal{T}} x_0 + x_1$ and equivalently $\mathcal{T}(t_0)$ represents the symbolic expression $x_0 + x_1$. As a shorthand when y is a vector of symbolic names, $\mathcal{T}(y)$ represents a vector of symbolic expressions.

The function that generates the code, let us call it `gen_fft()`, is identical in structure with a recursive function that calculates the FFT with the following differences:

- The input is a vector of symbolic variables and not numbers.
- There is a side effect, whenever an arithmetic operation is required involving symbolic variables, a new temporary variable is created, the arithmetic op-

eration in symbolic form is entered in the lookup table and the new symbol substitutes the "result of the expression".

- The output is a vector of symbolic variables.

The code implementing the n-point FFT calculation is produced by ordering the listing of the assignment statements involving the temporary variables. This is done by taking the closure of the dependency graph of the temporary variables, starting from the output variables.

For FFT's, there are certain optimizations that can be done "on the fly". It can be shown that an FFT factorization consists of matrices that are either

- diagonal matrices of weights,
- Kronecker products of identity and DFT matrices of smaller sizes,
- or permutation matrices.

The first fact can be exploited by performing the weight multiplications in place, without introducing a new temporary variable. From the second fact, in the case of vector lengths that are a power of two, we can conclude that the temporary variables are assigned an expression that is the sum of two terms and moreover each temporary variable is used in exactly two expressions. This knowledge can be used to simplify the job of optimal instruction scheduling, by declaring these constraints to the instruction scheduler. For example when generating code for a RISC processor, you can reassign the register and pre-empt the cache when the variable that is held by the register has been used twice.

Using the above technique we are able to generate FFT radices of any size, that is up to a practical size. It is interesting to note that the code for radix of length 256 has a size of about 100KB and is almost impossible to be compiled by a compiler with the optimization options on.

4.3.1 An example

Consider the following simplified version of `gen_fft` in a MATLAB -like language that is able to handle symbolic operations and the colon notation:

```
function y = gen_fft(x)
global T
  n = length(x)
  if n == 2
    y = new_temp([x(0)+x(1); x(0)-x(1)])
  else
    [p,m] = factor(n)
    y = StridePerm(p,x)
    for j=1:p
      y((j-1)*m:j*m-1) = gen_fft(y((j-1)*m:j*m-1))
    endfor
    T(y) = w(p,m) .* T(y)
    for j=1:m
      y((j-1):m:n-1) = gen_fft(y((j-1):m:n-1))
    endfor
  endif
return
```

The reader should agree that this function is almost identical to a function implementing the radix FFT. `T` is the global lookup table (that we have described above using the notation $\mathcal{T}$, `new_temp` is a function that on an input of a vector of symbolic expressions, it generates a vector of the same size of brand-new symbolic names, associates those symbolic names with the corresponding expressions in the lookup table `T` and returns the symbolic vector. In our notation the code for `new_temp` is:

```
function y = new_temp(x)
global T
  n = length(x)
  for j=0:n-1
    t = new_symbol()
    T(t) = x(j)
    y(j) = t
  endfor
return
```

with new_symbol like:

```
function s = new_symbol()
global next_free
  s = ['t', int2str(next_free)]
  next_free = next_free + 1
return
```

factor and StridePerm are straightforward functions and we will not deal with them here. Let us trace the events when we call gen_fft with a symbolic vector x with "values" $[x_0, x_1, x_2, x_3]$. From now on we will use the typewriter font, like x, to express program variables and the italic font like x to express symbols.

We want to build a routine that calculates $F_4 x$ for a complex vector x, so we call gen_fft with argument a vector of symbolic variables x and length 4. Control proceeds in the else part of the conditional statement. We have the values $p = 2$ and $m = 2$ and the rest of the program actually implements the following expression:

$$y = (F_2 \otimes I_2) D_{2,2} (I_2 \otimes F_2) P_{4,2} x. \tag{4.19}$$

First the permutation is executed as if the data were numerical and since it does not require any numerical operations involving the symbols in x, there is no side effect. The result is

$$\mathrm{y} = \mathrm{P}_{4,2}\mathrm{x} = \begin{bmatrix} x_0 \\ x_2 \\ x_1 \\ x_3 \end{bmatrix}. \tag{4.20}$$

The next loop performs the Kronecker product operation

$$\mathrm{y} = (\mathrm{I}_2 \otimes \mathrm{F}_2)\mathrm{y} = \begin{bmatrix} F_2 \begin{bmatrix} x_0 \\ x_2 \end{bmatrix} \\ \\ F_2 \begin{bmatrix} x_1 \\ x_3 \end{bmatrix} \end{bmatrix}. \tag{4.21}$$

We call the building routine `gen_fft` two times. For the first call the input is $[x_0,\ x_2]$. This time $n == 2$, so the 'then' part of the conditional is executed. The variable `y` is assigned to $[t_0,\ t_1]$ with $\mathtt{T}(t_0) = x_0 + x_2$ and $\mathtt{T}(t_1) = x_0 - x_2$. Respectively in the second call to `gen_fft`, variable `y` is assigned to $[t_2,\ t_3]$ with $\mathtt{T}(t_2) = x_1 + x_3$ and $\mathtt{T}(t_3) = x_1 - x_3$. After consolidating the results returned from these two calls we have $\mathtt{y} = [t_0,\ t_1,\ t_2,\ t_3]$.

Next comes the calculation $\mathtt{y} = D_{2,2}t$. This is done by directly modifying the entries of the lookup table `T` that becomes

$$\mathtt{T(t)} = \begin{bmatrix} x_0 + x_2 \\ x_0 - x_2 \\ x_1 + x_3 \\ -i(x_1 - x_3) \end{bmatrix} \tag{4.22}$$

because the diagonal entries of $D_{2,2}$ are all equal to 1 except the last entry that is equal to $-i$. We have to mention here that certain optimizations can be done at this point depending on the type of the multiplier, μ. Let $\Re(\mu)$ be the real part of complex number μ and $\Im(\mu)$ its imaginary part.

- $|\Re(\mu)| = 1$ or $|\Im(\mu)| = 1$ then no multiplication flops are required,
- $|\Re(\mu)| = |\Im(\mu)|$ then only 4 multiplication flops are required,
- otherwise 6 multiplication flops are required.

The second loop performs the Kronecker product operation $\mathtt{y} = (F_2 \otimes I_2)t$ that is equivalent to the following, after using the Kronecker product identity (1.6) and the fact that the DFT matrix is symmetric

$$\mathrm{y} = (\mathrm{F}_2 \otimes \mathrm{I}_2)\mathrm{t} = \mathrm{vec}(\begin{bmatrix} t_0 & t_2 \\ t_1 & t_3 \end{bmatrix} \mathrm{F}_2). \tag{4.23}$$

Again, this calculation involves two calls to the builder routine `gen_fft` to determine the 2-point FFT's. For the first call the input is $[t_0,\ t_2]$. Again $n == 2$, so the 'then' part of the conditional is executed. The variable `y` is assigned to $[t_4,\ t_5]$ with $\mathtt{T}(t_4) = t_0 + t_2$ and $\mathtt{T}(t_5) = t_0 - t_2$. Respectively in the second call to `gen_fft`, variable `y` is assigned to $[t_6,\ t_7]$ with $\mathtt{T}(t_6) = t_1 + t_3$ and $\mathtt{T}(t_7) = t_1 - t_3$. After consolidating the results returned from these two calls we have $\mathtt{y} = [t_4,\ t_6,\ t_5,\ t_7]$ (because of the *vec*() operator).

Let us recoup by reviewing what happened. On input $[x_0,\ x_1,\ x_2,\ x_3]$, `gen_fft` returns with $[t_4,\ t_6,\ t_5,\ t_7]$ and the global lookup table `T` has the following entries:

$$\mathtt{T} = \begin{bmatrix} t_0 & \rightarrow & x_0 + x_2 \\ t_1 & \rightarrow & x_0 - x_2 \\ t_2 & \rightarrow & x_1 + x_3 \\ t_3 & \rightarrow & -i(x_1 - x_3) \\ t_4 & \rightarrow & t_0 + t_2 \\ t_5 & \rightarrow & t_0 - t_2 \\ t_6 & \rightarrow & t_1 + t_3 \\ t_7 & \rightarrow & t_1 - t_3 \end{bmatrix} \tag{4.24}$$

From the above we can generate the 4-point FFT by matching the input and the output vectors to some abstract input output vectors `x` and `y` and listing the necessary assignments to get:

```
function y = fft4(x)
  t0 = x(0)+x(2)
  t1 = x(0)-x(2)
  t2 = x(1)+x(3)
  t3 = -i*(x(1)-x(3))
  y(0) = t0+t2
  y(1) = t1+t3
  y(2) = t0-t2
  y(3) = t1-t3
return
```

The same flop count is achieved using the split radix butterfly. There is no difference at all in handling any other factorizations like the split radix or factorizations for the the prime factor and the prime FFT. For the inverse FFT we only need to use the conjugates of the weights and divide the output by n.

On the final code generator we ended up using the split radix approach because it requires the lowest number of floating point operations. The next two subsections explain the choice.

4.3.2 Counting Floating Point Operations for the FFT

Suppose we have an efficient "base-case" FFT algorithm that operates on vectors of length p in f flops. We can use this algorithm to handle FFT's of length $n = p^t$, by using the general radix factorization:

$$F_n = (F_p \otimes I_{n/p}) D_{p,n/p} (I_p \otimes F_{n/p}) P_{n,p} \tag{4.25}$$

where I_n is the identity matrix of size n, D is the diagonal matrix of weights as defined in (4.4) and $P_{n,p}$ is the stride permutation matrix.

Therefore, after ignoring the cost of pre-calculating the weights, the number of flops $T(n)$ needed to calculate the FFT of length n obeys the following recursive equation:

$$T(n) = \frac{n}{p} T(p) + 6(p-1)\frac{n}{p} + pT(\frac{n}{p}) \tag{4.26}$$

with $T(p) = f$. The first term of the summation expresses the cost of $\frac{n}{p} = p^{t-1}$ FFT's of length p, the second term expresses the cost of performing $(p-1)p^{t-1}$ complex multiplications for the diagonal matrix times vector product and the last term represents the cost of p FFT's of length $\frac{n}{p}$. The solution of the recursion is

$$T(n) = \frac{1}{p}(f + 6(p-1)) n \log_p(n) + fn. \tag{4.27}$$

In the FFT literature, the number of floating point operations is usually given in the form of the highest complexity term together with its coefficient. For example, the use of the radix-2 algorithms requires $5n\log_2(n)$ flops for performing the Fourier transform of a vector of length n. The corresponding coefficient of the analytical form above of the term $n\log_2(n)$ is $\frac{f+6(p-1)}{p\log_2 p}$. For the radix-2 case, $f = 4$ because we need only 4 flops to calculate the FFT of length 2. Substituting p with 2 and f with 4 in (4.27) we get $5n\log_2(n)$.

The formula in (4.27) shows the relationship between the flops required for a single FFT to the coefficient of the highest complexity term $n\log_2(n)$. It is important to have the lowest possible number of flops required for the radix computation.

When we double the size of the radix, the number of flops can be reduced because we can avoid some multiplications. For the radix-4, 16 flops are required to calculate the FFT of length 4. Substituting p with 4 and f with 16 in (4.27) we get

$$\frac{17}{2}n\log_4(n) = 4.25n\log_2(n).$$

Tables 4.1, 4.2 and 4.3 show the number of flops required for the calculation of an FFT using various radices. With this and the following two tables we try to answer the question of which radix is the best choice for the calculation of an FFT of a given length. Each entry of the tables contains 2 numbers:

- The number of flops required for the calculation of an FFT of length determined by the row index, using as primary radix the size present as the column index. We say "primary" radix here, because the different tables correspond to different strategies for choosing a radix.

- The coefficient of $n\log_2 n$ if this length was used for the radix calculation.

Table 4.1: Flops required for the calculation of the FFT of specific length sizes 4 to 256 and the corresponding coefficient of $n \log_2 n$ for different radices from 2 to 16.

Radix FFT

	2		4		8		16	
4	16	4.250						
8	56	4.083	56	4.083				
16	176	4.156	172	4.094	186	4.313		
32	496	4.263	468	4.088	478	4.150	526	4.450
64	1296	4.359	1204	4.118	1190	4.083	1270	4.292
128	3216	4.440	2932	4.123	3014	4.214	3014	4.214
256	7696	4.509	6964	4.147	6918	4.125	7302	4.313

The strategy for choosing the radix for Table 4.1 is to use the primary radix (the one mentioned at the respective column) as often as possible, and then use the one immediately to the left. For example, the (128,8) entry of Table 4.1 says that 3014 flops are required to calculate an FFT of length 128, using radix 8 whenever possible. If this specific approach for calculating the length 128 FFT is used as a radix to solve problems with lengths 128^q then the coefficient of $n \log_2 n$ is 4.214.

The strategy for choosing the radix for Table 4.2 is to use the primary radix to split the problem unless there is a better algorithm to use from Table 4.1. As it is shown on Table 4.3, the hunt for the best radix calculation is decided as the performance of the split radix algorithm is far superior than the other "sub-radix" algorithms. It is also interesting to note that the coefficient of $n \log_2 n$ is dropping, as we increase the size of the butterfly. These are the smallest flop count coefficients known so far for the calculation of the FFT of a vector of length $n = p^t$.

Table 4.2: Flops required for the calculation of the FFT of specific length sizes 4 to 256 and the corresponding coefficient of $n \log_2 n$ for different radices from 2 to 32 using the best radix for each subproblem.

Best-radix FFT

	2		4		8		16		32	
4	16	4.250								
8	56	4.083	56	4.083						
16	176	4.156	172	4.094	186	4.313				
32	488	4.213	468	4.088	478	4.150	498	4.275		
64	1240	4.214	1204	4.118	1190	4.083	1214	4.145	1250	4.238
128	3004	4.201	2932	4.123	2902	4.089	2902	4.089	2942	4.134
256	7068	4.198	6908	4.120	6838	4.086	6838	4.086	6838	4.086

Table 4.3: Flops required for the calculation of the FFT of specific length sizes 4 to 256 and the corresponding coefficient of $n \log_2 n$ using the split radix algorithm.

Split-radix FFT

8	56	4.083
16	168	4.031
32	456	4.013
64	1160	4.005
128	2824	4.002
256	6664	4.001

4.3.3 The Split-Radix Algorithm

Unlike the general p-radix algorithms that are based on the synthesis of p FFTs of subvectors of lengths n/p, the split-radix algorithm is based on the synthesis of one half-length and two quarter-length FFTs. In the following we express the split-radix algorithm in matrix form and generalize it to arbitrary radix, that is we show that we are not restricted to only combining one half-length and two quarter-length FFTs but also a one-third and six one-ninth FFTs and so on.

More formally, let $n = km = k^2p$, then the DFT matrix can be written as

$$F_n = (F_k \otimes I_m)D_{k,m}(I_k \otimes F_m)P_{n,k} \tag{4.28}$$

that defines the family of radix-k algorithms with complexity $5n\log_2(n)$ if $k = 2$ and $4.25n\log_2(n)$ if $k = 4$, as we have already seen. Matrix $D_{k,m}$ is a block diagonal matrix with blocks of size m that are

$$[D_{k,m}]_{jj} = \Omega^j_{k,m} \text{ where } \Omega_{k,m} = \text{diag}(1, \omega_{km}, \cdots, \omega^{m-1}_{km}) \text{ and } j = 0, \cdots, k-1$$

a diagonal matrix with elements powers of the $n = km$ root of unity.

It is easy to see that the following lemma holds:

Lemma 4.3.1 *For all positive integers $n = km = k^2p$ and $j = 0, \cdots, k-1$, it holds*

$$\Omega^j_{k,m} = \Omega^j_{k,k} \otimes \Omega^j_{k^2,p}$$

Proof: The equality follows from the fact that $\omega_{k^2} = \omega^p_{k^2p} = \omega^p_n$.

