A Package for Generating, Manipulating, and Testing Convolution Algorithms

Anthony F. Breitzman
and
Jeremy R. Johnson

Technical Report DU-CS-03-05
Department of Computer Science
Drexel University
Philadelphia, PA 19104
November 2003

A Package for Generating, Manipulating, and Testing Convolution Algorithms

Anthony F. Breitzman and Jeremy R. Johnson

November 20, 2003

Abstract

This report describes a Maple package created for exploring the techniques of Winograd, Nussbaumer, and others for computing "fast" convolution algorithms. After codifying known convolution techniques into a common framework of bilinear algorithms built from parameterized matrices and algebraic operators, Maple's symbolic and algebraic computation facilities are used to derive and manipulate these algorithms.

The package provides an infrastructure for generating, manipulating, testing, and combining various convolution convolution algorithms within an interactive environment. The algorithms generated by the package can be exported to a domain-specific language called SPL (Signal Processing Language) and then translated into efficient C or FORTRAN code by the SPL compiler. By combining the strengths of Maple and the SPL compiler the benefits of existing algebraic computation tools are obtained without the need to embed high-performance compiler technology into a computer algebra system.

The resulting enviroment allows for the systematic application of the algebraic theory developed over the years to produce correct and efficient programs. Numerous algorithmic choices can be tried, allowing for rapid testing of various optimizations to find the best combinations of algorithms for a particular size convolution on a particular computer. Furthermore, automatic code generation and algebraic verification provides the ability to construct non-trivial examples with confidence that the resulting code is correct.

1 Introduction

Convolution is arguably one of the most important computations in signal processing. It also has applications outside of signal processing including the efficient computation of prime length Fourier Transforms, polynomial multiplication, and large integer multiplication. The importance of convolution has led to much research on the design and computer implementation of fast algorithms for performing convolution.

The careful study of convolution algorithms began with S. Winograd's investigation of the complexity of convolution and related problems. Winograd proved a lower bound on the number of multiplications required for convolution, and used the Chinese Remainder Theorem to construct optimal algorithms that achieve the minimum number of multiplies [13, 14]. Unfortunately, to reach the theoretical minimum in multiplications often requires an inordinate number of additions that may defeat the gain in multiplies. These results spurred further study in the design and implementation of "fast" convolution algorithms. The research on this problem over the last 25 years is summarized in several books ([9], [3], [2], and [11]).

Despite all of the development however, many questions remain about these algorithms. The main unanswered question is to determine the practicality of Winograd based algorithms (A Winograd based algorithm in this report denotes an algorithm constructed using the techniques introduced by Winograd) over the full range of input sizes. In particular, how do the Winograd algorithms compare to fast Fourier transform (FFT) based algorithms using the convolution theorem (See [11] for discussion of the convolution theorem). More generally, of the various techniques that have been proposed, which lead to fast implementations, and what is the best way to combine the various algorithms to optimize performance. One of the reasons this has not been done is the difficulty in implementing Winograd algorithms for general sizes, and the need to produce production quality implementations in order to obtain meaningful comparisons. Moreover, in order to do a systematic search for the best algorithm it is necessary to be able to fully explore different combinations of algorithms and implementations.

The package provides an infrastructure for generating, manipulating, testing, and combining various convolution convolution algorithms within an interactive environment. The algorithms generated by the package can be exported to a domain-specific language called SPL (Signal Processing Language) and then translated into efficient C or FORTRAN code by the SPL compiler. By combining the strengths of Maple and the SPL compiler the benefits of existing algebraic computation tools are obtained without the need to embed high-performance compiler technology into a computer algebra system.

The infrastructure has two components: a core component and a convolution component. The core component contains basic matrix types and operations that can be used as a foundation for an arbitrary class of algorithms. The convolution component is built upon the core component and consists of symbols and algorithms for computing convolutions or generating computer code to compute convolutions.

This report is a discussion of the Maple convolution package; a companion paper [5] provides a survey and algebraic basis for the convolution algorithms implemented here. Full source code for the Maple package and an example worksheet can be found at www.spiral.net. Documentation for SPL can be found in [15], and the latest version of the SPL compiler can be obtained at: http://www.ece.cmu.edu/∼spiral.

This paper is organized as follows. Section 2 provides a brief survey and review of some of the key algorithms contained in the convolution package. Section 2 provides a brief introduction to SPL and the SPL compiler. The convolution package creates SPL code for convolution algorithms, and the SPL compiler creates C or FORTRAN code from SPL code. Finally, section 3 discusses the convolution package including calling conventions, and the implementation details needed in order to extend the package.

2 Overview of Convolution Algorithms

This section gives a brief overview of convolutions and a survey of some of the algorithms contained in the Maple convolution package. A more detailed survey with complete proofs and derivations can be found in [5].

2.1 Convolution from Three Perspectives

Convolution can be viewed from three different perspectives: as a sum, a polynomial product, and a matrix operation. This allows polynomial algebra to be used to derive algorithms and the corresponding matrix algebra for manipulating and implementing algorithms.

The linear convolution of the vectors $\mathbf{u} = (u_0, \ldots, u_{M-1})$ and $\mathbf{v} = (v_0, \ldots, v_{N-1})$ is a vector of size $M + N - 1$. If both vectors are of the same size, $M = N$, the linear convolution is said to be of size N.

Definition 1 (Linear Convolution)
Let $\mathbf{u} = (u_0, \ldots, u_{M-1})$ *and* $\mathbf{v} = (v_0, \ldots, v_{N-1})$. *The* i*-th component of* $\mathbf{u} * \mathbf{v}$ *is equal to*

$$(\mathbf{u} * \mathbf{v})_i = \sum_{k=0}^{N-1} u_{i-k} v_k, 0 \leq i < 2N \tag{1}$$

If the vectors $\mathbf{u} = (u_0, \ldots, u_{M-1})$ and $\mathbf{v} = (v_0, \ldots, v_{N-1})$ are mapped to the polynomials

$$u(x) = \sum_{i=0}^{M-1} u_i x^i \text{ and } v(x) = \sum_{j=0}^{N-1} v_j x^j,$$

then $\mathbf{u} * \mathbf{v}$ is mapped to the polynomial $u(x)v(x)$.

The linear convolution sum is also equivalent to the following matrix vector multiplication.

$$\mathbf{u} * \mathbf{v} = \begin{bmatrix} u_0 & & & \\ u_1 & u_0 & & \\ \vdots & u_1 & \ddots & \\ u_{M-1} & \vdots & \ddots & u_0 \\ & u_{M-1} & & u_1 \\ & & \ddots & \vdots \\ & & & u_{M-1} \end{bmatrix} \mathbf{v} \tag{2}$$

Cyclic convolution of two vectors of size N is obtained from linear convolution by reducing the indices $i - k$ and k in Equation 1 modulo N.

Definition 2 (Cyclic Convolution)
Let $\mathbf{u} = (u_0, \ldots, u_{N-1})$ *and* $\mathbf{v} = (v_0, \ldots, v_{N-1})$*. The i-th component of the cyclic convolution of* $\mathbf{u}$ *and* $\mathbf{v}$*, denoted by* $\mathbf{u} \circledast \mathbf{v}$*, is equal to*

$$(\mathbf{u} \circledast \mathbf{v})_i = \sum_{k=0}^{N-1} u_{(i-k) \bmod N} v_k, 0 \leq i < N \tag{3}$$

Cicular convolution is obtained by multiplying the polyomials corresponding to $\mathbf{u}$ and $\mathbf{v}$ and taking the remainder modulo $x^N - 1$. It can also be recast in terms of matrix algebra, as the product of a *circulant matrix* $\mathbf{Circ}_N(\mathbf{u})$, times the vector v,

$$\mathbf{u} \circledast \mathbf{v} = \begin{bmatrix} u_0 & u_{N-1} & u_{N-2} & \ldots & u_1 \\ u_1 & u_0 & u_{N-1} & \ldots & u_2 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ u_{N-2} & \ldots & u_1 & u_0 & u_{N-1} \\ u_{N-1} & u_{N-2} & \ldots & u_1 & u_0 \end{bmatrix} \mathbf{v}.$$

This matrix is called a circulant matrix because the columns of the matrix are all obtained by cyclically rotating the first column.

A circulant matrix is generated by the shift matrix

$$S_N = \begin{bmatrix} 0 & \ldots & 0 & 0 & 1 \\ 1 & 0 & \ldots & 0 & 0 \\ 0 & 1 & 0 & \ldots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \ldots & 0 & 1 & 0 \end{bmatrix}, \tag{4}$$

which is so named because when it is applied to a vector it cyclically shifts the elements. It is easy to verify that

$$\mathbf{Circ}_N(\mathbf{u}) = \sum_{i=0}^{N-1} u_i S_N^i. \tag{5}$$

It is clear that a cyclic convolution can be created from a linear convolution, just by applying an additional reduction matrix, (see example 3 for an example of a reduction matrix), but it is also possible to create a linear convolution of size N by zero padding a cyclic convolution of size $2N - 1$ In fact, in [11], Tolimieri discusses a method for creating a linear convolution via a cyclic convolution of size $2N - 2$.

2.2 Bilinear Algorithms

A bilinear algorithm is useful way to describe convolution algorithms since it provides a convenient way to operate on, combine, and manipulate convolution algorithms [13, 14].

Definition 3 (Bilinear Algorithm)
A bilinear algorithm is a triple (C, A, B) of matrices, where the column dimension of C is equal to the row dimensions of A and B. When applied to a pair of vectors $\mathbf{u}$ and $\mathbf{v}$ the bilinear algorithm (C, A, B) computes $C\,(A\mathbf{u} \bullet B\mathbf{v})$, where $\bullet$ represents componentwise multiplication of vectors.

Example 1 *Consider a two-point linear convolution*

$$\begin{bmatrix} u_0 \\ u_1 \end{bmatrix} * \begin{bmatrix} v_0 \\ v_1 \end{bmatrix} = \begin{bmatrix} u_0 v_0 \\ u_0 v_1 + u_1 v_0 \\ u_1 v_1 \end{bmatrix}.$$

This can be computed with three instead of four multiplications using the following algorithm.

1. $t_0 \leftarrow u_0 v_0$;
2. $t_2 \leftarrow u_1 v_1$;
3. $t_1 \leftarrow (u_0 + u_1)(v_0 + v_1) - t_0 - t_1$;

The desired convolution is given by the vectors whose components are t_0, t_1, and t_2. This algorithm is equivalent to the bilinear algorithm

$$tc_2 = \left(\begin{bmatrix} 1 & 0 & 0 \\ -1 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \right). \tag{6}$$

This algorithm is known as the Toom-Cook algorithm [12, 4]. This particular example can be generated by evaluating $\mathbf{uv}$ at values 0, 1, and ∞ and then interpolating. See [5] for complete details of this example as well as a complete derivation of the Toom-Cook algorithm.

