MATHEMATICS OF COMPUTATION
VOLUME 61, NUMBER 204
OCTOBER 1993, PAGES 833–847

FAST FOURIER TRANSFORMS FOR SYMMETRIC GROUPS: THEORY AND IMPLEMENTATION

MICHAEL CLAUSEN AND ULRICH BAUM

ABSTRACT. Recently, it has been proved that a Fourier transform for the symmetric group S_n based on Young's seminormal form can be evaluated in less than $0.5(n^3 + n^2)n!$ arithmetic operations. We look at this algorithm in more detail and show that it allows an efficient software implementation using appropriate data structures. We also describe a similarly efficient algorithm for the inverse Fourier transform. We compare the time and memory requirements of our program to those of other existing implementations.

1. INTRODUCTION

In 1965, Cooley and Tukey [6] published a fast algorithm for the evaluation of Discrete Fourier Transforms. Since then, the DFT and its variants have become extremely important tools in many areas such as digital signal processing. In terms of representation theory, the DFT describes an algebra isomorphism from the complex group algebra $\mathbb{C}C_n$ of the cyclic group C_n (the signal domain) onto the algebra of n-square diagonal matrices over $\mathbb{C}$ (the spectral domain).

Wedderburn's theorem allows us to generalize the DFT to arbitrary finite groups: Let K be a splitting field of the finite group G with $\operatorname{char} K \nmid |G|$. Then the group algebra KG is isomorphic to an algebra of block diagonal matrices: $KG \simeq \bigoplus_{i=1}^{h} K^{d_i \times d_i}$, where the blocks correspond to the equivalence classes of irreducible representations of KG and h is the number of conjugacy classes of G. Every algebra isomorphism

$$D = \bigoplus_{i=1}^{h} D_i : KG \to \bigoplus_{i=1}^{h} K^{d_i \times d_i}$$

is called a *Fourier transform* for KG. The constituents $D_1, \ldots, D_h$ of D form a transversal of irreducible representations of KG. With respect to natural bases in KG and $\bigoplus_i K^{d_i \times d_i}$, every Fourier transform of KG can (and will) be viewed as a matrix $D \in K^{|G| \times |G|}$ and every $a \in KG$ as a column vector in $K^{|G|}$.

This gives rise to three closely related computational problems: Efficient *construction* of a suitably encoded Fourier transform matrix for a given group, fast *evaluation* of such a transform, which amounts to computing a matrix-vector

Received by the editor April 16, 1992.
1991 *Mathematics Subject Classification.* Primary 20C30, 20C40; Secondary 68Q40, 68R05.

product, and finally fast *interpolation*, which means fast evaluation of the inverse Fourier transform. Usually, the construction is a precomputation step that has to be done only once for multiple evaluations.

The simplest way to do this would be to precompute and store the whole $|G|$-square Fourier matrix and then to evaluate it using the standard matrix-vector multiplication formula. But this procedure is very inefficient: The precomputation step requires computing the representing matrices $D_i(g)$ for all i and all $g \in G$. Obviously, this takes a great deal of time, and one has to store $|G|^2$ numbers. The evaluation would take of the order $|G|^2$ arithmetic operations. Both time and memory restrictions prohibit using this approach for large groups. To obtain practically feasible algorithms, one takes advantage of the group structure and the properties of the chosen transversal of irreducible representations. Nonabelian groups have irreducible representations of degree > 1 which have different matrix forms depending on the choice of bases in the corresponding simple KG-modules. Hence, there are essentially different Fourier transforms which can also widely differ in their construction and evaluation complexities.

We are going to define a computational model for the evaluation of Fourier transforms and their inverses: The *K-linear complexity* of a matrix $A \in K^{n \times n}$ is the minimal number of K-linear operations (i.e., additions, subtractions, and scalar multiplications) sufficient to evaluate A at a generic input vector x. As nonabelian groups have many Fourier transforms, we define the *K-linear complexity $L_K(G)$ of a finite group G* as the minimum of the K-linear complexities of all Fourier transforms for KG.