□

Repeating (4.28) for $m = kp$ we get

$$F_m = (F_k \otimes I_p)D_{k,p}(I_k \otimes F_p)P_{m,k}$$

From (4.28), the definition of diagonal matrix D and Lemma 4.3.1, we see that we can "propagate" the diagonal scaling $\Omega^j_{k,m}$ for some index j one level forward

$$\begin{aligned}
\Omega^j_{k,m} F_m &= \Omega^j_{k,m}(F_k \otimes I_p) D_{k,p} (I_k \otimes F_p) P_{m,k} \\
&= (\Omega^j_{k,k} \otimes \Omega^j_{k^2,p})(F_k \otimes I_p) D_{k,p} (I_k \otimes F_p) P_{m,k} \\
&= (\Omega^j_{k,k} F_k \otimes I_m)(I_k \otimes \Omega^j_{k^2,p}) D_{k,p} (I_k \otimes F_p) P_{m,k}.
\end{aligned}$$

The product $(I_k \otimes \Omega^j_{k^2,p}) D_{k,p}$ is a diagonal matrix that can be precalculated. The cost of multiplying a vector x of length m by this diagonal matrix is $6m$ flops that is $6k$ flops more than the flops required to calculate $D_{k,p}x$ that is $6(k-1)p$. However, some flops can be saved by the calculation of $\Omega^j_{k,k}(F_k x)$. Therefore, we can state that

Theorem 4.3.2 *If the calculation of $\Omega^j_{k,k}x$ for some $j \in [1, \cdots, k-1]$ requires less than $6(k-1)$ flops for the scaling of $k-1$ complex numbers in vector x, the k-split-radix algorithm is faster than the corresponding k-radix algorithm.*

Proof: Let $T(k)$ be the number of flops required for the radix-k calculation. The traditional algorithm derived from (4.28), the complexity is given by

$$T(n) = (T(k) + 6(k-1))\frac{n}{k} + kT(\frac{n}{k})$$

with a solution

$$T(n) = \frac{T(k) + 6(k-1)}{k} n \log_k(n) + O(n)$$

as we have already seen in the previous section. The complexity for the split-radix algorithm is given by

$$T(n) = (T(k) + 6(k-2))\frac{n}{k} + (k-1)T(\frac{n}{k}) + (T(k) + \alpha)\frac{n}{k^2} + 6\frac{n}{k} + kT(\frac{n}{k^2})$$

and α the additional to $T(k)$ flops required for the calculation of $\Omega^j_{k,k} F_k x$. The solution to this recurrence is

$$T(n) = \frac{(k+1)T(k) + 6k(k-1) + \alpha}{k(k+1)} n \log_k(n) + O(n).$$

By requiring

$$\frac{(k+1)T(k) + 6k(k-1) + \alpha}{k(k+1)} < \frac{T(k) + 6(k-1)}{k}$$

we conclude that $alpha < 6(k-1)$, that is, the number of flops required to scale by the twiddle factors in the expression $\Omega^j_{k,k} x$. $\square$

The implication of this theorem is that we now have a quantitative way to express the difference between radix and split-radix calculations. We also show that the split-radix idea can be applied to any radix and not just 2 as well as to any dimensional problem since the matrix factorization for the split-radix algorithm extends directly to multiple dimensions with the help of the multiplicative properties of the Kronecker product. We can now search to find the best split radix algorithm by just looking at the types of the diagonal elements of matrix $\Omega^j_{k,k}$ for $j = 1, \cdots, k-1$.

Indeed, for $k = 2$ we have $\Omega_{2,2} = \text{diag}([1, -i])$ which means that the scaling $\Omega_{2,2} x$ can be done with no flops i.e. $\alpha = 0$. With $T(2) = 4$ we get the best complexity so far that is $4n \log_2(n)$. For $k = 4$ we have

$$\Omega^2_{4,4} = \text{diag}([1, \frac{\sqrt{2}}{2}(1-i), -i, -\frac{\sqrt{2}}{2}(1+i)])$$

that means that $\alpha = 8$. The resulting complexity leading-term remains 4.

We used the Split-Radix factorization to generate the radix computation codes for FFTs.

4.4 Benchmarks

Modern workstations achieve high performance because of the use of very fast RISC processors. A RISC (Reduced Instruction Set Computer) is designed to operate with very simple instructions that can start each data path cycle time (the time required to fetch the operands from their registers, operate on them in the Arithmetic Logic Unit (ALU) and store the results back into a register). All instructions operate on data that are present in registers and not in main memory, thus a RISC processor contains a large register set. There are only simple load and store instructions that move data from memory to registers and back. The RISC processors contain multiple execution units that are also pipelined, that is the units are able to start executing a new instruction every cycle even though the previous instructions did not complete their executions. The simplicity of the instruction set, the lack of microcode and the use of multiple pipelined execution units make a RISC processor very fast and keep the hardware as simple as possible.

However in order to keep a RISC processor so simple, all the complexity of programming has been passed to the compiler. To make this hardware perform useful work and fast, the compiler has to schedule instructions to multiple units, keep the pipelines full and properly allocate the registers. All these problems are NP-hard so heuristics are used to find respectable solutions. This is why a good optimizing compiler is so critical, and why code that cleverly matches the underlying architecture of the processor performs well.

Another issue that greatly affects performance is memory latency. Memories have always been slower than the processors they serve. Actually the problem is not due to technology but to economics. Fast memory costs a lot. The solution is to combine a small amount of fast memory, the *cache*, that matches the speed of the processor with large amounts of slow memory. Because of the locality principle which says that any meaningful program will only access a small part of the total available memory during any small time interval, cache memory can improve performance by storing frequenty accessed memory parts. Accesses to main memory cost many processor cycles so pipelines are interrupted. When a data item is found in the cache when requested, we call it a *cache hit*, otherwise it is a *cache miss* and a normal, time consuming memory request is issued. There are three types of cache memory, the *directly mapped* which is the cheapest but the most inefficient because many memory locations are mapped to the same cache location, *fully associative* which allows complete freedom of mapping but is the most expensive and the *set associative* which is a compromise of both.

Although a programmer has no direct way to define what is stored in the cache at each moment, there are programming techniques that indirectly provide partial control. The most common technique is *prefetching*, that is the data access pattern of an algorithm is modified so that an item that is likely to cause a cache miss is brought into cache memory before it is actually needed. This technique overlaps the memory latency with useful work since a lengthy interruption is avoided. Another programming technique is *cache line blocking*, that is the data structures used by an algorithm are aligned in the main memory in such a way that they do not map on the same cache line. This technique can avoid the catastrophic overhead of successive cache misses in the event of two array sections that are needed simultaneously, are competing for the same cache block.

The most well known public domain software for calculating FFTs is the FFTPACK from NETLIB. FFTPACK is a package of Fortran subprograms for the fast

Fourier transform of periodic and other symmetric sequences. It includes complex, real, sine, cosine, and quarter-wave transforms [Swa82].

MATLAB computes the FFT by calling optimized FFT routines of radix 2.

We have run FFT of various sizes on different workstations to compare the performance of our code with public domain code, as well as proprietary code like the IBM ESSL FFT. For our tests we only considered double precision complex FFTs in FORTRAN 77. The FORTRAN compilers achieved much better optimizations than the FORTRAN 90 and C compilers.

In general, as the results show in tables 4.4 and 4.5 and figures 4.3 and 4.4, our generated code is faster or at least as fast as the public domain code from FFTPACK, and much faster than the FFT from MATLAB . However, the automatically generated code performs worse when compared with the finely tuned FFT from the IBM ESSL. As it can be seen on table 4.4 the performance of the generated FFT code up to size 1024 are comparable or better than the IBM ESSL, for larger sizes though, when the cache utilization becomes critical (the size of the problem is too large for the data to fit in the cache), the ESSL FFT outperforms the competition[1].

Both programming techniques for improving cache use, prefetching and cache line blocking are used by IBM to produce highly tuned ESSL routines for each computing platform. Since the determination of the correct use of the above programming techniques is beyond the capabilities of current compiler technology, IBM chose to invest an additional effort to hand tune the code of ESSL routines that are commonly used, like the FFT. That is how the IBM ESSL routines achieve very close to theoretical peak performance [AG92].

[1]The use of pointers from FORTRAN 90 allowed a dramatic reduction of about 50% to our times on the IBM RS6000 that we report here.

Table 4.4: IBM RS6000 benchmarks and comparison with ESSL FFT.

	Matlab	ESSL
4	2.37e-04	1.20e-05
8	2.34e-04	
16	2.51e-04	1.60e-05
32	2.91e-04	
64	3.32e-04	5.10e-05
128	4.18e-04	
256	5.92e-04	1.58e-04
512	9.72e-04	
1024	1.76e-03	7.08e-04
2048	4.05e-03	
4096	7.85e-03	5.43e-03
8192	8.14e-02	
16384	1.78e-01	2.52e-02
32768	3.79e-01	
65536	8.54e-01	1.15e-01
131072	1.80e+00	
262144	4.00e+00	4.92e-01
524288	1.23e+01	
1048576	2.86e+01	2.24e+00

	4	8	16	32	64	128	256
2^2	1.00e-05						
2^3		1.20e-05					
2^4	2.70e-05		1.50e-05				
2^5				2.30e-05			
2^6	9.10e-05	6.20e-05			4.10e-05		
2^7						8.30e-05	
2^8	3.72e-04		1.93e-04				3.81e-04
2^9		5.25e-04					
2^{10}	1.68e-03			7.94e-04			
2^{12}	1.66e-02	1.19e-02	9.58e-03		7.97e-03		
2^{14}	8.01e-02					4.88e-02	
2^{15}		2.03e-01		1.34e-01			
2^{16}	3.40e-01		3.17e-01				5.46e-01
2^{18}	1.79e+00	1.89e+00			1.29e+00		
2^{20}	7.97e+00		8.21e+00				

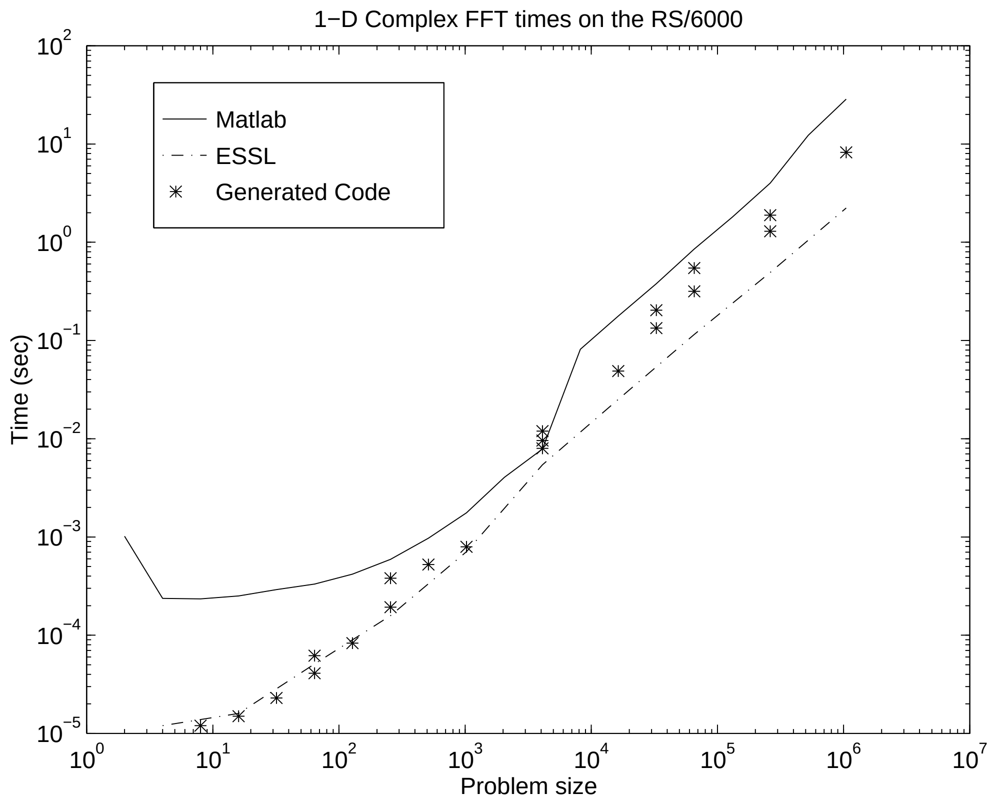

Figure 4.3: IBM RS6000 benchmarks and comparison with ESSL FFT.

Table 4.5: SUN SPARC benchmarks and comparison with FFTPACK.

	Matlab	FFTPACK
4	7.06e-04	1.73e-04
8	7.11e-04	2.03e-04
16	7.87e-04	2.51e-04
32	1.37e-03	3.67e-04
64	1.05e-03	5.77e-04
128	1.43e-03	1.01e-03
256	2.47e-03	1.91e-03
512	5.55e-03	4.26e-03
1024	1.18e-02	8.47e-03
2048	2.42e-02	1.73e-02
4096	5.09e-02	3.55e-02
8192	1.08e-01	8.24e-02
16384	2.51e-01	1.82e-01
32768	5.80e-01	4.22e-01
65536	1.35e+00	9.28e-01
131072	5.45e+00	2.24e+00
262144	1.46e+01	5.07e+00
524288	3.47e+01	1.09e+01
1048576	7.54e+01	2.04e+01

	4	8	16	32	64	128	256
2^2	1.72e-04						
2^3		1.57e-04					
2^4	2.37e-04		2.19e-04				
2^5				2.56e-04			
2^6	4.91e-04	3.42e-04			4.53e-04		
2^7						1.05e-03	
2^8	1.70e-03		1.24e-03				3.28e-03
2^9		2.20e-03					
2^{10}	7.06e-03			4.83e-03			
2^{12}	3.16e-02	2.04e-02	2.38e-02		2.48e-02		
2^{14}	1.50e-01					1.85e-01	
2^{15}		2.58e-01		1.72e-01			
2^{16}	1.14e+00		7.87e-01				1.52e+00
2^{18}	6.61e+00	4.10e+00			4.13e+00		
2^{20}	2.61e+01		1.62e+01				

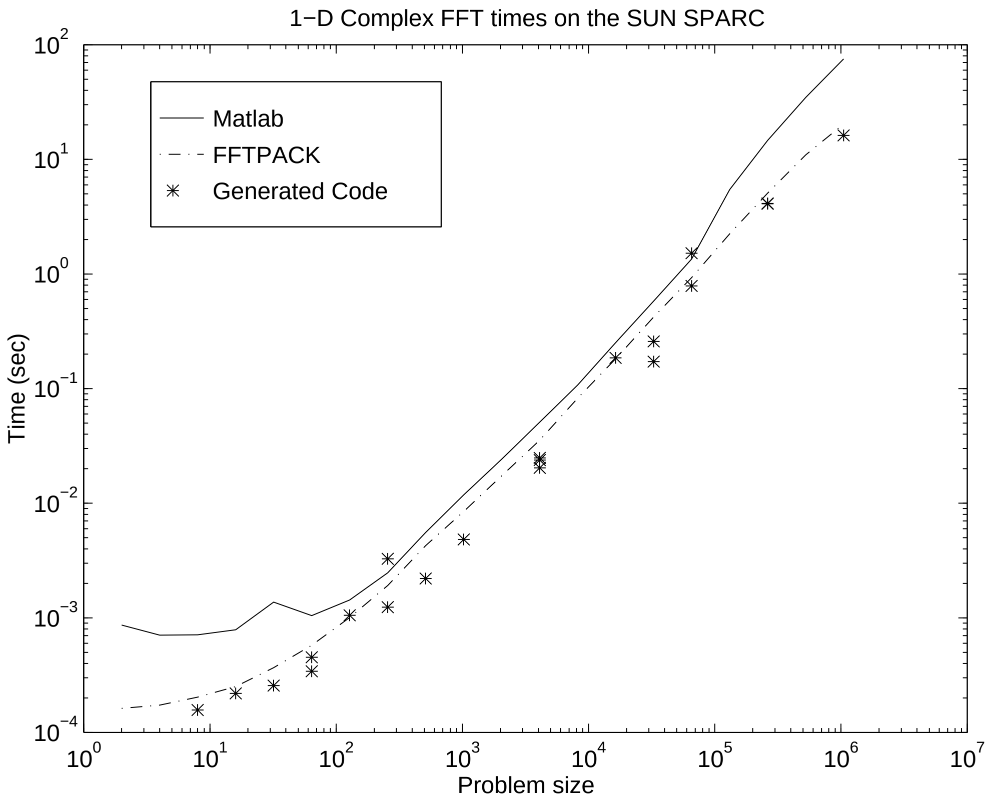

Figure 4.4: SUN SPARC benchmarks and comparison with FFTPACK.