The Toom-Cook algorithm is closely related to the convolution theorem, which provides a method for computing cyclic convolutions via the Fourier Transform (see [11, 5]).

In a few rare cases, the standard method of multiplying polynomials learned in high school might be the best choice for a linear convolution algorithm. This can be turned into a bilinear algorithm of matrices in the obvious way.

Example 2 *A 3×3 linear convolution given by the Standard Algorithm is :*

$$
\begin{aligned}
sb_3 &= \left(\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \right) \\
&= (sb_3[C], sb_3[A], sb_3[B])
\end{aligned}
$$

Definition 4 (Bar Notation)
Let $A(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$ we will denote by $\overline{A(x)}$ the equivalent vector $\begin{bmatrix} a_0 & a_1 & \ldots & a_n \end{bmatrix}^T$

Cyclic convolutions can be created from linear convolutions via a reduction matrix. Let $M(f(x))$ denote the reduction matrix such that $M(f(x))\overline{A(x)} = \overline{A(x) \mod f(x)}$.

Example 3 *Composing*

$$M(x^2 - 1) = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix},$$

with the Toom-Cook bilinear algorithm of (6), yields:

$$(M(x^2 - 1)C_2, A_2, B_2) = \left(\begin{bmatrix} 1 & 0 & 1 \\ -1 & 1 & -1 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix} \right),$$

which is a bilinear algorithm for 2-point cyclic convolution.

2.3 Kronecker Product and Combining of Linear Convolutions

Definition 5 (Kronecker Product) *The Kronecker product, of an $m_1 \times n_1$ matrix A and a $m_2 \times n_2$ matrix B is the $m_1m_2 \times n_1n_2$ block matrix whose (i, j) block is obtained by multiplying the (i, j) element of the matrix A by the matrix B. The Kronecker product will be denoted by $\otimes$.*

The Kronecker product can be used to combine small linear convolution algorithms into larger ones in an efficient manner. This is important, because the Kronecker product of smaller convolution algorithms will generally use fewer operations than a direct larger convolution algorithm. For example combining a Toom-Cook algorithm of size 2 with a Toom-Cook algorithm of size 3, creates a linear convolution of size 6 that uses many fewer operations than a Toom-Cook convolution of size 6.

Theorem 1 (Kronecker Product of Linear Convolutions) *Let $\mathcal{L}_m$ and $\mathcal{L}_n$ be linear convolution algorithms of size m and n respectively. Then $O_{m,n}(\mathcal{L}_m \otimes \mathcal{L}_n)$ is a linear convolution algorithm of size mn, where $O_{m,n}$ is a sparse $(2m-1)(2n-1) \times (2mn-1)$ matrix. The non-zero entries are equal to one and occur in locations $jm+i, j(2m-1)+i$ and $jm+i, (j-1)(2m-1)+m+i$ for $0 \leq j < 2n-1$ and $0 \leq i < m-1$.*

The proof and an example can be found in [5].

2.4 Operations on Bilinear Algorithms

Let $\mathcal{B}_1 = (C_1, A_1, B_1)$ and $\mathcal{B}_2 = (C_2, A_2, B_2)$ be two bilinear algorithms. The following operations are defined for bilinear algorithms.

1. **[direct sum]** $\mathcal{B}_1 \oplus \mathcal{B}_2 = (C_1 \oplus C_2, A_1 \oplus A_2, B_1 \oplus B_2)$.
2. **[Kronecker product]** $\mathcal{B}_1 \otimes \mathcal{B}_2 = (C_1 \otimes C_2, A_1 \otimes A_2, B_1 \otimes B_2)$.
3. **[product]** $\mathcal{B}_1\mathcal{B}_2 = (C_2C_1, A_1A_2, B_1B_2)$, assuming compatible row and column dimensions.

As a special case of the product of two bilinear algorithms, let P and Q be matrices and assume compatible row and column dimensions.

$$P\mathcal{B}_1Q = (PC_1, A_1Q, B_1Q).$$

An additional property that will be required is known as the commutation theorem.

Theorem 2 (Commutation Theorem)
Let A be an $m_1 \times n_1$ matrix and let B be an $m_2 \times n_2$ matrix. Then

$$L^{m_1m_2}_{m_1}(A \otimes B)L^{n_1n_2}_{n_2} = (B \otimes A)$$

Here L is a permutation matrix known as a stride permutation, (see [5]), so that the commutation theorem means that the Kronecker product is almost commutative; that is, the arguments can be commuted provided additional permutations are applied to the inputs and outputs. This is useful since it is often the case that $A \otimes B$ uses fewer operations than $B \otimes A$, so the commutation theorem can provide a reduced operation count when combining convolution algorithms. The proof of the commutation theorem can be found in [6]. See [5] for additional details.

2.5 Winograd Convolution Algorithm and Prime Power Algorithm

The polynomial version of the Chinese Remainder provides a decomposition of a polynomial algebra, $C[x]/f(x)$ into a direct product of polynomial algebras.

Theorem 3 (Chinese Remainder Theorem)
Assume that $f(x) = f_1(x) \cdots f_t(x)$ *where* $\gcd(f_i(x), f_j(x)) = 1$ *for* $i \neq j$. *Then the polynomial algebra modulo* $f(x)$ *can be decomposed into a direct product of polynomial algebras modulo the factors of* $f(x)$.

$$C[x]/f(x) \cong C[x]/f_1(x) \times \cdots \times C[x]/f_t(x)$$

Winograd's algorithm for computing cyclic convolution follows from the Chinese Remainder theorem when applied to the irreducible rational factors of the polynomial $X^N - 1$. The irreducible rational factors of $x^N - 1$ are called cyclotomic polynomials.

The cyclotomic polynomials can be defined recursively from the formula $x^N - 1 = \prod_{d|N} \Phi_d(x)$, or alternatively $\Phi_N(x) = \prod_{\gcd(j,N)=1}(x - \omega_N^j)$, where ω_N is a primitive N-th root of unity. It follows that $\deg(\Phi_N(x)) = \phi(N)$, where ϕ is the Euler ϕ function. It is well known [7] that $\Phi_N(x)$ has integer coefficients and is irreducible over the rationals.

Stated simply, since $x^N - 1 = \prod_{d|N} \Phi_d(x)$, Winograd's algorithm computes an N-point cyclic convolution by applying the CRT to convolutions modulo $\phi_d(x)$ with $d|N$. For example since $x^4 - 1 = (x^2 + 1)(x - 1)(x + 1)$, a 4-point cyclic convolution can be created from a 2-point and two 1-point cyclic convolutions using the following theorem.

Theorem 4 (Winograd Convolution Algorithm)
Let $\mathcal{C}_f$ *denote a bilinear algorithm that multiplies elements of* $C[x]/f(x)$. *Then*

$$\mathcal{C} = R^{-1} \left(\bigoplus_{d|n} \mathcal{C}_{\Phi_d(x)} \right) R \tag{7}$$

where $R = [R_{d_1} \; R_{d_2} \; \ldots R_{d_k}]^T$ *and* $R_{d_i} f = f(x) \bmod \Phi_{d_i}(x)$ *is a bilinear algorithm for* N*-point cyclic convolution.*

Using the 2-point cyclic convolution algorithm in Example 3 and the cyclotomic polynomials $\Phi_1(x) = (x-1)$, $\Phi_2(x) = (x+1)$, and $\Phi_4(x) = (x^2+1)$ the following 4-point cyclic convolution algorithm is obtained.

Example 4

$$\left(R_4^{-1} \begin{bmatrix} 1 & & & & & \\ & 1 & & & & \\ & & & 1 & 0 & -1 \\ & & & -1 & 1 & -1 \end{bmatrix}, \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & 0 \\ & & 1 & 1 \\ & & 0 & 1 \end{bmatrix} R_4, \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & 0 \\ & & 1 & 1 \\ & & 0 & 1 \end{bmatrix} R_4 \right),$$

where R_4 is easily verified to be

$$R = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}.$$

A complete derivation with proofs of the Winograd algorithm can be found in [5].

When $N = p^k$ (p prime), the Winograd algorithm has additional structure. This structure, discussed in [10] follows from the following two properties of cyclotomic polynomials: $\Phi_p(x) = x^{p-1} + \cdots + x + 1$ and $\Phi_{p^k}(x) = \Phi_p(x^{p^{k-1}})$. This additional structure provides an efficient way to compute R_{p^k}.

Theorem 5 *Let R_{p^k} be the $p^k \times p^k$ projection matrix modulo $\{\Phi_{p^i}(x),\ i = 0, \ldots, k\}$. Then*

$$R_{p^k} = \begin{bmatrix} 1_p \otimes R_{p^{k-1}} \\ G_p \otimes I_{p^{k-1}} \end{bmatrix},$$

where G_n is the $(n-1) \times n$ matrix:

$$G_n = \begin{bmatrix} 1 & & & -1 \\ & 1 & & -1 \\ & & \ddots & -1 \\ & & 1 & -1 \end{bmatrix},$$

and 1_n is the $1 \times n$ matrix filled with $1's$. Moreover, $R_{p^k} = (R_{p^{k-1}} \oplus I_{(p-1)p^{k-1}})(R_p \otimes I_{p^{k-1}})$.

A complete proof as well as a derivation of the inverse of R_{p^K} can be found in [5].

Example 5 *A bilinear algorithm for a cyclic convolution of size 27 is (C, A, B), where:*

$$C = R_{3^3}^{-1} \begin{bmatrix} 1 & & & \\ & M(x^2+x+1)\mathcal{L}_2[C] & & \\ & & M(x^6+x^3+1)\mathcal{L}_6[C] & \\ & & & M(x^{18}+x^9+1)\mathcal{L}_{18}[C] \end{bmatrix}$$

$$A = \begin{bmatrix} 1 & & & \\ & \mathcal{L}_2[A] & & \\ & & \mathcal{L}_6[A] & \\ & & & \mathcal{L}_{18}[A] \end{bmatrix} R_{3^3} \text{ and } B = \begin{bmatrix} 1 & & & \\ & \mathcal{L}_2[B] & & \\ & & \mathcal{L}_6[B] & \\ & & & \mathcal{L}_{18}[B] \end{bmatrix} R_{3^3}.$$

where $\mathcal{L}_n$ is a bilinear algorithm for a linear convolution of size n of any method, and

$$R_{3^3} = \begin{bmatrix} 1 & 1 & 1 & \\ 1 & 0 & -1 & \\ 0 & 1 & -1 & \\ & & & I_{24} \end{bmatrix} \begin{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & -1 \\ 0 & 1 & -1 \end{bmatrix} \otimes I_3 & \\ & I_{18} \end{bmatrix} \begin{bmatrix} \begin{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & -1 \\ 0 & 1 & -1 \end{bmatrix} \otimes I_9 \end{bmatrix} \end{bmatrix}$$

2.6 Matrix Exchange

In filtering applications it is often the case that one of the inputs to be cyclically convolved is fixed. Fixing one input in a bilinear algorithm leads to a linear algorithm. When this is the case, one part of the bilinear algorithm can be precomputed and the precomputation does not count towards the cost of the algorithm. Let (C, A, B) be a cyclic convolution and assume that the first input vector x is fixed. Then the computation $(C, A, B)(x, y)$ is equal to $(C\,\mathrm{diag}(Ax)B)y$, where $\mathrm{diag}(Ax)$ is the diagonal matrix whose diagonal elements are equal to the vector Ax. Thus the resulting linear algorithm is given by the matrix factorization $C\,\mathrm{diag}(Ax)B$.