Obviously, $|G| - 1 \leq L_K(G) < 2|G|^2$. The classical FFT algorithms [6, 3, 17] show that $L_K(G) = O(|G| \log |G|)$ for abelian groups G. This has recently been extended to a much larger class containing the supersolvable groups [1] using monomial and symmetry-adapted representations which are also surprisingly simple to generate. In a restricted computational model, one can prove that these algorithms are asymptotically optimal [2].

This paper is concerned with fast Fourier transforms for symmetric groups and their implementation. Some interesting applications of these transforms in the statistical analysis of ranked data have been investigated by Diaconis [7, 8]. In the 1930s, Alfred Young found simple explicit formulae for two transversals of irreducibles of S_n, the so-called seminormal and orthogonal forms. Independently, Diaconis and Rockmore [9] and Clausen [5] have used Young's seminormal form as a basis of a more efficient algorithm for the evaluation of the corresponding Fourier transform of S_n. Although their methods are quite similar, the resulting upper bounds differ substantially: Diaconis and Rockmore show that

$$L_K(S_n) = O((n!)^{a/2} n),$$

provided that $(d \times d)$-matrices can be multiplied with $O(d^a)$ arithmetic operations. However, all known matrix multiplication algorithms with $a < 3$ are of no practical use for our problem: They use a large amount of memory and do not run faster than the naive algorithm for the d in question. Hence, with respect to implementations, one should assume $a = 3$. On the other hand,

Clausen has proved the explicit upper bound

$$L_K(S_n) < \tfrac{1}{2}(n^3 + n^2)n!, \tag{1.1}$$

which does not depend on advanced matrix multiplication methods.

Recently, Linton, Michler, and Olsson [13] suggested a completely different approach for computing Fourier transforms for symmetric groups. It is based on the Inglis-Richardson-Saxl model [10] consisting of a series $(\pi_t)_{0\le t\le \lfloor n/2\rfloor}$ of induced monomial representations π_t of degrees $n!/(2^t t!(n-2t)!)$. The crucial fact is that $\pi_0 \oplus \cdots \oplus \pi_{\lfloor n/2\rfloor}$ contains every irreducible representation of S_n with multiplicity one, i.e., for suitable block-diagonalizing matrices X_t the mapping $\bigoplus_t X_t \cdot \pi_t(\cdot) \cdot X_t^{-1}$ is a Fourier transform for S_n. This leads to an algorithm with an arithmetic complexity of

$$L_K(S_n) = O((n!)^{3/2} e^{\sqrt{n}} n^{-1/2}).$$

Diaconis and Rockmore, and Linton, Michler, and Olsson sketch implementations of their algorithms and present some information on their running times. Both implementations seem to be rather time- and space-consuming (see the tables in §5). In particular, the Fourier transform for S_{10} does not appear feasible using either of these algorithms.

In this paper, we look at the algorithm described in [5] and the necessary precomputations in more detail and show how to turn it into an efficient program using appropriate data structures. We also describe a similarly efficient implementation of the inverse Fourier transform. Finally, we compare our program to the other two implementations. It turns out that computing the Fourier transform for S_{10} is no problem with our program, even on a small workstation.

2. Fast precomputation

A uniform approach to designing efficient DFT algorithms is based on adapting the irreducible representations to a chain of subgroups: Suppose we want to evaluate a Fourier transform D of a finite group G at a generic input vector $a = \sum_{g\in G} a_g g \in KG$. Any algorithm based on the formula $D(a) = \sum_{g\in G} a_g D(g)$ has to specify the order of summation. This can be done in a natural way using subgroups: If U is a subgroup of G, one rewrites the input $a = \sum_{g\in G} a_g g \in KG$ according to the partition $G = \bigcup_j h_j U$ of G into left cosets of U: $a = \sum_j h_j a_j$ with $a_j := \sum_{u\in U} a_{h_j u} u \in KU$. Then $D(a) = \sum_j D(h_j) D(a_j)$ reduces the original problem to several evaluations of D at elements of KU. Now if the restriction of D to KU is itself (up to multiplicities) *equal* to a Fourier transform of the subgroup U, we can apply this method recursively without performing a base change. This idea can be formalized as follows.