Automatic code generation allows the production of larger radix codes. We have produced FFT radices up to size 256 to see the effect of code size to performance. The larger radix code makes better use of the memory hierarchy, compared to radix 2 algorithms, because the loop nest is stronger, that is, more operations are performed on data that most likely are either in registers on in the cache. Also, the use of a larger radix algorithm implies fewer passes over the data. For example, a radix-2 FFT will perform twice as many passes over the data of an FFT of length 2^{2t} compared to a radix-4 FFT. An advanced compiler can achieve loop nest strengthening by loop unrolling, a standard loop optimization technique, however the resulting code is not as efficient because a compiler cannot discover all the simplifications due to trigonometric identities that are implicitly present in the matrix factorization.

In our code generation system we have no target hardware specification and thus we proceed in the transformation steps without specific optimizations. It is an open question the proper use of code transformation to achieve prefetching and cache line blocking.

4.5 Extending the Approach

The FFT and the other transforms should not be seen as black box algorithms. There are many optimizations that can be done by taking advantage of their rich structure. A compiler is not capable of discovering all possible optimizations. When an algorithm is expressed as a sequence of conditionals, scalar assignments and branching statements, a much higher level of complexity is introduced, that no compiler can overcome.

Fast transform algorithms are not destinations by themselves, but rather important parts of other algorithms in scientific computing. There is a great need for an environment that is capable of choosing and combinining the various FFT implementations to weave more complicated applications. We review some cases below.

4.5.1 High Dimensional Problems

The two dimensional Fourier transform of a matrix X of size m-by-n is given by

$$\mathrm{vec}(F_m X F_n^T) = (F_n \otimes F_m)\mathrm{vec}(X).$$

We can use the Kronecker identity

$$(F_n \otimes F_m) = (I_n \otimes F_m)(F_n \otimes I_m)$$

and thus simplify the 2-dimensional Fourier transform to a sequence of multiple 1-dimensional FFT's, that we have already shown how to handle.

However we can join the two loops into one by the following approach. The result will be to perform the 2-dimensional transform as a sequence of 2-dimensional smaller transforms, instead of multiple 1-dimensional ones. For simplicity, let us assume that we need the 2-D FFT of matrix X of size m^t-by-n^t. Let the DFT matrix factorization be

$$F_{n^t} = \prod_{q=t,t-1}^{1} A_q \text{ and } F_{m^t} = \prod_{q=t,t-1}^{1} B_q$$

then the 2-D FFT can be written as

$$\begin{aligned}(F_{n^t} \otimes F_{m^t})x &= \left(\left(\prod_{q=t,t-1}^{1} A_q\right) \otimes \left(\prod_{q=t,t-1}^{1} B_q\right)\right) x \\ &= \left(\prod_{q=t,t-1}^{1} (A_q \otimes B_q)\right) x\end{aligned}$$

that corresponds to the code:

```
for $q = 1 : t$
    $X = B_q X A_q^T$
end
```

Therefore we can now try combinations of different algorithms for the "left" and "right" FFT's. We already know that the different FFT algorithms have radically different memory access patterns. In addition, the "left" FFT's handle vectors that are continuous in memory, –assuming FORTRAN storage– while the "right" FFT's handle vectors with successive elements of distance m (stride m). It makes sense then to use different algorithms for the left and right Fourier transform.

The same applies to higher dimensional problems. A d-dimensional problem can be written as a one dimensional problem

$$(\bigotimes_{k=d,d-1}^{1} F_{n_k})x$$

where n_k is the length along the k^{th} dimension.

The transformation rules we need to handle multidimensional problems are the following:

R11 Kronecker of a product: This rule handles expressions

$$y = ((M_1 M_2) \otimes (M_3 M_4))\, x$$

with

$$y = ((M_1 \otimes M_3)\, (M_2 \otimes M_4))\, x$$

The Mathematica code is

$$\begin{array}{c} KRON[TIMES[\mathbf{M1},\mathbf{M2}],TIMES[\mathbf{M3},\mathbf{M4}]] \quad :> \\ TIMES[KRON[\mathbf{M1},\mathbf{M3}],KRON[\mathbf{M2},\mathbf{M4}]] \end{array}$$

R12 Kronecker of Π Products: This is the extension of rule **R11** to substitute expressions

$$y = \left(\left(\prod_{q=1}^{t} A_q \right) \otimes \left(\prod_{q=1}^{t} B_q \right) \right) x$$

with

$$y = \left(\prod_{q=1}^{t} (A_q \otimes B_q) \right) x$$

The code implementing the rewrite is

$$\begin{aligned} &KRON[PROD[\mathbf{q}, \mathbf{idx}, \mathbf{M1}], \\ &\qquad PROD[\mathbf{q}, \mathbf{idx}, \mathbf{M2}]] \quad :> \quad PROD[\mathbf{q}, \mathbf{idx}, KRON[\mathbf{M1}, \mathbf{M2}]] \end{aligned}$$

The assumption is that both Π products have the same number of terms, otherwise we add identity matrices to the product with the fewer factors.

R13 $A \otimes I \otimes B$. This rule permits the exposure of an identity matrix that is hidden in a series of Kronecker products. The assignment

$$y = (A_{n_1} \otimes I_n \otimes B_{n_2})x$$

can be rewritten using the commutative property of Kronecker products

$$y = (P_{n_1 n n_2, n_1}(I_n \otimes B_{n_2} \otimes A_{n_1}) P_{n_1 n n_2, n n_2})x.$$

The code implementing the rewrite is

$$\begin{aligned} ASSIGN[\mathbf{y}, TIMES[\quad & KRON[\mathbf{M1}, I[\mathbf{ni}], \mathbf{M2}], \mathbf{x}]] :> \\ & (n1 = SIZE[\mathbf{M1}]; n2 = SIZE[\mathbf{M2}]; \\ ASSIGN[\mathbf{y}, TIMES[\quad & P[n1 * n2 * \mathbf{ni}, n1], \\ & KRON[I[\mathbf{ni}], \mathbf{M2}, \mathbf{M1}], \\ & P[n1 * n2 * \mathbf{ni}, \mathbf{ni} * n2], \mathbf{x}]]) \end{aligned}$$

An interesting situation arises when we extend our transformation rules to handle multidimensional algorithms. In the grammar of the matrix language we defined TIMES and KRON as binary operators (that is with exactly two operants). However, both operators correspond to associative operations, both matrix multiplication and Kronecker product are associative, it does not matter how we parenthesize a sequence of matrix multiplications or Kronecker products. So we used to associate both operators to the right, that is the expression $A \otimes B \otimes C$ was always reduced to `KRON[A,KRON[B,C]]` but this created problems in trying to describe patterns of the form $A \otimes B \otimes C \otimes I$, where we are interested in generating a loop out of the last identity matrix. But by associating to the right, you cannot know how deeply nested the identity matrix is. The solution to this problem is to define the operators TIMES and KRON as flat with the statements

```
Attributes[KRON] = {Flat, OneIdentity}
Attributes[TIMES] = {Flat, OneIdentity}
Attributes[SEQ] = {Flat, OneIdentity}
```

that means that TIMES, KRON and SEQ for Mathematica are not binary. Expressions like `KRON[A,KRON[B,C]]` reduce to `KRON[A,B,C]`, while `KRON[A]` reduces to `A`.

The new element in multidimensional transforms is that we now have many possible alternatives to explore. For example

$$\begin{aligned} A_{n_1} \otimes I_n \otimes B_{n_2} &= (P_{n_1 n n_2, n_1}(I_n \otimes B_{n_2} \otimes A_{n_1}) P_{n_1 n n_2, n n_2}) \\ &= (P_{n_1 n n_2, n_2}(B_{n_2} \otimes A_{n_1} \otimes I_n) P_{n_1 n n_2, n_2}). \end{aligned}$$

Therefore we need to define a cost function and a search procedure in the space of semantically equivalent transforms. We elaborate on these ideas in §4.7.

4.5.2 High Level Optimizations

We envision a software engineering environment that knows all about FFT and performs code transformations either along some predefined strategies (as in a theorem prover) or interactive user-assisted searches in the space of program transforms trying to develop an optimal program for a given computational problem and computer architecture. We give two examples below where such a system can be useful.

The *Hadamard product* of two matrices A and B of the same shape, is the matrix resulting from their elementwise multiplication. It is written as

$$C = A \odot B \Leftrightarrow c_{ij} = a_{ij} b_{ij}.$$

The following lemma holds

Lemma 4.5.1 *If the pair of matrices A and C, and the pair B and D are of the same shape respectively, then*

$$(A \otimes B) \odot (C \otimes D) = (A \odot C) \otimes (B \odot D). \tag{4.29}$$

Proof: Let us consider the (i, j) block of the left-hand-side matrix of 4.29, it is

$$(a_{ij} B) \odot (c_{ij} D) = a_{ij} c_{ij} (B \odot D)$$

that is the (i, j) block of the right-hand-side matrix. $\square$

Convolution Via FFT

Let us consider the convolution, or polynomial multiplication algorithm that uses the FFT. Given two polynomials $p_1(x)$ and $p_2(x)$ of degrees d_1 and d_2 respectively, we construct two vectors p_1 and p_2 both of length $n = d_1 + d_2 + 1$, that contain the coefficients of the corresponding polynomials and zeros at the rest of the positions. We know that the coefficients of the polynomial $p(x)$ that is the product of p_1 and p_2 are given by

$$p = F_n^{-1}((F_n p_1) \odot (F_n p_2)).$$

As we have already seen, certain FFT algorithms have a part that performs the bit reversal permutation. These are the Cooley-Tukey, Pease and Gentleman-Sande. The Stockham and transposed Stockham algorithms avoid the bit reversal by performing it in stages, together with the intermediate FFT operations. The above polynomial multiplication algorithm can be performed without 3 bit reversals and here is how: Let $F_n = A_n R_n$ the factorization of the DFT corresponding to the Cooley-Tukey algorithm, where R is the bit reversal permutation matrix. The DFT matrix is symmetric therefore

$$F_n = F_n^T = (A_n R_n)^T = R_n^T A_n^T.$$

Actually the new factorization corresponds to the Gentleman-Sande algorithm. Moreover the inverse is given by

$$F_n^{-1} = \frac{1}{n}\bar{F}_n = \frac{1}{n}\bar{A}_n R_n$$

Then the above algorithm is equivalent to

$$\begin{aligned} p &= \frac{1}{n}\bar{A}_n R_n((R_n^T A_n^T p_1) \odot (R_n^T A_n^T p_2)) \\ &= \frac{1}{n}\bar{A}_n R_n R_n^T((A_n^T p_1) \odot (A_n^T p_2)) \\ &= \frac{1}{n}\bar{A}_n((A_n^T p_1) \odot (A_n^T p_2)) \end{aligned}$$

because for all permutation matrices P and general matrices X and Y of conforming sizes it holds that

$$(PX) \odot (PY) = P(X \odot Y).$$

Given that the bit reversal can take as much as 30% of the time of the FFT, the above transformations achieve considerable savings. There are no known compiler optimization techniques that could have eliminated all that unnecessary data shuffling.

Phase Correlation Via FFT

In 1975 Kuglin and Hines have presented the *phase correlation* method for aligning two images which are shifted relative to each-other [KH75]. For example, suppose you are trying to capture a panorama with a normal camera by allowing the right part of a picture overlap with the left part of the next picture. The phase correlation algorithm finds the displacements necessary to best align those two pictures.

More formally, given two vectors x and y of the same length n that only differ by a displacement δ, that is

$$y(j) = x(j - \delta)$$

we can find the phase difference δ by locating the maximum element of the vector

$$z = \left| F^{-1} \left(Fx \odot \overline{Fy} \right) \right|$$

where $\overline{v}$ is the conjugate of vector v.

The same gains can be achieved by applying exactly the same transformations on an algorithm for matching images, from digital image processing. Given images M_1 and M_2, the position of the element of maximum length of the matrix

$$M = (FM_1F^T) \odot \overline{(FM_2F^T)}$$

determines the displacement that provides the best matching of the two overlapping parts [Bro92]. As above, we perform the FFT's without the bit reversals, we only need to bit reverse the indices of the max position of the new M we have calculated.

4.6 Wavelets

Multi-resolution analysis provides the most natural way to link the wavelet transforms together, moreover the semantics of this analysis give us a factorization of the wavelet matrices.

Suppose we have a discrete signal $s^{(t)}$, a vector of length 2^t. We want to find a process to *decompose* s into a lower resolution "main" signal $s^{(t-1)}$ and a "detail" signal $d^{(t-1)}$ with the following properties:

- signals $d^{(t-1)}$ and $s^{(t-1)}$ occupy the same space as the original signal $s^{(t)}$,
- the original signal $s^{(t)}$ can be *reconstructed* from its constituents $s^{(t-1)}$ and $d^{(t-1)}$,
- the process can be recursively applied to $s^{(t-1)}$.

For example if we apply this process recursively to a signal $s^{(3)}$ of length $2^3 = 8$, we will get the resulting vector:

$$[s_1^{(0)}\ d_1^{(0)}\ d_1^{(1)}\ d_2^{(1)}\ d_1^{(2)}\ d_2^{(2)}\ d_3^{(2)}\ d_4^{(2)}]^T. \tag{4.30}$$

Let L and H two linear operators, L a low pass filter that lets constants and low frequencies to pass through and H a high pass filter such that $L_{2^t}s^{(t)} = s^{(t-1)}$ and $H_{2^t}s^{(t)} = d^{(t-1)}$. We join them together to form matrix

$$M_{2^t} = \begin{bmatrix} L_{2^t} \\ H_{2^t} \end{bmatrix} \text{ so that } M_{2^t}s^{(t)} = \begin{bmatrix} s^{(t-1)} \\ d^{(t-1)} \end{bmatrix}$$

moreover, since we must be able to reconstruct $s^{(t)}$, matrix M_{2^t} must be invertible.

At the next stage, we apply $M_{2^{t-1}}$ only in the $s^{(t-1)}$ signal; in reference to the original signal $s^{(t)}$ this can be written as

$$\begin{bmatrix} M_{2^{t-1}} & O \\ O^T & I_{2^{t-1}} \end{bmatrix} \mathrm{M}_{2^t}\mathrm{s}^{(t)}. \tag{4.31}$$

In general, we can consider a transformation matrix

$$\mathrm{T}_{2^{\mathrm{t}}} = \prod_{\mathrm{q}=1}^{\mathrm{t}} \begin{bmatrix} M_{2^q} & O \\ O^T & I_{2^t-2^q} \end{bmatrix} \tag{4.32}$$

where O is a matrix of zeros of the appropriate size and I_n the n-by-n identity matrix, that apply directly to $s^{(t)}$ to give us $s^{(0)}$ and the multi-resolution details $d^{(0)}, \cdots, d^{(t-1)}$. This is the *wavelet transform matrix.*

It is easy to see that when the matrices M_{2^q} are orthogonal then the wavelet transform matrix is orthogonal, moreover the inverse of the transform matrix is given by

$$\mathrm{T}_{2^{\mathrm{t}}}^{-1} = \mathrm{T}_{2^{\mathrm{t}}}^{\mathrm{T}} = \prod_{\mathrm{q}=\mathrm{t},\mathrm{t}-1}^{1} \begin{bmatrix} M_{2^q}^T & O \\ O^T & I_{2^t-2^q} \end{bmatrix}. \tag{4.33}$$

Given this definition-factorization of the general transform matrix T we can revisit the wavelet transforms of Haar and Daubechies to redefine them by choosing the appropriate down-sampling matrix M.