In most cases the C portion of the bilinear algorithm is much more costly than the A portion of the algorithm, so it would be desirable if this part could be precomputed. Given a bilinear algorithm for a cyclic convolution, the matrix exchange property allows the exchange of the C and A matrices.

2.7 The Agarwal-Cooley and Split-Nesting Algorithms

The Agarwal-Cooley algorithm [1] uses the Kronecker product to create a larger cyclic convolution from smaller cyclic convolutions. The Split-Nesting algorithm, due to Nussbaumer [9], follows directly from Agarwal-Cooley using simple properties of the tensor product.

Theorem 6 (Agarwal-Cooley Algorithm)
Assume $\gcd(m, n) = 1$ *and let* $\mathcal{C}_m = (C_m, A_m, B_m)$ *and* $\mathcal{C}_n = (C_n, A_n, B_n)$ *be bilinear algorithms for cyclic convolution of size* m *and* n*. Let* $Q_{m,n}^{-1}$ *be the permutation that maps* i *to* $(i \mod m)n+(i \mod n)$*. It follows that* $Q_{m,n}^{-1}(\mathcal{C}_m \otimes \mathcal{C}_n)Q_{m,n}$ *computes a cyclic convolution of size* mn*.*

Let $R_m^{-1}\left(\bigoplus_{i=0}^{k_1}\mathcal{C}_{m_i}\right)R_m$ and $R_n^{-1}\left(\bigoplus_{i=0}^{k_2}\mathcal{C}_{n_i}\right)R_n$ be bilinear algorithms to compute m, and n-point Winograd cyclic convolutions. Then combining Agarwal-Cooley with the Winograd algorithm yields the bilinear algorithm

$$Q_{m,n}^{-1}\left(R_m^{-1}\left(\bigoplus_{i=0}^{k_1}\mathcal{C}_{m_i}\right)R_m\right)\otimes\left(R_n^{-1}\left(\bigoplus_{j=0}^{k_2}\mathcal{C}_{n_j}\right)R_n\right)Q_{m,n} \tag{8}$$

for computing an mn-point cyclic convolution, (provided $gcd(m, n) = 1$).

Rearranging this equation into a double sum of tensor products leads to the "Split-Nesting Algorithm" which was first derived by Nussbaumer in [9], who observed that it requires fewer additions then (8). The following theorem describes this transformation.

Theorem 7 (Split Nesting) *Let* $\mathcal{C} = \bigoplus_{i=0}^{s-1} \mathcal{C}_i$ *and* $\mathcal{D} = \bigoplus_{j=0}^{t-1} \mathcal{D}_j$. *Then*

$$\mathcal{C} \otimes \mathcal{D} = P^{-1} \left(\bigoplus_{i=0}^{s-1} \bigoplus_{j=0}^{t-1} \mathcal{C}_i \otimes \mathcal{D}_j \right) P,$$

where P *is a permutation.*

See [5] for a complete derivation and proof.

Example 6 *Let*

$$\mathcal{C}_4 = R_4^{-1}(1 \oplus 1 \oplus \mathcal{C}_2)R_4$$
and
$$\mathcal{C}_{27} = R_{27}^{-1}(1 \oplus \mathcal{D}_2 \oplus \mathcal{D}_6 \oplus \mathcal{D}_{18})R_{27},$$

where $\mathcal{C}_2 = M(x^2+1)\mathcal{L}_2$, $\mathcal{D}_2 = M(x^2+x+1)\mathcal{L}_2$, $\mathcal{D}_6 = M(x^6+x^3+1)\mathcal{L}_6$, $\mathcal{D}_{18} = M(x^{18}+x^9+1)\mathcal{L}_{18}$, *be the algorithms for cyclic convolution on 4 and 27 points given in Examples 4 and 5. By Agarwal-Cooley,*

$$Q_{4,27}^{-1}(R_4^{-1}(1 \oplus 1 \oplus \mathcal{C}_2)R_4) \otimes (R_{27}^{-1}(1 \oplus \mathcal{D}_2 \oplus \mathcal{D}_6 \oplus \mathcal{D}_{18})R_{27})Q_{4,27}$$

is an algorithm for cyclic convolution on 108 points. The split nesting theorem transforms this algorithm into

$$(Q_{4,27}^{-1}(R_4^{-1} \otimes R_{27}^{-1})P^{-1}$$
$$(1 \oplus \mathcal{D}_2 \oplus \mathcal{D}_6 \oplus \mathcal{D}_{18}) \oplus (1 \oplus \mathcal{D}_2 \oplus \mathcal{D}_6 \oplus \mathcal{D}_{18}) \oplus (\mathcal{C}_2 \oplus \mathcal{C}_2 \otimes \mathcal{D}_2 \oplus \mathcal{C}_2 \otimes \mathcal{D}_6 \oplus \mathcal{C}_2 \otimes \mathcal{D}_{18}))$$
$$P(R_4 \otimes R_{27})Q_{4,27}$$

where $P = I_{27} \oplus I_{27} \oplus P_3$ *and* $P_3 = (I_2 \oplus L_2^4 \oplus L_2^{12} \oplus L_2^{36})L_{27}^{54}$.

Note that in the example above choices made for the cyclic convolutions of size 4 and 27 required forced the use of certain linear algorithms in the computation of the size 108 cyclic convolution. The improved split-nesting algorithm discussed in [5] overcomes this restriction and allows substitution of the best linear algorithms at each stage of the computation.

In this section many convolution algorithms were introduced briefly, in order to make this report self-contained. However, since this report is primarily about the Maple implementation, this section is deliberately brief. The net effect is not only a terse description of most of the basic algorithms, but also a complete omission of most proofs. A complete discussion of each of these methods, with proofs and numerous examples, can be found in [5].

3 Maple Infrastructure

In this section, a Maple package for implementing the algorithms discussed in previous sections is described. The implementation is based on a programming language called SPL and a Maple infrastructure that aids in the creation and manipulation of SPL code. The latest version of the SPL Compiler can be obtained at http://polaris.cs.uiuc.edu/ jxiong/spl/ and the latest version of the Maple package can be obtained at http://www.spiral.net.

3.1 Overview of SPL and the SPL Maple Package

Having codified known convolution techniques into a common framework of bilinear algorithms built from parameterized matrices and algebraic operators, Maple's symbolic and algebraic computation facilities are used to derive and manipulate these algorithms. The infrastructure provided by the package allows for the generation, manipulation, testing, and combining of various convolution algorithms within an interactive environment. The algorithms generated by the package can be exported to a domain-specific language called SPL (Signal Processing Language) and then translated into efficient C or FORTRAN code by the SPL compiler. By combining the strengths of Maple and the SPL compiler the benefits of existing algebraic computation tools are realized without the need to embed high-performance compiler technology in a computer algebra system.

The resulting environment allows one to systematically apply the algebraic theory developed over the years to produce correct and efficient programs. Numerous algorithmic choices can be tried, allowing the user to rapidly test various optimizations to find the best combination of algorithms for a particular size convolution on a particular computer. Furthermore, automatic code generation and algebraic verification provides the ability to construct non-trivial examples with confidence that the resulting code is correct.

3.1.1 SPL Language

This section briefly outlines the SPL language. Further details are available in [15], where, in addition, an explanation of how SPL programs are translated to programs is provided.

SPL provides a convenient way of expressing matrix factorizations, and the SPL compiler translates matrix factorizations into efficient programs for applying the matrix expression to an input vector. SPL programs consist of the following: 1) SPL formulas, which are symbolic expressions used to represent matrix factorizations, 2) constant expressions for entries appearing in formulas; 3) "define statements" for assigning names to subexpressions; and 4) compiler directives, which are used to change the behavior of the SPL compiler in some way (e.g. to turn loop unrolling on or off). SPL formulas are built from general matrix constructions, parameterized symbols denoting families of special matrices, and matrix operations such as matrix composition, direct sum, and the tensor product. The elements of a matrix

can be real or complex numbers. In SPL, these numbers can be specified as scalar constant expressions, which may contain function invocations and symbolic constants like `pi`. For example, `12`, `1.23`, `5*pi`, `sqrt(5)`, and `(cos(2*pi/3.0),sin(2*pi/3))` are valid scalar SPL expressions. All constant scalar expressions are evaluated at compile-time. SPL uses a prefix notation similar to lisp to represent formulas. SPL code is compiled into C or FORTRAN by invoking the SPL compiler and entering the lisp like SPL formulas interactively, or by sending a text file of the SPL formulas to the SPL compiler.

General matrix constructions. Examples include the following.

- `(matrix (`a_{11} `...` a_{1n}`)...(`a_{m1} `...` a_{mn}`))` - the $m \times n$ matrix $[a_{ij}]_{0 \le i < m,\ 0 \le j < n}$.
- `(sparse (`i_1 j_1 $a_{i_1 j_1}$`) ... (`i_t j_t $a_{i_t j_t}$`))` - the $m \times n$ matrix where $m = \max(i_1, \ldots, i_t)$, $n = \max(j_1, \ldots, j_t)$ and the non-zero entries are $a_{i_k j_k}$ for $k = 1, \ldots, t$.
- `(diagonal (`a_1 `...` a_n`))` - the $n \times n$ diagonal matrix $\mathrm{diag}(a_1, \ldots, a_n)$.
- `(permutation (`σ_1 `...` σ_n`))` - the $n \times n$ permutation matrix: $i \mapsto \sigma_i$, for $i = 1, \ldots, n$.

Parameterized Symbols. Examples include the following.

- `(I n)` - the $n \times n$ identity matrix I_n.
- `(F n)` - the $n \times n$ DFT matrix F_n.
- `(L mn n)` - the $mn \times mn$ stride permutation matrix L^{mn}_n.
- `(T mn n)` - the $mn \times mn$ twiddle matrix T^{mn}_n.

Matrix operations. Examples include the following.