The Haar wavelet transform matrix is given by choosing the *down-sampling* filters

$$\mathrm{M}_{2^{\mathrm{q}}} = \begin{bmatrix} L_{2^q} \\ H_{2^q} \end{bmatrix} = \begin{bmatrix} I_{2^{q-1}} \otimes [1\ 1] \\ I_{2^{q-1}} \otimes [1\ -1] \end{bmatrix}. \tag{4.34}$$

Matrix L_{2^q} is the averaging function and H_{2^q} the differencing function, both Kronecker products of the identity matrix and the coefficients of the dilation equations. In order to have orthogonality, M_{2^q} must be scaled by $1/\sqrt{2}$.

For the Daubechies wavelet transform matrix, M_q is

$$\mathrm{M}_{2^{\mathrm{q}}} = \begin{bmatrix} L_{2^q} \\ H_{2^q} \end{bmatrix} = \begin{bmatrix} I_{2^{q-1}} \otimes [c_0\ c_1] + J_{2^{q-1}} \otimes [c_2\ c_3] \\ I_{2^{q-1}} \otimes [c_3\ -c_2] + J_{2^{q-1}} \otimes [c_1\ -c_0] \end{bmatrix} \tag{4.35}$$

where c_0, c_1, c_2, c_3 are the coefficients of the dilation equations. Matrix J is a "up-shift" permutation matrix that shifts all elements of a vector up and puts the initial first element to the last position. The filter matrices L_q and H_q are convolution matrices (Toeplitz) but with every other line missing (to achieve the down-sampling). Similar to the Haar case, in order matrix M_q to be orthogonal, it must be scaled by $1/\sqrt{2}$.

4.7 Conclusions - Future Work

The current version of the Kronecker product source-to-source compiler translates matrix expressions into parallel and sequential loops of assignment statements, using term rewriting rules. Since the rules are based on linear algebra identities and program equivalences, there can be conflicting rules that cancel each other's action if applied in succession. For example, the Kronecker product identity

$$A \otimes B = (A \otimes I)(I \otimes B)$$

defines two possible reductions, rewriting the left-hand side of the identity to the right-hand side and vice-versa. The "intelligence" of the system to avoid infinite loops and generate efficient implementations comes from strategy files that are provided by an expert user. In a strategy file, the term rewriting rules are grouped in sets and the order of the application of those sets of rules is defined. Although, the strategy files provide a means to meta-level programming, they require the expertise of the user, moreover they cannot guarantee that the generated code will be the optimal for any parallel or sequential computer. Modeling of the performance of a computing system can provide some clues for the direction of reductions that have to be followed by the Kronecker compiler, however there is no substitute for benchmarks on the target architecture. We would like to extend the Kronecker compiler to perform a search in the space of equivalent code transforms by employing techniques from the field of artificial intelligence. When the state-space of equivalent transforms is relatively small –as it is the case with one dimensional fast transform algorithms–, the compiler will perform a dynamic programming type of search of all states with cost function the actual performance of the generated code on the target architecture. This will require the automation of running and collecting the performance measurements. When a larger space has to be searched, like in the case of multidimensional fast transform algorithms, the compiler will perform an A* type of search, where the choice of the transformation will be based on a heuristic cost function and the depth of the search. When a leaf of the search tree is reached and the actual performance data are collected, the heuristic function will be calibrated accordingly and all open nodes in the search will be re-evaluated with the updated heuristic.

We are interested in other similar transforms like the Gauss transform [GS91], the fractional Fourier transform [BS91], FFT for nonequispaced data [DR93] and generalized FFTs [MR96]. We will try to express them using the Kronecker product notation. Also of interest is the expression of the original FFT algorithm as a multi-grid algorithm [BH90].

Wavelet Transforms have gained a lot of appeal recently in a lot of areas of function approximation applications in signal processing, such as image and video compression and fingerprint matching [Str93]. The vast amounts of data in these applications require fast transform algorithms. In our efforts to build Fast Wavelet Transforms, it would be a waste to have to reinvent all the expertise that so painstakenly has been collected in building FFT's the last 30 years.

Appendix A

Kronecker Product Approximation Codes

A.1 Convergence Issues

We presented a family of iterative algorithms for solving the unconstrained and constrained nearest Kronecker product. Table A.1 gives a summary. For the unconstrained and homogeneous linear constraints, the algorithms of Lanczos and Power Method have been studied extensively and their convergence performance, as well as the subtleties of their failures are well known. The same is also true for the Newton's method used for the non-homogeneous linear and the orthogonality constraints, however our algorithms do not use the exact Newton step but, due to constraints, they are closer to inexact Newton's method.

We did not provide any algorithms for the cases that the constraints are crossing boundaries, like for example requiring B to be orthogonal and C to be symmetric Toeplitz, although it is possible to construct such algorithms.

Table A.1: Algorithm characterization for the Kronecker Product Approximation

Problem-Approach			*Algorithm Type*
Unconstrained	Singular Value Decomposition		Lanczos
	Totally Separable Least Squares		Power Method
Constrained	Linear	Homogeneous	Unconstrained
		Non-Homogeneous	Newton's Method
	Non-linear		Newton's Method

A.2 A Unified MATLAB Front-End

Here is the MATLAB function that solves the constrained Kronecker product approximation and moreover understands keywords for some common constraints like `none`, for the unconstrained case, `symmetric`, `skewsymmetric`, `tril`, `triu` for lower and upper triangular matrix constraints, `diagonal`, `tridiagonal`, `toeplitz`, `circulant`, `hankel`, `markov`, `special` and `orthogonal`.

```
% kronecker product matrix approximation
% nikos 1/23/95
%
% A     : the array to be approximated
% mb,nb : the size of B
%
% returns arrays B,C such that norm(A - kron(B,C),'fro') is minimum

function [B,C] = kpa(A,m1,n1,epsilon,C1,C2,Sp1,Sp2);

[ma,na] = size(A);
m2 = ma/m1; n2 = na/n1;

for k=1:2
  [m,n] = size(eval(['C' int2str(k)]));
  mk = eval(['m' int2str(k)]); nk = eval(['n' int2str(k)]);
  Ta = [];
  for r=1:m
    T = [];
```

```
if findstr(lower(eval(['C' int2str(k) '(r,:)'])),
           'symmetric') == 1
  disp(['Request symmetric']);
  for i=1:min(mk,nk)
    for j=1:i-1
      row = zeros(1,mk*nk);
      row((j-1)*mk+i) = 1;
      row((i-1)*mk+j) = -1;
      T = [T; row];
    end
  end
elseif findstr(lower(eval(['C' int2str(k) '(r,:)'])),
               'skewsymmetric') == 1
  disp(['Request skewsymmetric']);
  for i=1:min(mk,nk)
    for j=1:i-1
      row = zeros(1,mk*nk);
      row((j-1)*mk+i) = 1;
      row((i-1)*mk+j) = 1;
      T = [T; row];
    end
  end
elseif findstr(lower(eval(['C' int2str(k) '(r,:)'])),
               'tril') == 1
  disp(['Request lower triangular']);
  for i=1:mk
    for j=i+1:nk
      row = zeros(1,mk*nk);
      row((j-1)*mk+i) = 1;
      T = [T; row];
    end
  end
elseif findstr(lower(eval(['C' int2str(k) '(r,:)'])),
               'triu') == 1
  disp(['Request upper triangular']);
  for i=2:mk
    for j=1:i-1
      row = zeros(1,mk*nk);
      row((j-1)*mk+i) = 1;
      T = [T; row];
    end
  end
elseif findstr(lower(eval(['C' int2str(k) '(r,:)'])),
               'diagonal') == 1
  disp(['Request diagonal']);
  for i=1:mk
    for j=1:nk
      if i~=j
        row = zeros(1,mk*nk);
        row((j-1)*mk+i) = 1;
        T = [T; row];
      end
    end
  end
elseif findstr(lower(eval(['C' int2str(k) '(r,:)'])),
               'tridiagonal') == 1
  disp(['Request tridiag']);
  for i=1:mk
    for j=1:nk
      if abs(i-j)>1
        row = zeros(1,mk*nk);
        row((j-1)*mk+i) = 1;
        T = [T; row];
      end
    end
  end
elseif findstr(lower(eval(['C' int2str(k) '(r,:)'])),
               'toeplitz') == 1
  disp(['Request toeplitz']);
  for j=1:nk-1
    for i=1:mk-1
      row = zeros(1,mk*nk);
      row((j-1)*mk+i) = 1;
      row(j*mk+i+1) = -1;
      T = [T; row];
    end
  end
elseif findstr(lower(eval(['C' int2str(k) '(r,:)'])),
               'circulant') == 1
  disp(['Request circulant']);
  if mk ~= nk
    error('Matrix has to be square')
  end
  C=toeplitz([1;(nk:-1:2)'],1:nk);
  for j=1:nk
    idxs = find(C == j);
    for i=2:length(idxs)
      row = zeros(1,mk*nk);
      row(idxs(1)) = 1;
      row(idxs(i)) = -1;
```

```
          T = [T; row];
        end
      end
    elseif findstr(lower(eval(['C' int2str(k) '(r,:)'])),
                   'hankel') == 1
      disp(['Request hankel']);
      for j=1:nk-1
        for i=2:mk
          row = zeros(1,mk*nk);
          row((j-1)*mk+i) = 1;
          row(j*mk+i-1) = -1;
          T = [T; row];
        end
      end
    elseif findstr(lower(eval(['C' int2str(k) '(r,:)'])),
                   'special') == 1
      disp(['Request special']);
      T = eval(['Sp' int2str(k)])';
    elseif findstr(lower(eval(['C' int2str(k) '(r,:)'])),
                   'none') == 1
      disp(['Request none']);
      % T = []; zeros(1,m2*n2)
    else
      disp(['Unrecognized request:' eval(['C' int2str(k) '(r,:)'])])
    end
    Ta = [Ta; T];
  end
  eval(['T' int2str(k) '= Ta;']);
end

if findstr(lower(C1(1,:)),'none') == 1 &
   findstr(lower(C2(1,:)),'none') == 1
  [B,C] = unconstr_kpa(A,m1,n1,epsilon);
else
  T1 = makefullrank(T1);
  T2 = makefullrank(T2);
  [B,C] = constr_kpa_h(A,m1,n1,T1',T2');
end

return
```

A.3 Strong Kronecker product

Given two matrices A and B written as n_1-by-n_2 and n_2-by-n_3 block matrices respectively, the n_1-by-n_3 block matrix C is said to be the strong Kronecker product of A and B if and only if, each block of C can be written as

$$C_{ij} = \sum_{k=1}^{n_2} (A_{ik} \otimes B_{kj}).$$

and denote the strong Kronecker product of A and B with $A \circ B$.

```
function C = strongKron(A,B,mA,nA,mB,nB)
% calculate the strong Kronecker product of A and B
% mA,nA is the block size of A
% mB,nB is the block size of B

[m,n] = size(A); n1 = m/mA; n2 = n/nA;
[m,n] = size(B); n22 = m/mB; n3 = n/nB;

if n2 ~= n22
  error('Incompatible block sizes');
end

C = zeros(n1*mA*mB,n3*nA*nB);
for i=1:n1
  for j=1:n3
    CC = zeros(mA*mB,nA*nB);
    for k=1:n2
      Aik = A((i-1)*mA+1:i*mA,(k-1)*nA+1:k*nA);
      Bkj = B((k-1)*mB+1:k*mB,(j-1)*nB+1:j*nB);
      CC = CC + kron(Aik,Bkj);
    end
    C((i-1)*mA*mB+1:i*mA*mB, (j-1)*nA*nB+1:j*nA*nB) = CC;
  end
end
```

A.3.1 Approximating with a Strong Kronecker product

Here is a direct way to approximate a given matrix C with a strong Kronecker product $A \circ B$ of matrices with predefined block sizes, using the singular value decomposition. Note that if the user does not specify the rank of the approximation, then the exact strong Kronecker product is calculated.

```
function [A,B,r] = strongKronApprox(C,mA,nA,mB,nB,n2)
% approximate C with the strong Kronecker product of A and B
% mA,nA is the block size of A
% mB,nB is the block size of B

[m,n] = size(C);
n1 = m/(mA*mB); mbint(n1);
n3 = n/(nA*nB); mbint(n3);

Ch = zeros(n1*mA*nA,n3*mB*nB);
for i=1:n1
  for j=1:n3
    Ch((i-1)*mA*nA+1:i*mA*nA,(j-1)*mB*nB+1:j*mB*nB) = ...
        tilde(C((i-1)*mA*mB+1:i*mA*mB, (j-1)*nA*nB+1:j*nA*nB),mA,nA);
  end
end

r = rank(Ch);
if nargin == 5
  n2 = r;
elseif nargin == 6
  if r < n2, n2 = r; end
else
  error('Wrong number of input arguments');
end

[U,S,V] = svd(Ch); s = sqrt(diag(S));
X = U(:,1:n2)*diag(s(1:n2));
Y = (V(:,1:n2)*diag(s(1:n2)))';

A = zeros(n1*mA,n2*nA); A0 = zeros(mA,nA);
for i=1:n1
  for j=1:n2
    A0(:) = X((i-1)*mA*nA+1:i*mA*nA,j);
    A((i-1)*mA+1:i*mA,(j-1)*nA+1:j*nA) = A0;
  end
end

B = zeros(n2*mB,n3*nB); B0 = zeros(mB,nB);
for i=1:n2
  for j=1:n3
    B0(:) = Y(i,(j-1)*mB*nB+1:j*mB*nB);
    B((i-1)*mB+1:i*mB,(j-1)*nB+1:j*nB) = B0;
  end
end
```

A.4 Lanczos SVD

As we have already seen, the Kronecker product approximation of a very large sparse matrix A can be found using the Lanczos SVD. Such a matrix is not explicitly represented, rather a subroutine is given that calculates the product of A with a given input vector x. The code `HugeAX` demonstrates such an instance. In this example A is a tridiagonal matrix with main diagonal 4, subdiagonal -2 and superdiagonal -1.

```
c
      subroutine HugeAX (m, n, x, w)
c
c     computes  w <- A*x
c
      integer          m, n, i,mn
      Double precision x(n), w(m)
c
      do i=1,m
         w(i) = 0.d0
      end do

      w(1) = 4.d0*x(1) - 1.d0*x(2)
      mn = min(m,n)
      do i=2,mn-1
         w(i) = 4.d0*x(i) - 2.d0*x(i-1) - 1.d0*x(i+1)
      end do
      w(mn) = 4.d0*x(mn) - 2.d0*x(mn-1)

      if (m .lt. n) then
         w(mn) = w(mn) - 1.d0*x(mn+1)
      endif
      if (m .gt. n) then
         w(mn+1) = - 2.d0*x(mn)
      endif
c
      return
      end
```

For the Kronecker product approximation though, we need $\tilde{A}$. Given the above routine, the following two routines implement the application of $\tilde{A}$ and $\tilde{A}^T$ to a vector x, using only calls to HugeAX.

```
      subroutine TildeAX (m, n, mb, nb, mc,nc,x, y)
c     compute  y = tilde(A,mb,nb)*x
c       m : rows of Atilde
c       n : cols of Atilde
      implicit none
      integer          m, n, mb, nb, mc, nc
      Double precision x(n), y(m)

      integer          maxm, maxn
      parameter       (maxm = 500, maxn=250)

      integer          i, j, k, jj
      Double precision z(maxm,maxm), e(maxm)

      do i=1,m
         y(i) = 0d0
      enddo

      do i=1,nb
         k = (i-1)*mb+1
         do j=1,nc
            do jj=1,nb*nc
               e(jj) = 0.d0
            end do
            e((i-1)*nc+j) = 1.d0

            call HugeAX(mb*mc,nb*nc,e,z)

            call dgemv('T',mc,mb,1.d0,z,mc,x((j-1)*mc+1),1,1.d0,y(k),1)
         enddo
      enddo

      return
      end

      subroutine TildeATX (m, n, mb, nb, mc, nc, x, y)
      implicit none
c
c     computes  y <- tilde(A,mb,nb)'*x
c
      integer          m, n, mb, nb, mc, nc
```

```
Double precision x(m), y(n)

integer          maxm, maxn
parameter       (maxm = 500, maxn=250)

integer          i, j, k,jj
Double precision z(maxm,maxm), e(maxm)

do i=1,n
   y(i) = 0d0
enddo

do i=1,nc
   k = (i-1)*mc+1
   do j=1,nb
      do jj=1,nb*nc
         e(jj) = 0.d0
      end do
      e((j-1)*nc+i) = 1.d0
      call HugeAX(mb*mc,nb*nc,e,z)
      call dgemv('N',mc,mb,1.d0,z,mc,x((j-1)*mb+1),1,1.d0,y(k),1)
   enddo
enddo

return
end
```

Note that both `TildeAX` and `TildeATX` are very expensive to use. We strongly recommend the user to code her own version of the above routines for the specific matrix A. The routines `TildeAX` and `TildeATX` allow the use of any iterative SVD package that requires the matrix to be factored to be defined as a subroutine.

Sometimes the sparsity structure of A does not permit us to express it as a subroutine. Instead only the non-zero elements of A are stored. There are many different encodings to store a sparse matrix. For the code that accompanies this thesis we have located some MATLAB functions that allow you to read different sparse matrix formats as MATLAB sparse matrices. We have modified our `tilde` and `lanczos` MATLAB code to work with large sparse matrices. In the case that reorthogonalization is need to preserve accuracy in baddly conditioned problems, we include the Lanczos code with partial and complete reorthogonalization from the collection `regutools` [Han94]. If the user needs the superior performance of compiled code, we suggest the use of SVDPACK [Ber80].

A.5 Least Squares with Inequality and Equality Constraints

The routines are based in the algorithms presented in Lawson and Hanson [LH74].

A.5.1 Non-Negative Least Squares

The non-negative least squares problem (NNLS) is the optimization problem

$$\min_{x \geq 0} \|Ax - b\|_2.$$

```
% Minimize norm(A*x-b) s.t. x >= 0
% Non-Negative Least Squares
function [x] = nnls(A,b)

[m,n] = size(A);
epsilon = 1e-12;
x = zeros(n,1); P = zeros(1,n); Z = ones(1,n);

while 1
  w = A'*(b - A*x);
  fZ = find(Z);
  if fZ==[] | all(w(fZ)<0 | w(fZ) <= epsilon), return; end
  [mx, t] = max(w(fZ)); t = fZ(t);
  Z(t) = 0; P(t) = 1;

  while 1
    z = zeros(n,1);
    fP = find(P); z(fP) = A(:,fP)\b;

    if all(z(fP)>0)
      x = z;
      break;
    end
    j = find(z(fP)<0 | z(fP)<epsilon); j = fP(j);
    alpha = min(x(j) ./ (x(j) - z(j)));
    x = x + alpha*(z - x);
    j = find(abs(x(fP))<=epsilon); j = fP(j);
    P(j) = zeros(length(j),1);
    Z(j) = ones(length(j),1);
  end
end

return
```

A.5.2 Least Distance Programming

The least distance programming problem (LDP) is finding the minimizer of

$$\min_{Gx \geq h} \|x\|_2.$$

There a reduction of LDP to NNLS.

```
% Minimize norm(x) s.t. G*x >= h
% Least Distance Programming
function [x] = ldp(G,h)

[m,n] = size(G);

E = [G'; h'];
f = [zeros(n,1); 1];
u = nnls(E,f);
r = E*u - f;

if norm(r)>1e-12
  x = -r(1:n)/r(n+1);
end

return
```

A.5.3 Least Squares with Linear Inequality Constraints

The LSI problem

$$\min_{Gx \geq h} \|Ex - f\|_2$$

is reducible to an LDP.

```
% Minimize norm(E*x-f) s.t. G*x >= h
% Least Squares with Inequality Constraints
function [x] = lsi(E,f,G,h)

[H,R,K,k] = hrk(E);
H1 = H(:,1:k); R11 = R(1:k,1:k); K1 = K(:,1:k);
A = G*K1*inv(R11);
z1 = ldp(A, h - A*(H1'*f));

y1 = R11 \ (z1 + H1'*f);
x = K1*y1;

return
```

A.5.4 Least Squares with Linear Equality and Inequality Constraints

The LSIE problem

$$\min_{Gx \geq h \; Cx = d} \|Ex - f\|_2$$

is reduced to an LSI problem.

```
% Minimize norm(E*x-f) s.t. G*x >= h and C*x == d
% Least Squares with Inequality and Equality Constraints
function [x] = lsie(E,f,G,h,C,d)

[m,n] = size(C);
[Q,R] = qr(C');

M = E*Q; E1 = M(:,1:m); E2 = M(:,m+1:n);
M = G*Q; G1 = M(:,1:m); G2 = M(:,m+1:n);

y1 = R(1:m,1:m)' \ d;
y2 = lsi(E2, f - E1*y1, G2, h - G1*y1);

x = Q*[y1; y2];

return
```

A.6 Solving the secular equation problem

The secular equation problem is of the form $(A+\lambda I)x = b$, where λ and x are the unknowns and they depend on each other. The solution of such a problem comes up on the following situation; suppose we want to solve the optimization problem

$$\min_{\|x\|_2 <= 1} \|Ax - b\|_2. \tag{A.1}$$

If the solution of the unconstrained least squares problem $x_{LS} = A^{+}b$ is inside the unit ball i.e. $\|x_{LS}\|_2 <= 1$ then we are done, otherwise, due to continuity, the best x must reach the unit ball, therefore we need to solve

$$\min_{\|x\|_2 = 1} \|Ax - b\|_2. \tag{A.2}$$

that is equivalent to unconstrained problem $\min \phi(x, \lambda)$ with

$$\phi(x,\lambda) = \frac{1}{2}\|Ax - b\|_2 + \frac{1}{2}\lambda(x^T x - 1) \tag{A.3}$$

where λ is the Lagrange multiplier. The gradient of $\phi()$ with respect to x is

$$\frac{\partial \phi}{\partial x} = A^T A x - A^T b + \lambda x \tag{A.4}$$

which when set to zero gives the secular equation $(A^T A + \lambda I)x = A^T b$. The problem is solved as follows. Let $A = U\Sigma V^T$ the singular value decomposition of A, then the secular equation can be written as $(V\Sigma^T \Sigma V^T + \lambda V V^T)x = V\Sigma^T U^T b$ that is equivalent to $(\Sigma^T\Sigma + \lambda I)V^T x = \Sigma^T U^T b$. With a change of variables $y = V^T x$ and $c = \Sigma^T U^T b$, we get the following scalar relation, $(\sigma_i^2 + \lambda) y_i = c_i$ for $i = 1, \cdots, \min(m, n)$ where m and n are the dimensions of A.

Since $\|x\|_2^2 = 1$ then $\|V^T x\|_2^2 = 1$, therefore $\|y\|_2^2 = 1$ and the following must hold:

$$\sum_{i=1}^{\min(m,n)} \left(\frac{c_i}{\sigma_i^2 + \lambda}\right)^2 = 1. \tag{A.5}$$

The above is a non-linear equation with unknown λ, and can be solved using the Newton iteration

$$\lambda_+ = \lambda + \frac{\sum_{i=1}^{\min(m,n)} (\frac{c_i}{\sigma_i^2+\lambda})^2 - 1}{2\sum_{i=1}^{\min(m,n)} \frac{c_i^2}{(\sigma_i^2+\lambda)^3}}. \tag{A.6}$$

A good initial guess for λ is zero. The MATLAB function that solves the secular problem is

```
% solve min(norm(A*x - b))
% s.t. norm(x)<=1
function [x] = secular(A,b)

x = A\b;
if norm(x)<=1
  return
end

[m,n] = size(A); mn = min(m,n);
[U,S,V] = svd(A); sigma = diag(S);
c = S'*U'*b; c = c(1:mn);

lambda = 0;
fl = sum( (c(:)./(sigma(:).^2 + lambda)).^2) - 1;

iter = 0;
while abs(fl) >= 1e-8
  iter = iter + 1;
  lambda = lambda + 0.5*fl/sum(c(:).^2./(sigma(:).^2 + lambda).^3);
  fl = sum( (c(:)./(sigma(:).^2 + lambda)).^2) - 1;
end
x = (A'*A + lambda*eye(n)) \ A'*b;
```

Appendix B

Code Generation Programs

In this Chapter we list the code used to symbolically generate FFT and other transforms, as well as give a small manual of directions on how to use those programs.

B.1 Radix Generation

The radix generation has been described in section 4.3. We used Maple [Wat94] for the symbolic manipulation necessary for the loop unrolling because of the clear programming interface Maple provides for loops and indices.

The code that performs everything follows. It uses the split radix algorithm, since this is the FFT that provides the best (lowest) flop count

```
readlib(write):
with (linalg):
#-------------------------------------------------------------------

# Initialize all work space, reset index counters
reset_index :=
   proc ()
      global T, flop_count, temp_index, FirstTime;
      T := 'T';
      flop_count := 0;
      temp_index := -1;
      FirstTime := true
   end:

# Provide a new index for the generation of a new temporary variable
new_index :=
   proc ()
      global temp_index;
      temp_index := temp_index+1;
      temp_index
   end:

# Perform a stride permutation on the input vector
StridePerm :=
   proc (n,p,x)
      local j, k, l, r;
      r := array(0..(n-1));
      l := 0;
      for j from 0 to (p-1) do
         for k from j by p to (n-1) do
            r[l] := x[k]; l := l+1;
         od
      od;
      op(r);
   end:

# Generate a new name for a temporary variable
new_temp :=
   proc (expr)
      local n, temp;
      global T, t, Vector_T;

      if evalb(Vector_T) then
         temp := t[new_index()];
      else
        temp := t.(new_index());
      fi;
      T[temp] := expr;
      temp
   end:

# Routine that handles the in-place twiddle factor multiplications
# It contains the optimizations related to the type of the multiplier
multip_flops :=
   proc (m,temp)
```

```
      global flop_count, T;
      local zr, zi;

      zr := evalf(evalc(Re(m)),16); zi := evalf(evalc(Im(m)),16);
#      print(`On Input `,m,temp,T[temp]);
      if zr = 1.0 then
         # do nothing
      elif zr = -1.0 then # do it in place
         T[temp] := -T[temp]
      elif zi = 1.0 then
         T[temp] := dcmplx(-dimag(T[temp]), dreal(T[temp]));
      elif zi = -1.0 then
         T[temp] := dcmplx( dimag(T[temp]),-dreal(T[temp]));
      elif zr = zi then
         T[temp] := dcmplx(zr*dreal(T[temp])-zr*dimag(T[temp]),
                           zr*dreal(T[temp])+zr*dimag(T[temp]));
         flop_count := flop_count + 4
      elif zr = -zi then
         T[temp] := dcmplx(zr*dreal(T[temp])+zr*dimag(T[temp]),
                          -zr*dreal(T[temp])+zr*dimag(T[temp]));
         flop_count := flop_count + 4
      else
         T[temp] := evalf(dcmplx(zr,zi),16)*T[temp];
         flop_count := flop_count + 6;
#         print(m)
      fi;
#      print(`Generated `, T[temp]);
   end:


# The Split Radix algorithm that operates symbolically on its input
split_radix :=
   proc (n,x)
      local n1,n2, m,xt,xb,yt,yb,zt,zb,j,px,p, Dp, Dp3,u,v;
      global T, flop_count;
      # option trace;
      if n=1 then
         convert(x,list)
      elif n=2 then
         flop_count := flop_count + 4;
         [new_temp(x[0]+x[1]), new_temp(x[0]-x[1])]
      else
         px := StridePerm(n,2,x);
         m := n/2;
         p := n/4;
```

```
xt := array(0..m-1); xb := array(0..m-1);
for j from 0 to (m-1) do
   xt[j] := px[j];
   xb[j] := px[m+j]
od;

xt := array(0..m-1,split_radix(m,xt));

yt := array(0..p-1); yb := array(0..p-1);
for j from 0 to (p-1) do
   yt[j] := xt[j];
   yb[j] := xt[p+j]
od;

xb := StridePerm(m,2,xb);
zt := array(0..p-1); zb := array(0..p-1);
for j from 0 to (p-1) do
   zt[j] := xb[j];
   zb[j] := xb[p+j]
od;
zt := array(0..p-1,split_radix(p,zt));
zb := array(0..p-1,split_radix(p,zb));

Dp := array(0..p-1);
for j from 0 to (p-1) do Dp[j] := exp(-2*Pi*j*I/n) od;
Dp3 := array(0..p-1);
for j from 0 to (p-1) do Dp3[j] := exp(-2*Pi*3*j*I/n) od;

for j from 0 to p-1 do
   multip_flops(Dp[j], zt[j]);
   multip_flops(Dp3[j],zb[j]);
od;

u := array(0..p-1); v := array(0..p-1);
for j from 0 to p-1 do
   u[j] := new_temp(zt[j] + zb[j]);
   v[j] := new_temp(zt[j] - zb[j]);
   flop_count := flop_count + 4;
   multip_flops(I,v[j])
od;

for j from 0 to p-1 do
   px[j]     := new_temp(yt[j] + u[j]);
   px[p+j]   := new_temp(yb[j] - v[j]);
   px[2*p+j] := new_temp(yt[j] - u[j]);
```

```
            px[3*p+j] := new_temp(yb[j] + v[j]);
            flop_count := flop_count + 8
         od;
         convert(px,list)
      fi
   end:

# norm(F(n)- kron(F(p),I(m))*diag(d)*kron(I(p),F(m))*P(n,p))

# The Final Code generation routine
# It generates the radix in Fortran 77
gen_code :=
   proc (z,n,many)
      local j, k, temp, on_final;
      global temp_index, Vector_V, Vector_T, store_out;
      # option trace;

open(`Fortran_Radices/FFT`.n.`-`.many.`.f`);

writeln(`      subroutine fft(x,n,z,maxn,Wlong)`);
writeln(`      integer n, many`);
writeln(`      complex*16 z(0:(n-1)), x(0:n-1)`);
write  (`      parameter (nn = `.n);
write  (`, many = `.many);
writeln(`)`);
writeln(`      complex*16 Wlong(1:nn-1,0:(maxn-1)/(nn-1)-1)`);

writeln(`      integer tau,q,j,k,L,r,Ls,rs,nu`);
writeln(`      complex*16 y(0:nn**many-1)`);

if Vector_V then
  writeln(`      complex*16 v(0:nn-1)`);
else
  writeln(`      implicit complex*16(v)`);
fi;
if store_out then
  writeln(`      complex*16 w(0:nn-1)`);
fi;
if Vector_T then
  writeln(`      complex*16 t(0:`.temp_index.`)`);
else
  writeln(`      implicit complex*16(t)`);
fi;

writeln(`      complex*16 i`);
writeln('      real*8 pi');
writeln('      parameter(pi = 3.14159265358979d0, i = (0d0,1d0))');
writeln('      tau = log(real(n))/log(real(nn))');

writeln('      do j=0,n-1');
writeln('         z(j) = x(j)');
writeln('      enddo');

writeln('      L = 1');
writeln('      do q = 1,tau');
writeln('         L = L*nn');
writeln('         r = n/L');
writeln('         Ls = L/nn');
writeln('         rs = nn*r');
writeln('         nu = (Ls-1)/(nn-1)');
writeln('         do j=0,n-1');
writeln('            y(j) = z(j)');
writeln('         enddo');
writeln('         do j = 0,Ls-1');
writeln('            do k = 0,r-1');

if Vector_V then
  writeln('            v(0) = y(j*rs+k)');
  writeln('            do kk = 1,nn-1');
  writeln('               v(kk) = Wlong(kk,nu+j) * y(j*rs+kk*r+k)');
  writeln('            enddo');
else
  writeln('       v0 = y(j*rs+k)');
  for j from 1 to n-1 do
    writeln('    v'.j.' = Wlong('.j.',nu+j) * y(j*rs+'.j.'*r+k)');
  od
fi;
      on_final := table(sparse);
      for j from 1 to n do on_final[z[j]] := j od;

      for j from 0 to temp_index do
         if evalb(Vector_T) then
           temp := t[j];
         else
           temp := t.j;
         fi;

         if on_final[temp]=0 then
            write(fortran([temp = T[temp]],digits=16));
         else
```

```
            if store_out then
              write(fortran([`w(`.(on_final[temp]-1).`)` = T[temp]]));
            else
             write(fortran(
               [`z((j+`.(on_final[temp]-1).`*Ls)*r+k)` = T[temp]]))
            fi
         fi
      od;
      if store_out then
  writeln(`              do kk = 0,nn-1`);
  writeln(`                 z((j+kk*Ls)*r+k) = w(kk)`);
  writeln(`              enddo`);
      fi;

writeln(`            enddo`);
writeln(`         enddo`);
writeln(`      enddo`);

writeln(`      return`);
writeln(`      end`);
close()

   end:

log2 := proc (n) log(n)/log(2) end:

nlogn_coef := proc (fc,n) (fc+(n-1)*6)/(n*log2(n)) end:

#############################################################
# Main Program that controls everything
#############################################################
reset_index():

Vector_V := true;
Vector_T := false;
store_out:= true;

if Vector_T then
  t :=array(0..10000);
else
  t := 't';
fi;

n := 4; many := floor(evalf(log[n](2^20)));
```

```
if Vector_V then
  v := array(0..n-1);
  z := split_radix(n,v):
else
  x := array(0..n-1);
  for j from 0 to n-1 do x[j] := `v`.j od:
  z := split_radix(n,x):
fi;

precision := 'double';
gen_code(z,n,many);
print(`Flop Count = `.flop_count);

evalf(nlogn_coef(flop_count, n));

quit;
```