- `(compose` A_1 `...` A_t`)` - the matrix product $A_1 \cdots A_t$.
- `(direct-sum` A_1 `...` A_t`)` - the direct sum $A_1 \oplus \cdots \oplus A_t$.
- `(tensor` A_1 `...` A_t`)` - the tensor product $A_1 \otimes \cdots \otimes A_t$.
- `(conjugate A P)` - the matrix conjugation $A^P = P^{-1} \cdot A \cdot P$, where P is a permutation.

It is possible to define new general matrix constructions, parameterized symbols, and matrix operations using a template mechanism. To illustrate how this is done, an example showing how to add the `stack` operator to SPL is given. Let A and B be $m \times n$ and $p \times n$ matrices respectively, then `(stack A B)` is the $(m + p) \times n$ matrix

$$\begin{bmatrix} A \\ B \end{bmatrix}.$$

Given a program to apply A to a vector and a program to apply B to a vector, a program to apply `(stack A B)` to a vector is obtained by applying A to the input and storing the result in the first m elements of the output and applying B to the input and storing the result in the remaining p elements of the output. The following SPL template enables the SPL compiler to construct code for `(stack A B)` using this approach. (This example is given to illustrate that a process exists for adding new functionality to SPL; the reader need not be concerned with the specific syntax.)

```
(template (stack any any)
[$p1.nx == $p2.nx]
(
  $y(0:1:$p1.ny_1) = call $p1( $x(0:1:$p1.nx_1) )
  $y($p1.ny:1:$p0.ny_1) = call $p2( $x(0:1:$p1.nx_1) )
))
```

The first part of the template is the pattern `(stack any any)` which matches `(stack A B)` where `A` and `B` match any SPL formulas. The code for `A` and `B` is accessed through the `call` statements, where `A` is referenced as `$p1` and `B` is referenced as `$p2`. The field `nx` refers to the input dimension and `ny` refers to the output dimension (`_1` subtracts one).

3.1.2 SPL Maple Package

Programming directly in SPL is a cumbersome process, creating a need to provide an interactive version of SPL in Maple (or some other interactive scripting environment). In this environment, it is much easier to add new features and to extend the language, and it is possible to write simple scripts, using Maple's algebraic computation engine to generate SPL code [8]. In particular, SPL was extended to include bilinear computations in addition to linear computations. Also all of the parameterized matrices and bilinear algorithms discussed in Section 2 were added, and Maple's polynomial algebra capabilities were exploited to generate SPL objects obtained from the Chinese remainder theorem.

This implementation centers around the concept of an SPL object, which corresponds to a multi-linear computation. SPL objects have a name, a type, (the default type is complex), and fields that indicate the number of inputs as well as the input and output dimensions. In addition, there may be a list of parameters which may be set to Maple expressions such as an integer, list, or polynomial or other SPL objects. Since parameters include both primitive data types and SPL objects, an SPL object can be used to represent general matrix constructions, parameterized matrices, or operators. There are methods to construct an SPL object, evaluate an SPL object to a matrix or a triple of matrices in the case of bilinear algorithms, apply an SPL object, count the number of arithmetic operations used by an SPL object, and export an SPL object. Once exported an SPL object can be compiled by the SPL compiler.

SPL objects can be bound or unbound. An unbound object is a named, parameterized, multi-linear map which does not have a specified method of computation (i.e. it does not have an apply method). Alternatively, an SPL object may have an apply method, but be considered unbound because one or more of its parameters are unbound. Unbound objects can be bound by using a provided bind function. The bind function allows SPL objects to be defined in terms of other SPL objects. Parameterized matrices and operators may be defined using other parameterized matrices and operators. Since the SPL objects defining an SPL object may themselves be unbound, bind may need to be applied recursively. It is possible to specify the number of levels that bind is to be applied.

Using unbound symbols has several advantages: 1) the size of an SPL expression can be significantly shorter when symbols are not always expanded, 2) it is easier to see the structure in a complicated formula if sub-formulas are named, 3) parameterized matrices and operators not available to the SPL compiler can be used provided a bind function is available that defines them using formulas supported by the compiler, and 4) an SPL expression can be constructed whose components are unspecified and therefore, alternative computation methods can be used when applying an SPL object. The last point can be used to apply the optimization techniques presented in Section 3.3.2.

3.2 Implementation Details of Core SPL Package

The implementation centers around the concept of an 'SPL object', which corresponds to a linear or bilinear computation. (It is helpful to think of an SPL object as a matrix or triple of matrices.) An SPL object can be unbound, which essentially means the object is a placeholder or name for another (generally more complicated) SPL object.

3.2.1 Core SPL Commands

Since SPL is a language for creating algorithms for fast linear computations, the package must necessarily contain the capability to apply SPL objects to vectors. The package also contains many other capabilities for manipulating SPL objects. These capabilities, called SPL commands, are shown here:

1. `SPLEval(T:SPL object)` Evaluates an SPL object into a corresponding matrix.
2. `SPLApply(vecList:list,T:SPL object)` Applies the SPL object to a list of one or more vectors to obtain an output vector.
3. `SPLCountOps(T:SPL object,[adds,muls,assignments])` Counts the number of operations and assignments required to apply an SPL object to a vector. If `adds, muls,` and `assignments` are included, the counts are returned in these variables, otherwise the counts are printed to the screen.
4. `SPLBind(T:SPL object,[bindLevel:posint]` Binds an unbound SPL object to an actual SPL object. If `bindLevel` is ∞ or omitted, it completely binds the object, otherwise it binds the object `bindLevel` steps.
5. `SPLPrint(T:SPL object,[bindLevel:posint],[fd:file descriptor])` Shows the SPL object as a sequence of SPL commands. The default behavior is to print the sequence to the screen, unless a file descriptor is provided as the third argument. A bind-level can be passed in as an optional second argument. If a bind level is provided, the command is equivalent to `SPLPrint(SPLBind(T,bindLevel))`.

3.2.2 Core SPL Objects

The five commands act on SPL objects representing either linear or bilinear algorithms, so a mechanism is needed to create SPL objects. In SPL, there are two fundamental types of SPL objects: parameterized matrices and operators. For example, `(I n)` is a parameterized matrix that takes a size n and returns an Identity matrix of the specified size, while (compose ...) is an operator that acts on two or more SPL objects. Within the Core package there are parameterized matrices and operators as well, but the package is designed so that both types of objects are represented with the same data structure. This allows for a very simple implementation of the five basic commands discussed above.

The list of all available parameterized matrices and operators provided within the Core Maple package can be found in tables 1 and 2.

Table 1: **Core Maple parameterized Matrices**

Bound Symbols	
`SPLDiagonal(d1,...,dn)`	Diagonal matrix of entries $d_1, \ldots, d_n$
`SPLFourier(n,omega)`	$n \times n$ Fourier Matrix with optional argument omega
`SPLIdentity(r,c)`	$r \times c$ (c optional) identity matrix
`SPLAntiIdentity(n)`	$n \times n$ Anti-identity matrix
`SPLMatrix([[a11,a12,...],` `[a21,a22,...],...])`	Entry specified matrix
`SPLPermutation([p1,...,pn])`	Permutation matrix
`SPLPermutationInv([p1,...,pn])`	Inverse or reverse permutation Matrix
`SPLSparse([[i,j,aij],...])`	Sparse matrix with non-zero entries specified
`SPLStride(mn,n)`	Stride permutation
`SPLTwiddle(n,[omega])`	Twiddle matrix
`SPLZeroMatrix(r,c)`	$r \times c$ zero matrix
Unbound Symbols	
`SPLOnes(m,n)`	$m \times n$ matrix with all entries equal to 1.

`SPLBilinear` is an unusual operator that warrants further explanation. This operator operates on a set of 3 SPL objects to represent a bilinear algorithm. Note that all other operators (`SPLCompose, SPLTensor,` etc.) can operate on bilinear algorithms as well as linear algorithms (matrices), and that if the input is a linear algorithm then the output will be linear algorithms, and similarly if the input is a bilinear algorithm, then the output will be bilinear. Again, a bilinear algorithm is simply a triple of linear algorithms, and the only way to convert a bilinear algorithm to a linear algorithm is to apply a single vector to one of the two inputs.

Implementation of Bound Symbols and Operators in the Core Package

The package was built so that new symbols and operators could be added with relative ease,

Table 2: **Core Maple Operators**

Bound Operators	
`SPLAdd(m1,...,mn)`	Add SPL ojects
`SPLCompose(m1,...,mn)`	Compose SPL objects
`SPLConjugate(m,Pinv,P)`	Equivalent to SPLCompose(Pinv,m,P)
`SPLDirectSum(m1,...,mn)`	Direct sum of SPL objects
`SPLInverse(m)`	Inverse of SPL object
`SPLTensor(m1,...,mn)`	Tensor product of SPL objects
`SPLTranspose(m)`	Transpose of SPL object
Unbound Operators	
`SPLBilinear(c,a,b)`	Create a bilinear object from 3 SPL objects
`SPLAugmentMatrix(m1,...,mn)`	Augment matrices
`SPLStackMatrix(m1,...,mn)`	Stack matrices
`SPLConvertPerm(P)`	Converts an SPL expression P, that represents a sequence of operations on permutations,into a single permutation. (e.g. SPLTensor(p1, p2) can be converted to a single permutation)
`SPLTensorI(m1,...,mn)`	$(m_1 \otimes I)(I \otimes m_2 \otimes I) \ldots (I \otimes m_n)$
`SPLCommuteTensorI(m1,m2)`	Commutation theorem for linear and bilinear algorithms

and so that new symbols would not require changes to existing symbols and operators. In order to gain that flexibility, all new objects must provide a constructor that returns an SPL object, and each object must supply its own Apply, Eval, Print, and Count functions. In this way, nothing is assumed about SPL objects by the package, and thus there are no built-in limitations.

As mentioned above, in the SPL language, there are two types of objects: symbols and operators. However, removing this distinction simplifies the implementation and provides a more uniform view; in later sections it will be shown that the implementation of convolution algorithms is equivalent to writing down SPL objects as one might write down the formulas for each convolution algorithm found in Section 2. It is therefore useful to think generically of SPL objects rather than symbols and operators. in the Maple package all SPL objects, whether operators or symbols are treated the same and implemented in the same manner.

Every SPL object must have the following fields defined within its constructor:

1. `apply` - This is a pointer to an apply function for the object. The apply function allows the user to apply a an object to a vector.

2. `bind` - This is a pointer to a bind function. For bound objects this is set to NULL, for unbound objects this points to a function that will rewrite the unbound object into a sequence of bound objects.

3. `countOps` - This is a pointer to a function that counts the number of operations required to apply the object to a vector.

4. `eval` - This is a pointer to an evaluation function, that will display the object as a matrix, or triple of matrices if the object is bilinear.