A user need only change the constants defined in the main program. Here is an explanation of their semantics

`n` : the size of the radix.

`precision` the required precision of the variables, possible values are 'double' and 'single'.

`Vector_V` : when set to true, the input subvector values required by the radix calculation are copied to a temporary vector `v`, so that the radix code need not access the whole data vector. This option is needed in the case of providing a stand alone subroutine that only performs the radix calculation. For a high performance setting, the preffered value of `Vector_V` is False.

`Vector_T` : when set to true, the temporary variables needed for the intermediate values of the generated radix FFT are stored on a local temorary vector `v`. The effect of such a choice is apparent when a large radix FFT is generated, where a clever fortran compiler might provide some data prefetching.

For a small radix FFT, `Vector_T` should be set to false, because the scalar temporaries usually trigger the register allocation.

`store_out` Similar to `Vector_V`, only now it refers to output subvector. The same comments as to `Vector_V` apply.

Different combinations of the values for the last 3 variables resulted to big differences in the performance of the generated code, we advise the user to experiment with the given problem sizes and target hardware/compiler.

B.2 Loop Nest Generation

The loop nest generation is done via code transformation using Mathematica. The term rewriting rules are listed below together with some comments explaining their purpose, proper use and limitations. For a higher level description of this material, the reader should see Chapter 4.

We start by initializing some counters for the generation of unique loop index and variable names.

```
count = -1
newint = Function[ , count = count+1]
newvar = Function[ , ToExpression[StringJoin["i",ToString[newint[]]]]]
```

Rules to determine the size of a square matrix or a vector.

```
(* Size of a square matrix *)
SIZE[B[j_,k_]] := Simplify[j*k]
SIZE[F[n_]] := n
SIZE[I[n_]] := n
SIZE[P[m_,n_]] := m
SIZE[KRON[A_,B_]] := SIZE[A] * SIZE[B]
SIZE[TIMES[A_,B_]] := SIZE[A]       (* Assume square matrices *)
SIZE[PROD[v_, IDX[f_,s_,l_], M_]] := SIZE[M /. v :> f]
SIZE[TRANSP[A_]] := SIZE[A]
SIZE[A_[n_]] := n
(* Size of a vector *)
```

```
SIZE[VEC[x_,idx_]] := SIZE[idx]
SIZE[IDX[f,s,l]] := (f-l+1)/s /; Not[s == 0]
```

Rules that simplify a transposed matrix. Note the direct correspondence with linear algebra identities.

```
TRANSP[F[n_]] := F[n]
TRANSP[P[m_,n_]] := P[m,n/m]
TRANSP[I[n_]] := I[n]
TRANSP[W[n_,p_]] := W[n,p]
TRANSP[TIMES[A_,B_]] := TIMES[TRANSP[B],TRANSP[A]]
TRANSP[PROD[v_, IDX[f_,s_,l_], M_]] := PROD[v, IDX[l,-s,f], TRANSP[M]]
TRANSP[KRON[A_,B_]] := KRON[TRANSP[A],TRANSP[B]]
```

A set of rules to perform index operations.

```
MODIDX[IDX[f_,s_,l_], n_] :=
  IDX[MOD[Simplify[f],n],MOD[Simplify[s],n],MOD[Simplify[l],n]]

FLOORIDX[IDX[f_,s_,l_], n_] :=
  IDX[FLOOR[Simplify[f],n],FLOOR[Simplify[s],n],FLOOR[Simplify[l],n]]

FLOOR[j, n^(q-1)] := 0
FLOOR[n^(q-1), n^(q-1)] := 1
FLOOR[j-n^(q-1)+n^q, n^(q-1)] := n-1

FLOOR[k*n, n] := k
FLOOR[1, n] := 0
FLOOR[-1 + n + k*n, n] := k

FLOOR[k + j*n^(-q+t), n^(-1+t)] := 0
FLOOR[n^(-1+t), n^(-1+t)] := 1
FLOOR[k - n^(-1+t) + n^t + j*n^(-q+t), n^(-1+t)] := n-1

MOD[j, n^(q-1)] := j
MOD[n^(q-1), n^(q-1)] := 0
MOD[j-n^(q-1)+n^q, n^(q-1)] := j

MOD[k*n, n] := 0
MOD[1, n] := 1
MOD[-1 + n + k*n, n] :=  n-1

MOD[k + j*n^(-q+t), n^(-1+t)] := k+j*n^(t-q)
MOD[n^(-1+t), n^(-1+t)] :=  0
MOD[k - n^(-1+t) + n^t + j*n^(-q+t), n^(-1+t)] := k+j*n^(t-q)
```

```
REINDEX[VEC[x_,IDX[f1_,s1_,l1_]], IDX[f2_,s2_,l2_]] :=
  VEC[x,IDX[f1+f2*s1,s1*s2,f1+l2*s1]]

INVIDX[IDX[f1_,s1_,l1_], IDX[f_,s_,l_]] :=
  IDX[(f-f1)/s1,s/s1,(l-f1)/s1]

IDX/: IDX[f1_,s1_,l1_] + IDX[f2_,s2_,l2_] := IDX[f1+f2,s1+s2,l1+l2]

IDX/: a_ * IDX[f1_,s1_,l1_] := IDX[a*f1,a*s1,a*l1]
```

The set of expanding rules that are independently named so we can refer to them.

```
(* Expanding Rules *)
butterfly = B[j_,k_] :> TIMES[KRON[F[j],I[k]], W[j,k]]

matmatvec = ASSIGN[x_, TIMES[A_, TIMES[B_, y_]]] :>
            SEQ[ASSIGN[x, TIMES[B, y]],
                ASSIGN[x, TIMES[A, x]]]

weights   = ASSIGN[y_, TIMES[W[j_,k_], VEC[x_,idx_]]] :>
            ASSIGN[y,DOTSTAR[VEC[w,IDX[0,1,j*k-1]],VEC[x,idx]]]

product   = ASSIGN[x_, TIMES[PROD[q_,IDX[f_,s_,l_],M_], x_]] :>
            LOOP[q,IDX[l,-s,f],
                 ASSIGN[x ,TIMES[M,x]]]

IkronA    = ASSIGN[x_,TIMES[KRON[I[p_],M_], y_]] :>
              ( v = Switch[p,
                          n^(t-q), k,
                          n^(q-1), j,
                          _, newvar[]];
                sz = SIZE[M];
                newidx = IDX[v*sz,1,(v+1)*sz-1];
              PARALLEL[v, IDX[0,1,p-1],
                   ASSIGN[REINDEX[x, newidx],
                          TIMES[M,REINDEX[y, newidx]]]] )

AkronI    = ASSIGN[x_,TIMES[KRON[M_,I[m_]], VEC[y_,idx_]]] :>
              ( v = Switch[m,
                          n^(t-q), k,
                          n^(q-1), j,
                          _, newvar[]];
                sz = SIZE[M];
```

```
                  newidx = IDX[v,m,v+m*(sz-1)];
                PARALLEL[v, IDX[0,1,m-1],
                     ASSIGN[REINDEX[x, newidx],
                            TIMES[M,REINDEX[VEC[y,idx], newidx]]]] )
```

The corresponding set of contracting rules.

```
(* Contracting Rules *)

pushweight = {SEQ[ASSIGN[VEC[x_,i1_], DOTSTAR[d_,y_]],
                    PARALLEL[j_, idx_,
                        ASSIGN[z_,TIMES[A_,VEC[x_,i2_]]]]] :>
                PARALLEL[j, idx,
                     SEQ[ASSIGN[REINDEX[VEC[x,i1],INVIDX[i1,i2]],
                                DOTSTAR[REINDEX[d,INVIDX[i1,i2]],
                                        REINDEX[y,INVIDX[i1,i2]]]],
                         ASSIGN[z,TIMES[A,VEC[x,i2]]]]],
                SEQ[PARALLEL[j_, idx_,
                        ASSIGN[z_,TIMES[A_,VEC[x_,i2_]]]],
                    ASSIGN[VEC[x_,i1_], DOTSTAR[d_,y_]]] :>
                PARALLEL[j, idx,
                     SEQ[ASSIGN[z,TIMES[A,VEC[x,i2]]],
                         ASSIGN[REINDEX[VEC[x,i1],INVIDX[i1,i2]],
                                DOTSTAR[REINDEX[d,INVIDX[i1,i2]],
                                        REINDEX[y,INVIDX[i1,i2]]]]]]}

pushstride ={SEQ[ASSIGN[VEC[x_,idx1_], TIMES[P[l_,n_], z1_]],
                  PARALLEL[v_,anyidx_,
                   SEQ[ASSIGN[z2_,DOTSTAR[d_, VEC[x_, idx2_]]],
                           anystat_]]] :>
                ( m = l/n;
                  idx = INVIDX[idx1, idx2];
                SEQ[ASSIGN[VEC[y,IDX[0,1,l-1]], z1],
                    PARALLEL[v, anyidx,
                         SEQ[ASSIGN[VEC[x, idx2],
                                    VEC[y,MODIDX[idx,m]*n+
                                                 FLOORIDX[idx,m]]],
                              SEQ[ASSIGN[z2,DOTSTAR[d, VEC[x, idx2]]],
                                  anystat]]]] ),

                SEQ[ASSIGN[VEC[x_,idx1_], TIMES[P[l_,n_], z1_]],
                    PARALLEL[v1_,idxv1_,
                         PARALLEL[v2_,idxv2_,
                        SEQ[ASSIGN[z2_,DOTSTAR[d_, VEC[x_, idx2_]]],
                                  anystat_]]]] :>
                  ( m = l/n;
                  idx = INVIDX[idx1, idx2];
                SEQ[ASSIGN[VEC[y,IDX[0,1,l-1]], z1],
                    PARALLEL[v1, idxv1,
                         PARALLEL[v2, idxv2,
                         SEQ[ASSIGN[VEC[x, idx2],
                                    VEC[y,MODIDX[idx,m]*n+
                                                 FLOORIDX[idx,m]]],
                             SEQ[ASSIGN[z2,DOTSTAR[d, VEC[x, idx2]]],
                                 anystat]]]]] ),

SEQ[ASSIGN[VEC[x_,idx1_], TIMES[P[l_,n_], z1_]],
    PARALLEL[v1_,idxv1_, ASSIGN[z2_,OPER_[d_, VEC[x_, idx2_]]]]] :>
                ( m = l/n;
                  idx = INVIDX[idx1, idx2];

SEQ[ASSIGN[VEC[y,IDX[0,1,l-1]], z1],
    PARALLEL[v1,idxv1,
             SEQ[ASSIGN[VEC[x, idx2],
                        VEC[y,MODIDX[idx,m]*n+FLOORIDX[idx,m]]],
                 ASSIGN[z2,OPER[d, VEC[x, idx2]]]]]])
}

loopchange = SEQ[PARALLEL[j_,idxj_,
                        ASSIGN[VEC[x_,idxx1_],rhs1_]],
                 PARALLEL[k_,idxk_,
                   PARALLEL[j_,idxj_,
                     SEQ[ASSIGN[VEC[x_,idxx2_], rhs2_],stats_]]]] :>
               SEQ[PARALLEL[j,idxj,
                          ASSIGN[VEC[x,idxx1],rhs1]],
                   PARALLEL[j,idxj,
                          PARALLEL[k,idxk,
                               SEQ[ASSIGN[VEC[x,idxx2], rhs2],stats]]]]

loopjoin =   SEQ[PARALLEL[j_,idxj_, stats1_],
                 PARALLEL[j_,idxj_, stats2_]] :>
               PARALLEL[j,idxj,
                    SEQ[stats1, stats2]]

strideperm = ASSIGN[x_, TIMES[P[s_,p_], x_]] :>
             (  v = Switch[p,
                          n^(t-q), k,
                          n^(q-1), j,
                          _, newvar[]];
               m = Simplify[s/p];
               SEQ[ASSIGN[VEC[y, IDX[0,1,s-1]], x],
                    PARALLEL[v, IDX[0, 1, p-1],
                         ASSIGN[REINDEX[x, IDX[v*m, 1, (v+1)*m-1]],
                                    VEC[y, IDX[v, p, v+(m-1)*p]]]]] )

assignjoin = SEQ[ASSIGN[x_, rhs1_],
                    ASSIGN[y_, rhs2_]] :>
             ASSIGN[y, (rhs2 /. x :> (rhs1))] /; !FreeQ[rhs2, x]
```