5. `inv` - This is a pointer of a function that will create the objects inverse when it exists. (Note that inverse and transpose are not implemented in the SPL compiler.)

6. `print` - This is a pointer to a function that prints the object into SPL code.

7. `vecInputs` - This specifies whether the object accepts 1 vector (linear), 2 vectors, (bilinear) etc. Although only linear and bilinear objects are currently supported, there is no reason why the package couldn't be slightly modified to support trilinear and multilinear objects.

8. `trans` - This is a pointer to a function that can create an object's transpose.

9. `name` - A name for the object (e.g. compose, matrix, sparse, etc.). This is the name printed by the print function, and corresponds to a name recognized by the SPL compiler in the case of bound objects.

10. `rowdim` - Row dimension of the object, or row dimension of the C component of a bilinear algorithm.

11. `coldim` - Column dimension of the object, or column dimension of the A and B components of a bilinear algorithm.

12. `parameters` - Parameters of the object. For example for Identity the parameters will be the row and column dimension, while for matrix the parameters will be a list of rows.

13. `numParameters` - The number of parameters. This is useful for cases such as compose, where there can be a variable number of input parameters.

14. `bound` - Is the object bound? As discussed below, the existence of a bind function is not sufficient to determine whether an object is bound. (This parameter and the next parameter could be avoided by using an `isBound` function, but instead these items are stored to improve speed.)

15. `parametersBound` - Are an objects parameters bound? For example an object might be a composition of several unbound SPL objects.

In short, an SPL object is simply a Maple table (a Maple table is basically an associative array, see [8]) with these 15 fields defined as well as the corresponding eval, apply, and other functions provided. It is useful to conceptually think of an SPL object as a table with these 15 fields, however to save space the first eight fields in the list above (which are static) are stored in a separate associative array indexed by the object's name, so that only the next

seven fields actually have to be stored in the object's table. This implementation can lead to substantial savings in the size of the table used to store an SPL object. For example, for large convolution algorithms it is not uncommon to have more than 100 identity matrices involved in the computation. By storing the eight static fields, more than 800 objects are removed from the SPL object table. (Actually the savings would be larger, because the same system is used for all symbols and operators).

To make the discussion clearer, consider the implementation of `SPLIdentity`. The complete code for `SPLIdentity` is shown below. The entries to the global table `symTabl` are assigned when the package is loaded. Whenever an identity object is needed, the user calls the constructor `SPLIdentity` with the desired dimensions as parameters. Since the constructor does little more than check for errors and fill in the seven variable fields discussed above, the constructor for identity is very similar to the constructors for most other SPL objects. All of the functionality of the SPL objects are defined in the functions pointed to within the symbol table as is shown here.

```
#########################################################################
# Identity Matrix
# Inputs:
#   n : positive integer.
# Notation:    (I m n) or (I n)
# Properties:  permutation, symmetric.
# Definition   (I n) i -> i.
#########################################################################

symTabl["I"][apply] := applyIdentity;
symTabl["I"][bind] := NULL;
symTabl["I"][countOps] := countOpsIdentity;
symTabl["I"][eval] := evalIdentity;
symTabl["I"][inv] := invIdentity;
symTabl["I"][print] := defaultSymPrint;
symTabl["I"][vecInputs] := 1;
symTabl["I"][trans] := transIdentity;
```

```
# Constructor for symbol.  Creates an SPL object.
# Inputs: Symbol parameters m=rowdim, args[2]=n=coldim (optional).
SPLIdentity := proc(m::posint) local t,n;
  if nargs > 1 then
    n := args[2];
    if (not type(n,posint)) then
      ERROR("arguments to SPLIdentity should be positive integers");
    fi;
  else
    n := args[1];
  fi;
  t := table();  t[name] := "I";
  t[rowdim] := eval(m);  t[coldim] := eval(n);
  t[parameters] := [eval(m),eval(n)]; t[numParameters] := 2;
  t[bound]:= true; t[parametersBound] := true;
  RETURN(eval(t));
end;

#transpose of an identity
transIdentity := proc(U) local t;
  t := SPLIdentity(U[parameters][2],U[parameters][1]);
  RETURN(eval(t));
end;

#operation count for identity; note no adds or multiplications for
#identity, only assignments
countOpsIdentity:= proc(t,adds::evaln,muls::evaln,assigs::evaln);
  assigs := eval(assigs) + t[parameters][1];
end;

#inverse of identity = identity
invIdentity := proc(U);
  if (U[parameters][1]<>U[parameters][2]) then
    ERROR("Inverse doesn't exist - invIdentity")
  else
    RETURN(eval(U));
  fi;
end;

# Evaluate as a matrix
evalIdentity := proc(U) local m,n,i;
  m := U[parameters][1];  n := U[parameters][2];
  RETURN(linalg[matrix](m,n,(i,j) -> if (i=j) then 1 else 0 fi));
end;
```

```
# Apply Identity to an input vector.
applyIdentity := proc(veclist,T) local x,y,i;
  x := veclist[1];
  if (linalg[vectdim](x) <> T[coldim]) then
    ERROR("Incompatible dimensions");
  fi;
  y := linalg[vector](T[rowdim],0);
  for i from 1 to min(T[rowdim],T[coldim]) do
    y[i] := x[i];
  od;
  RETURN(eval(y));
end;

#################################################################
# End Identity
#################################################################
```

Some SPL objects take other SPL objects as parameters (e.g. `SPLCompose`, `SPLDirectSum`, `SPLTensor` and others). Since these parameters may not be bound, the `parametersBound` field is not automatically set to true as in the identity example above, but instead is set with the function `areParametersBound` which checks each parameter to see if it is bound. If a typically bound object (such as `SPLCompose`) contains unbound parameters, it is considered unbound and its `bound` field is set to false. If on the other hand the parameters are bound, the `bound` field is set to true for a bound operator. It is therefore not possible to tell whether an object is bound by checking for a bind function. See section 3.1.2 for a discussion of the advantages of unbound objects. An example of a typical bind function is provided by the example for `SPLCompose` shown here.

```
#bindCompose - do a one step bind of a compose object.
#  input: SPLCompose object U = SPLCompose(p1,p2,...,pk)
# output: t = SPLCompose(Bind1(p1),Bind1(p2),...,Bind1(pk)).
bindCompose := proc(U) local t;
  if (U[bound] = false) then
    t := SPLCompose(op(map(SPLBind1,eval(U)[parameters])));
    RETURN(eval(t));
  else
    RETURN(eval(U));
  fi;
end;

SPLBind1 := proc(U) local t;
  t := SPLBind(U,1);
  RETURN(eval(t));
end;
```

Note that since SPLCompose is a bound object in general, if its bound field is false that must mean that one or more of the parameters are unbound. The bind function therefore binds each of the parameters one level.

SPL objects that allow SPL objects as parameters must also handle the apply, eval, print, and countOps functions differently than in the identity example above. These functions must be called recursively on the parameters, rather than directly on the object itself. While the eval function for a parameterized matrix such as SPLIdentity simply created an identity matrix, the eval function for an operator must create matrices for each of its parameters and then operate on them. For example, the eval function for SPLCompose evaluates all of its parameters and then multiplies them together as in the code example below. The code for apply, print, and countOps operates on the parameters in a similar manner. The code for SPLDirectSum, SPLTensor, and other bound bound objects that can have unbound parameters is analogous.

```
# Evaluate composition of SPL objects as a matrix
# Inputs: SPLObject U with U[parameters]=A1,A2,...,An
# output: matrix representing A1A2...An
evalCompose := proc(U)
 local i;
 if U[numParameters] = 1 then
   RETURN(eval(SPLEval(U[parameters][1])));
 else
   RETURN(linalg[multiply](seq(SPLEval(U[parameters][i]),
                                        i=1..U[numParameters])));
 fi;
end:
```

Implementation of the Five SPL Commands

Since the user provides eval, apply, bind, print, and countOps functions for each object created, implementing the 5 basic commands consists of little more than calling the provided functions on the SPL object of interest. For example evaluation of an SPL object U can be accomplished with the line RETURN(symTabl[U[name]][eval](U)); Here, U's name is used to look up its eval function within the symbol table and then that function is called with U as a parameter. This works of course, because functions are first class objects in Maple.

In actuality, the code for SPLEval is slightly more complicated, because unbound objects cannot be immediately evaluated. The complete code for SPLEval is shown below.

```
#SPLEval - Evaluate a SPL Object
#  input: an SPL object U representing a sequence of SPL commands
# output: a matrix, or in the case of a bilinear object a list of
#         3 matrices.
```

```
SPLEval := proc(U) global symTabl;
  local i;
  if (U[bound]=false) then
     RETURN(SPLEval(SPLBind(U)));
  fi;
  RETURN(symTabl[U[name]][eval](U));
end;
```

SPLBind is only slightly more complicated. This is because it must handle multiple bind levels and check for errors.

```
#SPLBind - Bind an unbound object or algorithm to an actual SPL object
 SPLBind := proc(T) global symTabl;
  local i,X,bindLevel;

  if (T[bound] = true) then
    RETURN(eval(T));
  fi;
  if (T[bound] = false) then
    if (nargs > 1) then
      bindLevel := args[2];
    else
      bindLevel := infinity;
    fi;

    X := eval(symTabl[T[name]][bind])(T);  #bind T one level
    if (bindLevel > 1) then
      RETURN(SPLBind(eval(X),bindLevel-1));
    else
      RETURN(eval(X));
    fi;

  else
    ERROR("you are trying to bind a non-spl object");
  fi;

 end;
```

Two Special Case Operators: SPLTranspose and SPLInverse

Since transpose and inverse can not be implemented via any supported SPL symbol or operator, (without adding new templates to the SPL compiler), they must be defined when defining a symbol or operator. Thus when s is a symbol, `SPLTranspose(s)` returns, whatever was defined for the transpose of the symbol when it was defined. The same is true for

`SPLInverse`. When s is an operator or sequence of operators, `SPLTranspose(s)` is implemented as a rewrite rule that uses the definition of the transpose of the operator to rewrite the expression in terms of transposes of symbols, and then applies the transpose to the symbols as before. For example, let `a, b` be two arbitrary symbols, with `aT, bT` their respective transposes defined at the time of `a` and `b`. Then:

```
SPLTranspose(SPLCompose(a,b))
                    =SPLCompose(SPLTranspose(b),SPLTranspose(a))
                    =SPLCompose(bT,aT).
```

Thus, the `trans` function for compose simply reverses the parameters and calls each parameter's transpose function. Other operators work in a similar way, depending upon the particular rewrite rule. `SPLInverse` is implemented in a similar manner.

Implementation of Unbound Symbols and Operators

Unbound operators and symbols offer a powerful way to both extend the language and to simplify and add clarity to algorithms. An unbound object can be used whenever a symbol or operator that is not contained in the SPL compiler is needed. To illustrate how bind can be used to define an operator using existing operators and parameterized matrices, consider the `stack` operator discussed in Section 3.1.1. In this case the operator will be extended to take an arbitrary number of operands. Let $A_i,\ i = 1, \ldots, t$ be an $m_i \times n$ matrix, and observe that

$$(\texttt{stack}\ A_1\ \ldots\ A_t) = \begin{bmatrix} A_1 \\ \vdots \\ A_t \end{bmatrix} = \begin{bmatrix} A_1 & & \\ & \ddots & \\ & & A_t \end{bmatrix} (e_t^T \otimes I_n), \tag{9}$$

where e_t is the $1 \times t$ matrix containing all ones.

In the case of an unbound operator or symbol, only 8 of the 15 fields are required for bound symbols and operators as discussed above. The required fields for an unbound symbol or operator are `bind`, `name`, `parameters`, `numParameters`, `parametersBound`, `bound`, `rowdim`, and `coldim`.

The constructor `SPLStackMatrix` is typical of a constructor for an unbound operator; it creates a Maple table to store the object, fills in the dynamic fields, and does some error checking. The bind function, `bindStack`, uses the function `stackk` to construct the SPL formula shown in (9). It uses the parameterized matrix `(SPLOnes m n)`, which corresponds to the $m \times n$ matrix whose elements are all equal to 1. This symbol can be defined using `SPLMatrix([seq([seq(1,j=1..n)],i=1..m)])`. The complete code for the stack operator is shown below.

```
#####################################################################
# stack matrix - similar to Maple's stackMatrix operator.
# Inputs:     A1, A2, ... , At = args[1], args[2],...,args[nargs]
# Notation:   (stackMatrix A1 ... At)
#####################################################################

symTabl["stack"][bind] := bindStack;
symTabl["stack"][print] := defaultOpPrint;

SPLStackMatrix := proc() global symTabl;
 local T,i,l;
   l := [seq(args[i],i=1..nargs)];  T := table();
   T[name] := "stack";  T[numParameters] := nops(l);
   T[parameters] := [seq(eval(l[i]),i=1..nops(l))];
   T[parametersBound] := areParametersBound(T);
   T[bound] := false;
   if (T[parametersBound]) then
     for i from 1 to nargs do
       if (symTabl[args[i][name]][vecInputs]<>1) then
        ERROR("SPLStackMatrix only works on linear objects");
       fi;
     od;
   fi;
   RETURN(eval(T));
 end;

bindStack := proc(X) local T;
   T := Table();
   if (X[parametersBound] = true) then
     T := stackk(seq(X[parameters][i],i=1..X[numParameters]));
   else
     T := SPLStackMatrix(op(map(SPLBind1,eval(X)[parameters])));
   fi;
   RETURN(eval(T));
 end;

 stackk := proc() local t,n,T;
    n := args[1][coldim];  t := nargs;
    T := SPLCompose(SPLDirectSum(seq(args[i],i=1..t)),
                    SPLTensor(SPLOnes(t,1),SPLIdentity(n)));
    RETURN(eval(T));
  end;
```

An unbound object can be created for any symbol or operator in an analogous manner. For

example, virtually all of the objects in the convolution package discussed in Section 2 are implemented as unbound objects.

3.2.3 Creating Packages that use the SPL Core Package

As previously mentioned, the SPL Core package provides a superset of SPL functionality. The net result is that any application that uses matrix operations extensively can be approached by building a package that takes advantage of the Core package. For example Section 3.3 discusses a convolution package that builds upon the Core package, and illustrates the Core package's utility and flexibility. Numerous convolution algorithms are implemented based entirely on matrix operations available via the Core package.

In short, the Core package was developed to be an interactive, flexible front-end for the SPL language and compiler. Its use in this case is to provide a foundation for a convolution library, but it was built with enough generality that it can be used as the foundation for a wavelet package, a Fourier Transform package, or any other application whose algorithms arise from structured matrix operations.

3.3 Implementation of Convolution Package

The convolution package contains implementations of all of the convolution algorithms discussed in Section 2. The core package contains all of the building blocks needed to create parameterized matrices that are used by our convolution algorithms, so that creating convolution algorithms becomes equivalent to writing down formulas presented in Section 2. A list of parameterized matrices or symbols that are used to create convolution algorithms can be found in Table 3.

Implementing these symbols is considerably easier than implementing objects within the core package. These matrices have all been defined by formulas derived in Section 2 so that creating them within this package just becomes equivalent to writing down formulas. For example, in Section 2.5 it was shown that the symbol R_{p^k} used in the prime power convolution algorithm could be defined recursively as

$$R_{p^k} = (R_{p^{k-1}} \oplus I_{(p-1)p^{k-1}})(R_p \otimes I_{p^{k-1}}), \tag{10}$$

where,

$$R_p = \begin{bmatrix} 1_p \\ G_p \end{bmatrix},$$

G_n is the $(n-1) \times n$ matrix

$$G_n = \begin{bmatrix} 1 & & & -1 \\ & 1 & & -1 \\ & & \ddots & -1 \\ & & & 1 \quad -1 \end{bmatrix},$$

Table 3: **Linear Objects Contained in the Convolution Package**

`AgCooleyP(m1,...,mn)`	Permutation matrix used by the Agarwal-Cooley method
`AgCooleyPinv(m1,...,mn)`	Permutation matrix used by the Agarwal-Cooley method
`circulant(v1,...,vn)`	$n \times n$ Circulant matrix acting on vector v.
`CRT(vlen,l,indet)`	This is the Chinese Remainder Theorem; equivalent to `SPLStack(M(vLen,l[i],indet))`.
`Gn(p)`	G matrix used in Prime Power algorithm
`M(vLen,g,indet)`	M is such that $M\overline{A} = \overline{A \mod g}$ where $length(\overline{A})$ = vLen.
`overlap(m1,...,mn)`	Overlap matrix used in combining linear convolutions
`R(poly,indet)`	Reduction matrix used by reduceBilin
`Rader(p,conv)`	Returns a prime size Fourier Transform via a p-1 point cyclic convolution
`RaderQ(p,r)`	Rader permutation
`RaderQt(p,r)`	Transpose of Rader permutation
`Rpk(p,k)`	Reduction matrix used in Prime Power algorithm
`RpkInv(p,k)`	Inverse of Rpk
`Sn(n)`	$n \times n$ shift matrix
`V(m,n,[points])`	$m \times n$ Toom-Cook evaluation matrix
`Vinv(m,n,[points])`	Inverse of V

and 1_n is the $1 \times n$ matrix filled with $1's$.

The code for R_{p^K} shown below follows directly from these formulas.

```
#####################################################################
# Gn:This matrix is used to Generate Rpk
# inputs: n::posint
# output: n-1 x n matrix consisting of n-1 x n-1 identity matrix
#         augmented with a column of -1's
#####################################################################
symTabl["Gn"][print] := defaultSymPrint;
symTabl["Gn"][bind] := bindGn;

Gn := proc(n)
  local T;
  T := table();
  T[name] := "Gn";
  T[rowdim] := n-1;   T[coldim] := n;
  T[parameters] := n;
  T[bound] := false;
  RETURN(eval(T));
```

```
end;

 bindGn := proc(s) local t,i,n,m;
   n := s[parameters];
   m := SPLAugmentMatrix(SPLIdentity(n-1),
                         SPLMatrix([seq([-1],i=1..n-1)]));
   RETURN(eval(m));
 end;

#####################################################################
# Rpk:This matrix is used to Generate Rpk used by the prime power
#     algorithm.
# Inputs: p,k:posints
# output: p^k by p^k R matrix - see JSC paper for details.
#####################################################################
symTabl["Rpk"][print] := defaultSymPrint;
symTabl["Rpk"][bind] := bindRpk;

Rpk := proc(p,k)
  local T;
  T := table();
  T[name] := "Rpk";
  T[rowdim] := p^k;   T[coldim] := p^k;
  T[parameters] := [p,k];
  T[bound] := false;
  RETURN(eval(T));
end;

 bindRpk := proc(s) local t,p,k;
   p := s[parameters][1]; k := s[parameters][2];
   if (k=1) then
     t := SPLStackMatrix(oneN(p),Gn(p));
   else
     t := SPLCompose(SPLDirectSum(Rpk(p,k-1),
                     SPLIdentity(p^k - p^(k-1))),
                     SPLTensor(Rpk(p,1),
                     SPLIdentity(p^(k-1))));
   fi;
   RETURN(eval(t));
 end;
```

R_{p^k} also provides a good example of using bind levels to see the structure of the code, and to determine that an algorithm is working as it should.

Example 7 *Consider* R_2^3 *within the convolution package.*

```
> R2_3 := Rpk(2,3);

  R2_3 := table([
        coldim = 8
        parameters = [2, 3]
        bound = false
        name = "Rpk"
        rowdim = 8
        ])

> SPLPrint(R2_3);
(Rpk   2  3 )
```

If `R2_3` *is bound one level, it looks like the following, (as expected from (10)).*

```
> SPLPrint(R2_3,1);
(compose
    (direct_sum
        (Rpk   2  2 )
        (I   4  4 ))
    (tensor
        (Rpk   2  1 )
        (I   4  4 )))
```

Binding one more level shows that the second `Rpk` *in the SPL code above can be defined directly in terms of* `Gn`*.*

```
> SPLPrint(R2_3,2);
(compose
    (direct_sum
        (compose
            (direct_sum
                (Rpk   2  1 )
                (I   2  2 ))
            (tensor
                (Rpk   2  1 )
                (I   2  2 )))
        (I   4  4 ))
    (tensor
        (stack (1n   2 )(Gn   2 ))
        (I   4  4 )))
```

Note that even after binding three levels, there are still a number of SPL objects that cannot be used by the SPL compiler (without additional templates) such as `Rpk`, `stack`, `1n`, `Gn`. *To fully bind the code, use* ∞ *as the bind level in the print command as follows.*

```
> SPLPrint(R2_3,infinity);
(compose
    (direct_sum
        (compose
            (direct_sum
                (compose
                    (direct_sum
                        (matrix ( 1 1) )
                        (compose
                            (tensor (matrix ( 1 1) )(I   1  1 ))
                            (direct_sum (I   1  1 )(matrix ( (-1)) ))))
                    (tensor (matrix ( 1)( 1) )(I   2  2 )))
                (I   2  2 ))
            (tensor
                (compose
                    (direct_sum
                        (matrix ( 1 1) )
                        (compose
                            (tensor (matrix ( 1 1) )(I   1  1 ))
                            (direct_sum (I   1  1 )(matrix ( (-1)) ))))
                    (tensor (matrix ( 1)( 1) )(I   2  2 )))
                (I   2  2 )))
        (I   4  4 ))
    (tensor
        (compose
            (direct_sum
                (matrix ( 1 1) )
                (compose
                    (tensor (matrix ( 1 1) )(I   1  1 ))
                    (direct_sum (I   1  1 )(matrix ( (-1)) ))))
            (tensor (matrix ( 1)( 1) )(I   2  2 )))
        (I   4  4 )))
```

The previous example illustrates the power of unbound objects and bind levels. Since SPL code is like a mathematical assembly language, if SPL objects were always fully bound, even a small convolution algorithm would consist of hundreds of lines of incomprehensible code. Unbound objects and binding levels, allow the code to be easily understood and allows for easy debugging of new objects.