Code generation for Matlab . The expressions are translated in stages and then written to a file.

```
simp := {n^q :> L, n^(q-1) :> Ls, n^(t-q) :> r, n^(t-q+1) :> rs}

ToMatlab[prog_, fname_] :=
  ( fileout = fname; OpenWrite[fileout]; ToM[prog, ""];
    Close[fileout])

ToM[ASSIGN[lhs_, rhs_],tab_] :=
  ( WriteString[fileout, tab];
    ToM[lhs]; WriteString[fileout, " = "];
    ToM[rhs]; WriteString[fileout, "\n"])

ToM[LOOP[v_,idx_, stats_], tab_] :=
  (  WriteString[fileout, tab,"for ",ToString[v]," = "];
     ToMloopidx[idx]; WriteString[fileout, "\n"];
     ToM[stats,StringJoin["  ",tab]];
     WriteString[fileout, tab,"end\n"] )

ToM[PARALLEL[v_,idx_, stats_], tab_] :=
  (  WriteString[fileout, tab,"for ",ToString[v]," = "];
     ToMloopidx[idx]; WriteString[fileout, "\n"];
     ToM[stats,StringJoin["  ",tab]];
     WriteString[fileout, tab,"end\n"] )

ToM[SEQ[s1_,s2_],tab_] := ( ToM[s1,tab]; ToM[s2,tab] )
```

Assisting rules that handle special matrices, vectors and indices.

```
ToM[IDX[f_,1,l_]] :=
  WriteString[fileout, "((",InputForm[f //. simp],":",
                            InputForm[l //. simp],")+1)"]

ToM[IDX[f_,s_,l_]] :=
  WriteString[fileout, "((",InputForm[f //. simp],":",
                            InputForm[s //. simp],":",
                            InputForm[l //. simp],")+1)"]

ToMloopidx[IDX[f_,1,l_]] :=
        WriteString[fileout, InputForm[f //. simp],":",
                             InputForm[l //. simp]]

ToMloopidx[IDX[f_,s_,l_]] :=
        WriteString[fileout, InputForm[f //. simp],":",
                             InputForm[s //. simp],":",
                             InputForm[l //. simp]]

ToM[VEC[x_,idx_]] :=
  ( WriteString[fileout, ToString[x]]; ToM[idx])

ToM[TIMES[A_,B_]] :=
(
  ToM[A];
  WriteString[fileout, " * "];
  If[ToString[Head[B]]=="VEC",
     ToM[B],
     (WriteString[fileout, "("]; ToM[B]; WriteString[fileout, ")"]),
     Print["Undetermined If in times"]])

ToM[DOTSTAR[A_,B_]] :=
(
  ToM[A];
  WriteString[fileout, " .* "];
  If[ToString[Head[B]]=="VEC",
     ToM[B],
     (WriteString[fileout, "("]; ToM[B];
      WriteString[fileout, ")"]),
     Print["Undetermined If in dotstar"]])


ToM[F[n_]] :=
  WriteString[fileout, "F(",InputForm[n],")"]
ToM[P[m_,n_]] :=
  WriteString[fileout, "P(",InputForm[m],",",InputForm[n],")"]
ToM[A_[n_]] :=
  WriteString[fileout, InputForm[A],"(",InputForm[n],")"]
```

Finally two groupings of the rules and a transformation strategy that worked for all one dimensional problems.

```
expandingrules = Flatten[{product, matmatvec, AkronI, IkronA, weights,
        butterfly}]

contractingrules = Flatten[{pushweight, pushstride}]

transform[alg_] :=
    ((((((alg  //. expandingrules)
               //. pushweight)
               //. Flatten[{loopchange, loopjoin}])
               //. pushstride)
               //. strideperm)
               //. contractingrules) //. assignjoin
```

B.3 How to use the code

The purpose of this section is to link the presentation in the main chapters of the thesis to the use of the code that is made available.

B.3.1 Loop Nests

The loop transformation environment runs under Mathematica 2.2. The file is called `KronComp`. You read it in mathematica by entering `<< KronComp`.

A user needs to be familiar at the very least with the Mathematica user interface, as well as the contents of `KronComp` where the matrix grammar is defined. Familiarity with the transformation rules is also useful but not necessary.

Suppose we want to implement a new transposition algorithm. Let $x_{n_1 n_2}$ be a vector seen as an n_1 by n_2 matrix, where $n_2 = n$ and $n_1 = p * n_2$ for some integers n and p. We know from the definition of the stride permutation that the solution is the product

$$y = P_{n_1 n_2, n_1} x.$$

We will generate an implementation based on the factorization with Kronecker products

$$P_{pn^2, pn} = (I_p \otimes P_{n^2,n})(P_{pn,p} \otimes I_{n_2}).$$

The elementary program in matrix language that overwrites x with its transpose using the above factorization is

```
     ASSIGN[VEC[x,IDX[0,1,n*p*n-1]],
            TIMES[KRON[I[p],P[n^2,n]],
                  KRON[P[p*n,p],I[n]],
                  VEC[x,IDX[0,1,n*p*n-1]]]]
```

We may now attempt to apply individual transformations or use the predefined transformation strategy `transform`. The latter generates the following sequence of loops (displayed in FORTRAN90)

```
do i6 = 0,-1 + n
  v9(0:-1 + n*p) = x(i6:i6 + n*(-1 + n*p):n)
  do i8 = 0,-1 + p
    x(i6 + i8*n**2:i6 + n*(-1 + (1 + i8)*n):n) =
                v9(i8:i8 + (-1 + n)*p:p)
  enddo
enddo
do i7 = 0,-1 + p
  v11(0:-1 + n**2) = x(i7*n**2:-1 + (1 + i7)*n**2)
  do i10 = 0,-1 + n
    x(i10*n + i7*n**2:-1 + (1 + i10)*n + i7*n**2) =
                v11(i10:i10 + (-1 + n)*n:n)
  enddo
enddo
```

Note that index and temporary variables may differ since new ones are generated for successive runs.

You can export the generated loop in MATLAB , FORTRAN90 or C by issuing

```
ToMatlab[Out[n], "mytranspose.m"]
ToFortran90[Out[n], "mytranspose.f"]
ToC[Out[n], "mytranspose.c"]
```

respectively.

B.3.2 Radices

The radix generation environment runs under Maple V Release 3. Some familiarity with Maple is required, in order to change the underlying radix generation codes.

The latest version provided uses the mixed radix FFT that guarantees the lowest number of floating point operations.

The file containing the radix generator is `genFFT.m3`. For ease of use all control variables that change the behavior of the generator are listed at the end of the file. After properly setting the control variables (their semantics have been listed in section B.1), you can generate the specified radix code by executing from the shell command line

```
maple < genFFTv.m3
```

This will generate the file `FFTn.f` where `n` is substituted by the radix size.

To test the generated codes, we provide a generic FFT tester `testerfft.f` that can run in multiple architectures unchanged. The preprocessor flags that are available are

- `IBM` to generate code for IBM mainframes like the 3090.
- `RS6K` to generate code for IBM RS 6000 workstations and the SP1 and SP2 supercomputers.
- `CRAY` to generate code for Cray supercomputers.
- `SUN` to generate code for Sun SPARC workstations.
- `HP` to generate code for HP workstations.
- `NN` to set the radix size.
- `MANY` to set the maximum exponent size.

For example, in order to generate, time and test an FFT implementation of radix 4 and maximum size 4^9 for the SP2 you can execute the sequence of instructions:

```
maple < genFFTv.m3
cc -P -DMANY=9 -DNN=4 -DRS6K testerfft.f ; mv testerfft.i tester.f
xlf -O3 -qarch=pwr2 tester.f FFT4.f readrtc.s -lesslp2
./a.out
```

The FORTRAN compiler flags `-O3 -qarch=pwr2 -lesslp2` request level 3 code optimization for the POWER2 processor architecture and link with the ESSL library. The timer function `readrtc.s` is described in the next chapter.

Appendix C

Benchmark Codes

C.1 Timing Routine

Fortran does not provide a standard way to read the real time clock of the system that the program is running on. For this reason we had to resort to a specific solution on each of the systems we needed our code to run. Here is the Fortran subroutine that returns the real time clock reading for multiple systems:

```
      function second()
      real*8 second
#if RS6K
      external readrtc
      dimension itime(2)
      call readrtc(itime)
      second = itime(1) + 1.0d-9 * itime(2)
#elif CRAY
      call tsecnd(second)
#elif IBM
      real*8 s
      integer rcode
      call cputime(s, rcode)
      second = s
c     second is in microseconds
#elif HP | SUN
      real*4 timer(2), etime
      second = etime(timer)
```

```
#endif
      return
      end
```

For the IBM RS6000 workstation, the timing is performed via a combination of an assembly –kindly provided by IBM– and the Fortran routine above, that read the real time clock very accurately.

```
#  FUNCTION: Read Realtime clock
#
#  EXECUTION ENVIRONMENT:
#  Standard register usage and linkage convention.
#  Registers used r3-r8
#  Condition registers used: 0
#  No stack requirements.
#  No TOC requirements.
#
#  Temporary substitute for system times function.
#
#  The addresses are treated as usigned quantities,
#
#  RETURN VALUE DESCRIPTION: Value of RTCU to 0(3),
#  value of RTCL to 4(3).
#
#  r3=0 if ok   r3=-1 if no good read
#
#          S_PROLOG( readrtc )
#
#  Calling sequence:
#       R3   Address of 2 word block for RTC results
#
          .toc
          .csect readrtc[DS]
          .globl readrtc[DS]

          .long  .readrtc[PR]
          .long  TOC[t0]

.readrtc:
          .csect .readrtc[PR]
          .globl .readrtc[PR]

          mfspr    5,4         # RTCU
          mfspr    6,5          # RTCL
          mfspr    7,4          # RTCU
          mr       4,3          # Save address in case of retry
          cmpl     0,5,7        # Test for valid read
          lil      3,0          # Zero return code and data
          st       5,0(4)       # Return RTCU
          st       6,4(4)       # Return RTCL
          beqr                  # Return if valid read - r3 = 0
          lil      8,32         # Retry count - safety valve
          mtctr    8            # Count reg has retry count
ReRead:
          mfspr    5,4          # RTCU
          mfspr    6,5          # RTCL
          mfspr    7,4          # RTCU
          cmpl     0,5,7        # Test for valid read
          st       5,0(4)       # Return RTCU
          st       6,4(4)       # Return RTCL
          beqr                  # Return if valid read - r3 = 0
          bc       16,0,ReRead # Dec count and continue(branch) if not 0
          lil      3,-1         # Bad return code - retry count exceeded
          br                    # Return with failure code - r3 ^= 0
          .align  2
          .byte   0xdf,2,0xdf,0
```

C.2 ESSL and Parallel ESSL calls

For example the following subroutine calculates a two dimensional forward FFT of size n_1-by-n_2 on p processors using **pESSL** and **BLACS**. Variable `x` contains the source in block distrition along the second dimension. Variable `y` stores the results, again in block distrition.

```
      subroutine do_pessl_fft(x,y,n1,n2,p)
      implicit none
      complex*16 x(n1*n2/p),y(n1*n2/p)
      integer n1,n2,p

      integer icontxt,myrow,mycol
      integer*4 ip(40)
      real*8 second,time0,time_pESSL

! pESSL
      call blacs_get(0,0,icontxt)
      call blacs_gridinit(icontxt,'R',1,p)
      call blacs_gridinfo(icontxt,1,p,myrow,mycol)
      ip(1) = 1
      ip(2) = 1

      time0 = second()
      call pdcft2(x,y,n1,n2,1,1d0,icontxt,ip)
      time_pESSL = second()-time0
      print *,mycol,'Do pESSL 2D-FFT     ', time_pESSL

      call blacs_exit(1)

      return
      end
```

C.2.1 Kronecker Product Kernels

```
! Compute Y = kron(I(m1),F(p)) * P(n1,m1) * X
      subroutine IkronFtP(m1,p,X,Y,n1,m2)
      implicit none
      integer m1,p,n1,m2
      complex*16 x(n1*m2),y(n1*m2)

      integer naux1,naux2,j
      parameter (naux1=2097653, naux2=2135056)
      real*8 aux1(naux1), aux2(naux2)

      call dcft(1,x,m1,1,y,1,p,p,m1,1,1d0,aux1,naux1,aux2,naux2)
      do j=1,m2
         call dcft(0,x((j-1)*n1+1),m1,1,y((j-1)*n1+1),
     $        1,p,p,m1,1,1d0,aux1,naux1,aux2,naux2)
      enddo

      return
      end

! Compute X = kron(I(m1),F(p)) * X
      subroutine IkronF(m1,p,X,n1,m2)
      implicit none
      integer m1,p,n1,m2
      complex*16 x(n1*m2)

      integer naux1,naux2,j
      parameter (naux1=2097653, naux2=2135056)
      real*8 aux1(naux1), aux2(naux2)
      call dcft(1,x,1,p,x,1,p,p,m1*m2,1,1d0,aux1,naux1,aux2,naux2)
      call dcft(0,x,1,p,x,1,p,p,m1*m2,1,1d0,aux1,naux1,aux2,naux2)

      return
      end

! Compute X = D(m1,p) * X
      subroutine Dscale(m1,p,X,n1,m2)
      implicit none
      integer m1,p,n1,m2
      complex*16 x(n1*m2)

      integer i,j,k,l
      complex*16 ii,w0,w(p),d(n1+20),prefetch
      real*8 pi
      parameter (pi = 3.14159265358979d0)

      ii = dcmplx(0d0,1d0)
      w(1) = 1d0
      w0 = cdexp(-2d0*pi*ii/n1)
      do i=2,p
         w(i) = w(i-1) * w0
      enddo
      d(1:p) = 1d0
      do k=1,m1-1
         d(k*p+1:(k+1)*p) = w * d((k-1)*p+1:k*p)
      enddo
      do j=1,m2
         l = 1
         do k=(j-1)*n1+1,j*n1
            prefetch = d(l+20)
            x(k) = d(l) * x(k)
            l = l+1
         enddo
      enddo

      return
      end

! Compute Y = kron(F(m1),I(p)) * Y
      subroutine FkronI(m1,p,Y,n1,m2)
      implicit none
      integer m1,p,n1,m2
      complex*16 y(n1*m2)
      integer naux1,naux2, i,j,k,l
      parameter (naux1=2097653, naux2=2135056)
      real*8 aux1(naux1), aux2(naux2)

      call dcft(1,y,p,1,y,p,1,m1,p,1,1d0,aux1,naux1,aux2,naux2)
      do j=1,m2
         call dcft(0,y((j-1)*n1+1),p,1,y((j-1)*n1+1),
     $        p,1,m1,p,1,1d0,aux1,naux1,aux2,naux2)
      enddo

      return
      end
```

C.2.2 Stride Permutation as a Matrix Transpose

```
! Compute Y = P(n,p) * X
      subroutine StPerm(p,X,n,m,Y)
      implicit none
      integer p,n,m,j
      complex*16 x(n*m),y(n*m)

      do j=1,m
         call zgetmo(x((j-1)*n+1),p,p,n/p,y((j-1)*n+1),n/p)
      enddo
      return
      end
```

Bibliography

[AE93] Lars-Erik Andersson and Tommy Elfving. Two constrained Procrustes problems. LiTH-MAT-R-1993-39, Linköping University, Department of Mathematics, December 1993.

[AG92] R. C. Agarwal and F. G. Gustavson. Algorithm and architecture aspects of producing ESSL BLAS on POWER2. In *PowerPC and POWER2: Technical Aspects of the New IBM RISC System/6000*, pages 167–176. IBM Corporation, 1992.

[AGZ94] R. C. Agarwal, F. G. Gustavson, and M. Zubair. A high performance parallel algorithm for 1-d FFT. In *Supercomputing'94*, pages 34–40. IEEE Computer Society and ACM, IEEE Computer Society Press, November 1994.

[AS85] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* The MIT Press, 1985.

[BCFH92] James M. Boyle, Maurice Clint, Stephen Fitzpatrick, and Terence J. Harmer. The construction of numerical mathematical software for the AMT DAP by program transformation. In L. Bouge, M. Cosnar, Y. Robert, and D. Trystram, editors, *CONPAR'92 VAPP V*, Lecture Notes in Computer Science 634, pages 761–767. Springer-Verlag, September 1992.