Many of the linear objects shown in Table 3 are used to create the bilinear objects shown in Table 4.

Table 4: **Bilinear SPL Objects Contained in the Convolution Package**

`AgCooley(c1,...,cn)`	Combines cyclic convolutions via the Agarwal-Cooley method.
`combineLin(l1,...,ln)`	Combines linear convolutions via tensor product
`convThm(n)`	Returns cyclic convolution of size n via the convolution theorem.
`linearConv(n)`	Creates a placeholder for a size n linear convolution. When binding, this placeholder will first look for an entry in the hash-table; if none exists it will build a linear convolution of the requested size.
`primePowerAlg(p,k)`	Returns a p^k-point cyclic convolution via the prime power algorithm.
`redimBilinLin(n, colDim,newColDim)`	Modifies a bilinear algorithm to accept a smaller `input size`.
`reduceBilin(bilin, poly,indet)`	Reduces a bilinear algorithm modulo a polynomial.
`splitNest(n)`	Returns a size n cyclic convolution via the split-nesting method. Note that the improved split-nesting algorithm uses the same procedure, but with a modified hash table.
`standardBilinCyc(n)`	Returns an n-point cyclic convolution via the standard method.
`standardBilinLin(n)`	Returns an n-point linear convolution via the standard method.
`TolimLin(n,M,indet)`	Returns an n-point linear convolution via the Tolimieri method.
`ToomCookLin(n)`	Returns an n-point linear convolution via the Toom-Cook method.
`ToomCookCyc(n)`	Returns an n-point cyclic convolution via the Toom-Cook method.
`WinogCRT(n,pList, bList,indet)`	Returns an n-point cyclic convolution via Winograd's Chinese Remainder Theorem method
`WinogHash(n)`	Returns an n-point cyclic convolution via Winograd's method using linear algorithms from the hash table.
`WinogStandard(n)`	Returns an n-point cyclic convolution via Winograd's method using Standard linear algorithms.
`WinogToomCook(n)`	Returns an n-point cyclic convolution via Winograd's method using Toom-Cook linear algorithms.

Again, because of the way in which the infrastructure was built, creating these algorithms consists mainly of writing down the formulas derived in Section 2 and in [5]. For example, recall that the prime power algorithm described by [10] was presented in Section 2.5 as

$$\mathcal{C}_{x^{p^k}-1} = R_{p^k}^{-1} \left(\bigoplus_{i=0}^{k} \mathcal{C}_{\Phi_{p^i}(x)} \right) R_{p^k}$$

where $\mathcal{C}_f$ represents the convolution $\overline{\mathcal{C}[x] \mod f(x)}$, and $\Phi_d(x)$ are cyclotomic factors of $x^{p^k} - 1$.

The Maple code for generating the prime power algorithm follows directly from this formula as follows:

```
symTabl["primePowerAlg"][print] := defaultSymPrint;
symTabl["primePowerAlg"][bind] := bindPrimePowerAlg;

#PrimePowerAlg - computes a p^k-point cyclic convolution via the
#                Selesnick-Burris prime power algorithm.
#inputs:   p,k - integers
#output:   bilinear algorithm for p^k-point cyclic convolution.
primePowerAlg := proc(p::posint,k::posint) local t;
   t := table();
   t[name] := "primePowerAlg";
   t[parameters] := [p,k];
   t[bound] := false;
   t[rowdim] := p^k;
   t[coldim] := p^k;
   RETURN(eval(t));
 end;


primePowerDirectSum := proc(p::posint,k::posint) local t,cyc,l,i,n;
    l := [linearConv([1])];
    for i from 1 to k do
      cyc := numtheory[cyclotomic](p^i,'x');
      n := degree(cyc);
      l := [op(l),SPLCompose(M(2*n-1,cyc,'x'),linearConv([n]))];
    od;
    t := SPLDirectSum(op(l));
    RETURN(eval(t));
 end;


 bindPrimePowerAlg := proc(s) local T,n;
```

Table 5: **Utility Routines Contained in the Convolution Package**

`cycConv(v1,v2)`	Computes a cyclic convolution of two vectors. (Used for testing against other cyclic convolution algorithms)
`fixedVecCount(b)`	Operation counts for a bilinear algorithm assuming 1 vector fixed
`linConv(v1,v2)`	Computes a linear convolution of two vectors. (Used for testing against other linear convolution algorithms)
`putLinConv( sizeList,bilin)`	Stores a convolution algorithm of sizeList size into the hash table.
`printLinHash()`	Prints out the contents of the hash-table.
`resetLinHash()`	Empties out the hash table of stored linear convolution algorithms.
`splitNestNeeds( bilin)`	Shows a list of sizes for linear convolutions that will be used for the split nesting method

```
  RETURN(eval(SPLCompose(RpkInv(op(s[parameters])),
              primePowerDirectSum(op(s[parameters])),
              Rpk(op(s[parameters])))));
end;
```

Note the prime power algorithm is just the composition of a direct-sum of linear convolutions modulo cyclotomic polynomials conjugated with the symbol R_{p^k} presented above.

In addition to the bilinear and linear objects built into the convolution package, there are also a number of utility routines used for testing convolution algorithms, for counting operations, and for manipulating the hash table of linear convolutions that will be discussed in the next section. Table 5 shows the utilities available within the convolution package.

3.3.1 The Linear Convolution Hash Table

Most cyclic convolutions are built from smaller linear convolutions; thus in order to reduce operations or to create the fastest cyclic convolutions, a way of storing the linear convolutions that use the fewest operations, or are the fastest in terms of run-time is needed. In this section a hash table for storing and manipulating linear convolutions of various sizes is discussed. Since many large convolution algorithms use multiple combinations of the same smaller linear convolution algorithms, storing base algorithms in a hash table leads to an efficient implementation. In Maple, the hash table is just a simple global variable that is initialized upon loading of the convolution package. The following 3 Maple commands are executed to initialize the hash table when the package is loaded.

```
>linHash[1] := ToomCookLin(1):
>linHash[2] := ToomCookLin(2):
>linHash[3] := ToomCookLin(3):
```

The hash table is also used to store tensor products of linear convolutions since those are used in the split-nesting and improved split-nesting algorithms. The public routine `putLinConv` is used to store convolution algorithms within the hash table without directly manipulating the global variable `linHash`. For example to store a 4-point linear convolution consisting of a two Toom-Cook linear convolutions of size two, the Maple command `putLinConv([4],combineLin(ToomCookLin(2), ToomCookLin(2)))` is used. To store the tensor product of two Toom-Cook linear convolutions of size two, the command `putLinConv([2,2],SPLTensorI(ToomCookLin(2),ToomCookLin(2)))` is used. Note that `linHash[2,2]` is not the same as `linHash[4]` because the latter is actually equivalent to `SPLCompose(overlap(2,2),linHash[2,2])`.

The counterpart to `putLinConv` is `getLinConv`, which gives a linear convolution of a certain size without accessing the hash table directly. Actually `getLinConv` does more than access the hash table. For instance, if a size is requested that is not in the hash table, `getLinConv` will create a placeholder for an algorithm of the requested size. If at bind time, there is no entry in the hash table for that size, the bind function will create an algorithm.

Any size linear convolution can be created by combining convolutions and then reducing dimensions, provided there is a hash table entry for a size 2 linear convolution. This can easily be proved by induction: assume it is true for all linear convolutions up to size N, to show that it is true for a size $N+1$ convolution. If $N+1$ is prime than $N+2 = 2k$ where $k < N$, so that there exist convolutions of size 2 and k that can be created via size 2 convolutions. By combining a size 2 and k convolution, a size $N+2$ linear convolution is created that can be used as an $N+1$ convolution by reduction. If $N+1$ is not prime, then $N+1 = mn$, (since $m < N$ and $n < N$, each of these can be created via size 2 convolutions), so that an $N+1$ point linear convolution by combining m and n point linear convolutions.

The full implementation of `getLinConv` and `putLinConv` that uses these ideas is shown below:

```
#####################################################################
# Linear Convolution Hash Table subroutines and setup.
#####################################################################
symTabl["linearConv"][print] := defaultSymPrint;
symTabl["linearConv"][bind] := bindGetLinConv;

getLinConv := proc(sizeList)
   local T,n,i,l,rdim;
   T := table();
   T[name] := "linearConv";
   l := [];
```

```
   for i from 1 to nops(sizeList) do
     if (sizeList[i] > 1) then
       l := [op(l),sizeList[i]];
     fi;
   od;
   if (nops(l) = 0) then
     l := [1];
   fi;
   T[parameters] := eval(l); T[bound] := false;
   n := sizeList[1]; rdim := 2*n-1;
   for i from 2 to nops(sizeList) do
     n := n*sizeList[i];
     rdim := rdim*(2*sizeList[i]-1);
   od;
   T[coldim] := n; T[rowdim] := rdim;
   RETURN(eval(T));
 end;

bindGetLinConv := proc(T)
  global linHash;
  local n,A,B,C,row,t,i,j,k,sizeList,x;
  sizeList := T[parameters];
  if (linHash[op(sizeList)][bound] = true
      or linHash[op(sizeList)][bound] = false ) then
     t := linHash[op(sizeList)];
     RETURN(eval(t));
  fi;
  if (nops(sizeList) > 1) then
      linHash[op(sizeList)]:=SPLTensorI(seq(
                                          getLinConv([sizeList[i]])
                                          ,i=1..nops(sizeList)));
     else
       n := sizeList[1];
       if (isprime(n)) then
        t := redimBilinLin(getLinConv([n+1]),n+1,n):
        linHash[n] := eval(t);
        RETURN(eval(t));
       else
        readlib(ifactors):
        x := ifactors(n)[2];
        x := x[nops(x)][1];
        t := combineLin(getLinConv([x]),getLinConv([n/x]));
        linHash[n] := eval(t);
        RETURN(eval(t));
       fi;
```

```
    fi;
end;

putLinConv := proc(sizeList,lin) global linHash;
  linHash[op(sizeList)] := eval(lin);
end;
```

A call to `printLinHash()` shows the contents of the hash table, while `resetLinHash()` resets the hash table to empty out all of the entries. Note that the latter call empties the hash table but then fills the size 2 entry with a Toom-Cook algorithm to ensure that a size 2 entry always exists. If something other than a Toom-Cook algorithm is desired, the default size 2 entry can be overridden by using `putLinConv`.

3.3.2 A Comprehensive Example

This last section shows how loading the linear convolution hash table in a specific way leads to a convolution that minimizes operations. This is an illustration of the improved split-nesting algorithm, which is just the split-nesting algorithm described in section 2.7, but where all linear convolutions are replaced with those that minimize operations. In Example 6 it was shown that the split-nesting algorithm for size $108 = 4 \times 27$ was derived as follows:

Let

$$\mathcal{C}_4 = R_4^{-1}(1 \oplus 1 \oplus \mathcal{C}_2)R_4$$
and
$$\mathcal{C}_{27} = R_{27}^{-1}(1 \oplus \mathcal{D}_2 \oplus \mathcal{D}_6 \oplus \mathcal{D}_{18})R_{27},$$

where $\mathcal{C}_2 = M(x^2+1)\mathcal{L}_2$, $\mathcal{D}_2 = M(x^2+x+1)\mathcal{L}_2$, $\mathcal{D}_6 = M(x^6+x^3+1)\mathcal{L}_6$, $\mathcal{D}_{18} = M(x^{18}+x^9+1)\mathcal{L}_{18}$, be the algorithms for cyclic convolution on 4 and 27 points given in Examples 4 and 5. By Agarwal-Cooley,

$$Q_{4,27}^{-1}(R_4^{-1}(1 \oplus 1 \oplus \mathcal{C}_2)R_4) \otimes (R_{27}^{-1}(1 \oplus \mathcal{D}_2 \oplus \mathcal{D}_6 \oplus \mathcal{D}_{18})R_{27})Q_{4,27}$$

is an algorithm for cyclic convolution on 108 points. The split nesting theorem transforms this algorithm into

$$\begin{aligned}
&(Q_{4,27}^{-1}(R_4^{-1} \otimes R_{27}^{-1})P^{-1} \\
&(1 \oplus \mathcal{D}_2 \oplus \mathcal{D}_6 \oplus \mathcal{D}_{18}) \oplus (1 \oplus \mathcal{D}_2 \oplus \mathcal{D}_6 \oplus \mathcal{D}_{18}) \oplus (\mathcal{C}_2 \oplus \mathcal{C}_2 \otimes \mathcal{D}_2 \oplus \mathcal{C}_2 \otimes \mathcal{D}_6 \oplus \mathcal{C}_2 \otimes \mathcal{D}_{18})) \\
&P(R_4 \otimes R_{27})Q_{4,27}
\end{aligned} \tag{11}$$

where $P = I_{27} \oplus I_{27} \oplus P_3$ and $P_3 = (I_2 \oplus L_2^4 \oplus L_2^{12} \oplus L_2^{36})L_{27}^{54}$.

Calling `splitNest` in the convolution package, gives an algorithm equivalent to equation 11. The Maple session for creating a size 108 convolution follows:

```
> n := 4*27;
```

```
                              n := 108

> s := splitNest(n):
> SPLPrint(s);
(compose
    (AgCooleyP   4  27 )
    (tensorI (RpkInv   2  2 )(RpkInv   3  3 ))
    (compose
        (convert2perm (direct_sum (L  27  1 )(L  27  1 )(L  54  2 )))
        (direct_sum
            (linearConv   1 )
            (compose (M   3  x^2+x+1  x )(linearConv  2 ))
            (compose (M   11  x^6+x^3+1  x )(linearConv  6 ))
            (compose (M   35  x^18+x^9+1  x )(linearConv  18 ))
            (compose (M   1  x+1  x )(linearConv  1 ))
            (compose
                (tensor (M  3  x^2+x+1  x )(M  1  x+1  x ))
                (linearConv   2 ))
            (compose
                (tensor (M  11  x^6+x^3+1  x )(M  1  x+1  x ))
                (linearConv  6 ))
            (compose
                (tensor (M  35  x^18+x^9+1  x )(M  1  x+1  x ))
                (linearConv  18 ))
            (compose (M  3  x^2+1  x )(linearConv  2 ))
            (compose
                (tensor (M  3  x^2+x+1  x )(M  3  x^2+1  x ))
                (linearConv   2  2 ))
            (compose
                (tensor (M  11  x^6+x^3+1  x )(M  3  x^2+1  x ))
                (linearConv  6  2 ))
            (compose
                (tensor (M   35  x^18+x^9+1  x )(M   3  x^2+1  x ))
                (linearConv   18  2 )))
        (convert2perm (direct_sum (L 27 27)(L 27 27)(L 54 27))))
    (tensorI (Rpk   2  2 )(Rpk   3  3 ))
    (inverse (AgCooleyP   4  27 ))).
```

If the convolution will be used in a filtering application with the matrix exchange property applied, the following operation count is obtained using the default hash table:

```
> fixedVecCount(s);
B  Adds: 1407, B  Muls: 0, B  Assigs: 3866
At Adds: 1769, At Muls: 0, At Assigs: 4611
haddamards: 470, Total Ops: 3646.
```

The idea of the improved split-nesting algorithm is that all instances of linear convolutions and tensor products of linear convolutions are replaced with linear convolutions that minimize the number of operations. To see which hash table entries are used by the algorithm, use:

```
> splitNestNeeds(s);
(linearConv   1 )
(linearConv   2 )
(linearConv   6 )
(linearConv   18 )
(linearConv   1 )
(linearConv   2 )
(linearConv   6 )
(linearConv   18 )
(linearConv   2 )
(linearConv   2  2 )
(linearConv   6  2 )
(linearConv   18  2 ).
```

It can be shown that for a size 6 linear convolution, using a standard algorithm of size 3 requires fewer operations than a Toom-Cook of size 3, while for size 18, using both a standard algorithm of size 3 and a Toom-Cook of size 3 is better than using either alone. These points as well as using a size 36 linear convolution as the basis for [18,2] are discussed fully in [5]. Below are the hash entries used to minimize the operation count.

```
> tc2:=ToomCookLin(2): tc3:=ToomCookLin(3): sb3:=standardBilinLin(3):
> putLinConv([2],tc2):
> putLinConv([6],combineLin(sb3,tc2)):
> putLinConv([18],combineLin(sb3,tc2,tc3)):
> putLinConv([2,2],SPLTensorI(tc2,tc2)):
> putLinConv([6,2],SPLTensorI(combineLin(sb3,tc2),tc2)):
> putLinConv([18,2],SPLCompose(SPLTensor(overlap(3,2,3),
                                         SPLIdentity(3)),
                               SPLTensorI(sb3,tc2,
                                          SPLCommuteTensorI(tc3,tc2)))):
```

All of the entries are straightforward except for the size [18,2]. In this case an algorithm that is equivalent to a tensor product of a size 18 linear convolution and size 2 linear convolution is needed. However this cannot be obtained directly to minimize the number

of operations. Instead, the best size 36 convolution, which is equivalent to [12,3] must be converted to a size [18,2]. To do this, note that the A component of size [12,3] is $sb3[A] \otimes tc2[A] \otimes tc2[A] \otimes tc3[A]$. By using `SPLCommuteTensor`, the A component $sb3[A] \otimes tc2[A] \otimes (L^{15}_5(tc2[A] \otimes tc3[A])L^6_2)$ is obtained, which has operation counts the same as [12,3] but is equivalent to $sb3[A] \otimes tc2[A] \otimes tc3[A] \otimes tc2[A]$ and thus usable within [18,2]. Note the same permutation is done on the B component of the bilinear algorithm. Next the left composition of `SPLTensor(overlap(3,2,3),SPLIdentity(3))` occurs because the overlap matrix from the size 18 convolution is factored out.

This yields the minimum number of operations and as shown below, saves 750 operations over the split-nesting algorithm. This example is discussed fully in [5].

```
> fixedVecCount(s);
B  Adds: 672, B  Muls: 0, B  Assigs: 3209
At Adds: 1394, At Muls: 0, At Assigs: 4497
haddamards: 830, Total Ops: 2896.
```

4 Conclusion

An infrastructure for interactively generating, testing, and manipulating convolution algorithms is described. This infrastructure is not only useful for interactively working with convolution algorithms, but also provides a mechanism for generating a domain specific language called SPL, that can ultimately transform the algorithms into efficient C or FORTRAN code. The infrastructure was built to be a general problem solving tool, with a convolution package built upon this tool. As a result, other libraries for solving matrix based problems can be built upon the core package with relative ease. This report provided an overview of SPL, a brief survey of key convolution algorithms, a discussion of the use of the infrastructure, and a discussion of how the infrastructure can be extended by adding new symbols, operators, and algorithms.

References

[1] R. C. Agarwal and J. W. Cooley. New algorithms for digital convolution. *IEEE Trans. Acoust. Speech and Signal Proc.*, 25:392–410, 1977.

[2] Richard E. Blahut. *Algebraic Methods for Signal Processing and Communications Coding.* Springer-Verlag, New York, 1992.

[3] C.S. Burrus and T.W. Parks. *DFT/FFT and Convolution Algorithms: Theory and Implementation.* John Wiley and Sons, New York, NY, USA, 1985.

[4] S. A. Cook. *Thesis:On the minimum computation time of functions.* Harvard Thesis, 1966.

[5] J. R. Johnson and A. F. Breitzman. Automatic derivation and implementation of fast convolution algorithms. *Journal of Symbolic Computation*, In Press.

[6] J. R. Johnson, R. W. 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.

[7] Serge Lang. *Algebra.* Addison-Wesley, Redwood City, CA, 1984.

[8] M. B. Monagan, editor. *Maple V Programming Guide (Version A): Release 5.* Springer-Verlag, New York, 1998.

[9] H. J. Nussbaumer. *Fast Fourier Transform and Convolutional Algorithms.* Springer-Verlag, New York, 2nd. edition, 1982.

[10] I. Selesnick and C. Burrus. Automatic generation of prime length fft programs. *IEEE Transactions on Signal Processing*, 44:14–24, 1996.

[11] R. Tolimieri, M. An, and C. Lu. *Algorithms for Discrete Fourier Transform and Convolution*. Springer-Verlag, New York, NY, 1997.

[12] A. L. Toom. *Soviet Mathematics*, 3:714–716, 1963.

[13] S. Winograd. Some bilinear forms whose multiplicative complexity depends on the field of constants. *Math. Syst. Theor.*, 10:169–180, 1977.

[14] S. Winograd. *Arithmetic Complexity of Computations*. CBMS-NSF Regional Conf. Ser. Appl. Math. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1980.

[15] J. Xiong, J. Johnson, R. Johnson, and D. Padua. SPL: A Language and Compiler for DSP Algorithms. In *Proc. PLDI*, pages 298–308, 2001.