[Ber80] M. W. Berry. *Multiprocessor sparse SVD algorithms and applications.* Ph.D. dissertation, The University of Illinois at Urbana-Champaign, 1980.

[Ber85] Robert H. Berman. Fourier transform algorithms for spectral analysis derived with MACSYMA. In Richard Pavelle, editor, *Applications of computer algebra*, pages 210–241. Kluwer Academic Publishers, Boston, 1985.

[BH90] William L. Briggs and Van Emden Henson. The FFT as a multigrid algorithm. *SIAM Review*, 32(2):252–261, June 1990.

[BHSO87] William L. Briggs, Leslie B. Hart, Roland A. Sweet, and Abbie O'Gallagher. Multiprocessor FFT methods. *SIAM Journal of Scientific and Statistical Computing*, 8(1):s27–s42, January 1987.

[Bra91] Bert L. Bradford. *Fast Fourier Transforms for Direct Solution of Poisson's Equation.* Ph.D. dissertation, Department of Mathematics, University of Colorado, 1991.

[Bro92] Lisa G. Brown. A survey of image registration techniques. *ACM Computing Surveys*, 24(4):325–376, December 1992.

[BS91] David H. Bailey and Paul N. Swarztrauber. The fractional Fourier transform and applications. *SIAM Review*, 33(3):389–404, September 1991.

[CGM85] P. Concus, G. H. Golub, and G. Meurant. Block preconditioning for the conjugate gradient method. *SIAM J Sci Stat Comput*, 6(1):220–252, January 1985.

[Cha88] T. F. Chan. An optimal circulant preconditioner for Toeplitz systems. *SIAM Journal on Scientific and Statistical Computing*, 9:766–771, 1988.

[CHR94] Nikos Chrisochoides, Elias Houstis, and John Rice. Mapping algorithms and software environments for data parallel PDE iterative solvers. *Journal of Parallel and Distributed Computing*, 21:75–95, 1994.

[CJ92] Raymond H. Chan and Xiao-Qing Jin. A family of block preconditioners for block systems. *SIAM Journal of Scientific and Statistical Computing*, 13(5):1218–1235, September 1992.

[Coo87] James W. Cooley. How the FFT gained acceptance. In *ACM Conference on the History of Numeric and Scientific Computing*, May 1987.

[CS88] Hung-Yuan Chung and York-Yih Sun. Analysis and parameter estimation of nonlinear systems with Hammerstein model using Taylor series approach. *IEEE Transactions on Circuits and Systems*, 35(12):1539–1541, December 1988.

[CT65] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19:297–301, April 1965.

[Dau92] Ingrid Daubechies. *Ten Lectures on Wavelets*, volume 61 of *Regional Conference Series in Applied Mathematics.* SIAM, 1992.

[DGK$^+$94] D.L. Dai, S.K.S. Gupta, S.D. Kaushik, Lu J.H., R.V. Singh, C.-H. Huang, P. Sadayappan, and R.W. Johnson. EXTENT: A portable programming environment for designing and implementing high-performance block recursive algorithms. In *Supercomputing'94*, pages 49–58. IEEE Computer Society and ACM, IEEE Computer Society Press, 1994.

[DLS94] W. De Launey and J. Seberry. The strong Kronecker product. *Journal of Combinatorial Theory, Series A*, 66(2):192–213, May 1994.

[DR93] A. Dutt and V. Rokhlin. Fast Fourier transforms for nonequispaced data. *SIAM Journal of Scientific and Statistical Computing*, 14(6):1368–1393, November 1993.

[DRGGP95] L. De Rose, K. Gallivan, E. Gallopoulos, and D Padua. A MATLAB compiler and restructurer for the development of scientific libraries and applications. Technical Report CSRD 1430, CSRD, University of Illinois at Urbana-Champaign, May 1995.

[Els89] Ann C. Elster. Fast bit-reversal algorithms. In *ICASSP'89*, pages 1099–1102, 1989.

[FCK95] S. Fitzpatrick, M. Clint, and P. Kilpatrick. The automated derivation of sparse implementations of numerical algorithms through program transformation. Technical Report 1995, Department of Computer Science, The Queen's University of Belfast, April 1995.

[FF94] Donald W. Fausett and Charles T. Fulton. Large least squares problems involving Kronecker products. *SIAM Journal on Matrix Analysis*, 15(1), January 1994.

[FHB92] Stephen Fitzpatrick, Terence J. Harmer, and James M. Boyle. Deriving efficient parallel implementations of program transformation. In L. Bouge, M. Cosnar, Y. Robert, and D. Trystram, editors, *CONPAR'92 VAPP V*, Lecture Notes in Computer Science 634, pages 761–767. Springer-Verlag, September 1992.

[Fle90] Roger Flecher. *Practical Methods of Optimization.* John Wiley & Sons, 1990.

[GCT91] J. Granata, M. Conner, and R. Tolimieri. A tensor product factorization of the linear convolution matrix. *IEEE Transactions on Circuits and Systems*, 38(11):1364–1366, November 1991.

[GCT92a] J. Granata, M. Conner, and R. Tolimieri. Recursive fast algorithms and the role of the tensor product. *IEEE Transactions on SP*, 40(12):2921–2930, December 1992.

[GCT92b] J. Granata, M. Conner, and R. Tolimieri. The tensor product: A mathematical programming language for FFT's and other fast DSP operations. *IEEE SP Magazine*, pages 40–48, December 1992.

[GJ79] Michel R. Garey and David S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[GKH$^+$94] S. Gupta, S. Kaushik, C-H Huang, J. Johnson, R. Johnson, and P. Sadayappan. A methodology for generating data distributions from tensor product formulas. Technical report, The Ohio State University, 1994.

[GKHS93] S. Gupta, S. Kaushik, C.-H. Huang, and P. Sadayappan. On compiling array expressions for efficient execution for distributed-memory machines. Technical report, The Ohio State University, 1993.

[GKS92] A. Gupta, V. Kumar, and A. Sameh. Performance and scalability of conjugate gradient methods on parallel computers. *preprint*, 1992.

[GLO81] Gene H. Golub, Franklin Luk, and Mike Overton. A block Lanczos method for computing the singular values and corresponding singular vectors of a matrix. *ACM Transactions on Mathematical Software*, 7:149–169, 1981.

[GS91] Leslie Greengard and John Strain. The fast Gauss transform. *SIAM Journal of Scientific and Statistical Computing*, 12(1):79–94, January 1991.

[GVL89] Gene H. Golub and Charles F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, 1989.

[GW87] Rafael C. Gonzalez and Paul Wintz. *Digital image processing.* Addison-Wesley, second edition, 1987.

[Han94] Per Christian Hansen. Regularization Tools, A Matlab Package for Analysis and Solution of Discrete Ill-Posed Problems. *Numerical Algorithms*, 6:1–35, 1994.

[HC89] David C. Hyland and Emmanuel G. Collins. Block Kronecker products and block norm matrices in large-scale analysis. *SIAM Journal of Matrix Analysis*, 10(1):18–29, January 1989.

[Heg95] Markus Hegland. An implementation of multiple and multivariate Fourier transforms on vector processors. *SIAM Journal on Scientific Computing*, 16(2):271–288, March 1995.

[Hig88] Nicholas J. Higham. The symmetric Procrustes problem. *BIT*, 28:133–143, 1988.

[HJB85] Michael T. Heideman, Don H. Johnson, and C. Sidney Burrus. Gauss and the history of the fast Fourier transform. *Archive for History of Exact Sciences*, 34(3):265–277, 1985.

[HJJ90] C-H Huang, J. Johnson, and R. Johnson. A tensor product formulation of Strassen's matrix multiplication algorithm. *Applied Mathematics Letters*, 3(3):67–71, 1990.

[HJJ92] C-H Huang, J. Johnson, and R. Johnson. Generating parallel programs from tensor product formulas: A case study of Strassen's matrix multiplication algorithm. Technical report, The Ohio State University, 1992.

[HL87] Chua-Huang Huang and Christian Lengauer. The derivation of systolic implementation of programs. *Acta Informatica*, 24:295–632, 1987.

[Hor86] B. K. P. Horn. *Robot Vision*. The MIT Press, 1986.

[HPS83] H. V. Henderson, F. Pukelsheim, and S. R. Searle. On the history of the Kroneker product. *Linear and Multilinear Algebra*, 14:113–120, 1983.

[HS81] H. V. Henderson and S. R. Searle. The vec-permutation matrix, the vec operator and Kronecker products, a review. *Linear and Multilinear Algebra*, 9:271–288, 1981.

[Jac95] Paul B. Jackson. *Enhancing the Nuprl Proof Development System and Applying it to Computational Abstract Algebra*. Ph.D. dissertation, Department of Computer Science, Cornell University, 1995.

[Jai89] Anil K. Jain. *Fundamentals of digital image proccessing*. Information and System Sciences. Prentice-Hall International, 1989.

[JJRT90] J. Johnson, R. Johnson, D. Rodriguez, and R. Tolimieri. A methodology for designing, modifying, and implementing Fourier transform algorithms on various architectures. *Circuits Systems Signal Process.*, 9(4):449–500, 1990.

[JKFM89] S. Lennart Johnsson, Robert L. Krawitz, Roger Frye, and Douglas MacDonald. Cooley-Tukey FFT on the connection machine. Technical Report YALEU/DCS/TR-750, Department of Computer Science, Yale University, 1989.

[Kau83] Linda Kaufman. Matrix methods for queuing problems. *SIAM Journal of Scientific and Statistical Computing*, 4(3):525–552, September 1983.

[KH75] C. D. Kuglin and D. C. Hines. The phase correlation image alignment method. In *IEEE ICCS*, pages 163–165. IEEE Computer Society, IEEE Computer Society Press, September 1975.

[KHJ$^+$93] S. D. Kaushik, C.-H. Huang, J. R. Johnson, R. W. Johnson, and P. Sadayappan. Efficient transposition algorithms for large matrices. In *Supercomputing'93*, November 1993.

[KHJS92] S. D. Kaushik, C.-H. Huang, R. W. Johnson, and P. Sadayappan. A methodology for generating efficient disk-based algorithms from tensor product formulas. Technical report, The Ohio State University, 1992.

[KL92] Deepak Kapur and Yagiti N. Lakshman. Elimination methods: an introduction. In B. R. Donald, D. Kapur, and J. L. Mundy, editors, *Symbolic and Numerical Computation for Artificial Intelligence*, Computational Mathematics and Applications, pages 45–87. Academic Press, 1992.

[KSH$^+$92a] S. Kaushik, S. Sharma, C-H Huang, J. Johnson, R. Johnson, and P. Sadayappan. An algebraic theory for modeling multistage interconnection networks. In *International Conference on Parallel and Distributed Systems (ICPDS'92)*, pages 97–106, December 1992.

[KSH$^+$92b] S. Kaushik, S. Sharma, C-H Huang, J. Johnson, R. Johnson, and P. Sadayappan. An algebraic theory for modeling direct interconnection networks. In *Supercomputing'92*, pages 488–497, November 1992.

[KSH$^+$92c] S. Kaushik, S. Sharma, C-H Huang, J. Johnson, R. Johnson, and P. Sadayappan. A methodology for generating data distributions from tensor product formulas. Technical report, The Ohio State University, 1992.

[LCCM89] P. Lie Chin Cheong and S. D. Morgera. Iterative methods for restoring noisy images. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(4):580–585, April 1989.

[LCT93] Chao Lu, James W. Cooley, and Richard Tolimieri. FFT algorithms for prime transform sizes and their implementations on VAX, IBM3090VF, and IBM RS/6000. *IEEE Transactions on Signal Processing*, 41(2):638–647, February 1993.

[LH74] Charles L. Lawson and Richard J. Hanson. *Solving least squares problems.* Prentice-Hall, 1974.

[Mal89] Stephane G. Mallat. A theory for multiresolution signal decomposition: The wavelet representation. *IEEE Trans. PAMI*, 11(7):674–693, July 1989.

[Mat90] The MathWorks, Inc. *Pro-Matlab, User's Guide*, 1990.

[MR96] David K. Maslen and Daniel N. Rockmore. Generalized FFTs - a Survey of Some Recent Results. In *To Appear*, 1996.

[Nag95a] James G. Nagy. Applications of Toeplitz systems. *SIAM News*, 28(8):10–11, 1995.

[Nag95b] James G. Nagy. Iterative techniques for the solution of Toeplitz systems. *SIAM News*, 28(7):8–9, 1995.

[Neu81] Marcel F. Neuts. *Matrix-Geometric Solutions in Stochastic Models, An Algorithmic Approach.* The John Hopkins University Press, 1981.

[Nus82] Henri J. Nussbaumer. *Fast Fourier Transform and Convolution Algorithms.* Springer-Verlag, 1982.

[Pra91] William K. Pratt. *Digital Image Processing.* John Wiley & Sons Inc., second edition, 1991.

[PS73] V. Pereyra and G. Scherer. Efficient computer manipulation of tensor products with applications to multidimentional approximation. *Mathematics of Computation*, 27(123):595–605, July 1973.

[QD88] L. Qiu and E. J. Davison. A new method for the stability robustness determination of state space models with real perturbations. In *IEEE* 27^{th} *Conference on Decision and Control*, pages 538–543, Austin, Texas, December 1988.

[Rau80] Urho A. Rauhala. Introduction to array algebra. *Photogrammetric Engineering and Remote Sensing*, 46(2):117–192, February 1980.

[RM89] Phillip A. Regalia and Sanjit K. Mitra. Kronecker products, unitary matrices and signal processing applications. *SIAM Review*, 31(4):586–613, December 1989.

[SC93] David Sharp and Martin Cripps. Synthesis of the fast Fourier transform algorithm by functional language program transformation. In *Euromicro Workshop on Parallel and Distributed Processing*, pages 136–143, January 1993.

[Ste91] Willi-Hans Steeb. *Kronecker Product of Matrices and Applications.* Wissenschaftsverlag, 1991.

[Ste95] William J. Stewart. *Introduction to the Numerical Solution of Markov Chains.* Princeton University Press, 1995.

[Str86] Gilbert Strang. A proposal for Toeplitz matrix calculations. *Stud. Appl. Math*, 74:171–176, 1986.

[Str93] Gilbert Strang. Wavelet transforms versus Fourier transforms. *Bulletin of the AMS*, 28(2):288–305, April 1993.

[Swa82] Paul N. Swarztrauber. Vectorizing the FFT's. In G. Rodrigue, editor, *Parallel Computations*, pages 51–83. Academic Press, New York, 1982.

[Thi91] Thinking Machines Corp. *Programming in Fortran and Fortran Reference Manual*, 1991.

[Thi93] Thinking Machines Corp. *CMSSL for CM Fortran: CM-5 Edition*, 1993.

[VL92] Charles F. Van Loan. *Computational frameworks for the Fast Fourier Transform.* SIAM, 1992.

[VLP92] Charles F. Van Loan and Nikos P. Pitsianis. Approximation with Kronecker products. Technical Report CTC92TR109, Cornell Theory Center, November 1992.

[War94] J. Ward. Space-time adaptive processing for airborn radar. Technical Report TR1015, Lincoln Labs, MIT, December 1994.

[Wat94] Waterloo Maple Software. *Maple V Release 4*, 1994.

[Wil88] James Hardy Wilkinson. *The Algebraic eigenvalue problem.* Oxford University Press, 1988.

[Wol88] Stephen Wolfram. *Mathematica, A system for Doing Mathematics by Computer.* Addison-Wesley, 1988.

[Zip88] Paul Zipkin. The use of phase-type distributions in inventory-control models. *Naval Research Logistics*, 35:247–257, 1988.

[Zip93] Richard E. Zippel. The Weyl computer algebra substrate. In Alfonso Miola, editor, *Design and Implementation of Symbolic Computation Systems*, volume 722 of *Lecture Notes in Computer Science*, pages 303–318. Springer Verlag, 1993.