Definition 2.1. Let $\mathcal{T} = (G = G_n > \cdots > G_0 = \{1\})$ be a tower of subgroups of the finite group G and K a splitting field of all G_i with $\operatorname{char} K \nmid |G|$.[1] A

[1] Unless otherwise specified, we will assume in this section that G, $\mathcal{T}$, and K are defined as here.

matrix representation D of KG is called *$\mathcal{T}$-adapted* if for all j, $0 \le j \le n$, the following conditions hold:

(a) The restriction $D \downarrow KG_j$ of D to KG_j is *equal* to a direct sum of irreducible matrix representations of KG_j.
(b) Equivalent irreducible constituents of $D \downarrow KG_j$ are *equal.*

As copying is free in our computational model, condition (b) allows us to use intermediate results several times, saving arithmetic operations. This will lead to more efficient DFT algorithms. The concept of symmetry-adapted representations has also been successfully applied to various mathematical and physical problems (see, e.g., [12, 14]).

An easy induction argument shows that every representation of KG is equivalent to a $\mathcal{T}$-adapted representation. Moreover, $\mathcal{T}$-adapted representations are almost unique under certain conditions:

Theorem 2.1. *For a $\mathcal{T}$-adapted representation D of KG, the following statements are equivalent:*

(a) *D is multiplicity-free and for all j, $1 \le j \le n$, if F is an irreducible constituent of $D \downarrow KG_j$, then $F \downarrow KG_{j-1}$ is multiplicity-free.*
(b) *If Δ is a $\mathcal{T}$-adapted representation equivalent to D, then there exists a monomial matrix X such that $D(a) = X^{-1}\Delta(a)X$ for all $a \in KG$. (In this case, we call D and Δ* monomially equivalent.)

A proof can be found in [4] or [1]. As we shall see later, the last theorem applies to the tower

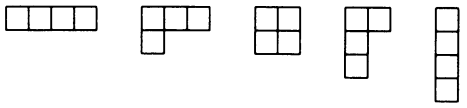
$$\mathcal{T}_n := (S_n > S_{n-1} > \cdots > S_1)$$

of S_n. Next, we are going to describe Young's seminormal form, which is a $\mathcal{T}_n$-adapted Fourier transform for S_n. Although it is well known (see, e.g., the standard text by James and Kerber [11]), we are going to revisit Young's construction from an implementation point of view.

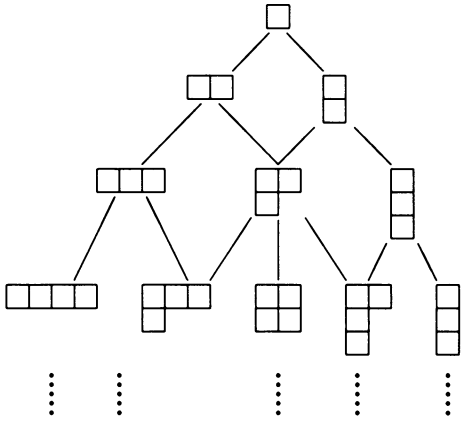
It is known that every field is a splitting field for S_n. So let K be any field with $\operatorname{char} K \nmid n!$. The conjugacy classes of S_n as well as the equivalence classes of irreducible representations of KS_n are usually parametrized by the partitions of n. A *partition* $\alpha = (\alpha_1, \alpha_2, \ldots)$ of n, abbreviated $\alpha \vdash n$, is a nonincreasing sequence of positive integers summing up to n. Partitions can be illustrated by the corresponding diagrams: The diagram of α is the set

$$\bigcup_i \{(i, j) \mid 1 \le j \le \alpha_i\},$$

which can be visualized as a left-justified arrangement of α_i boxes in the ith row. By abuse of notation we will make no difference between partitions and their diagrams. The following figure shows all diagrams for $n = 4$:



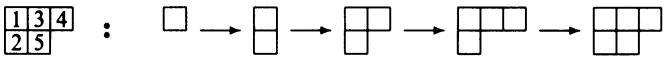
Hence, there are five unequivalent irreducible representations of KS_4. For every partition α of n, we denote by $[\alpha]$ an irreducible representation of S_n of "type" α. Inclusion defines a partial ordering on the set of all diagrams, the so-called *Young lattice*:



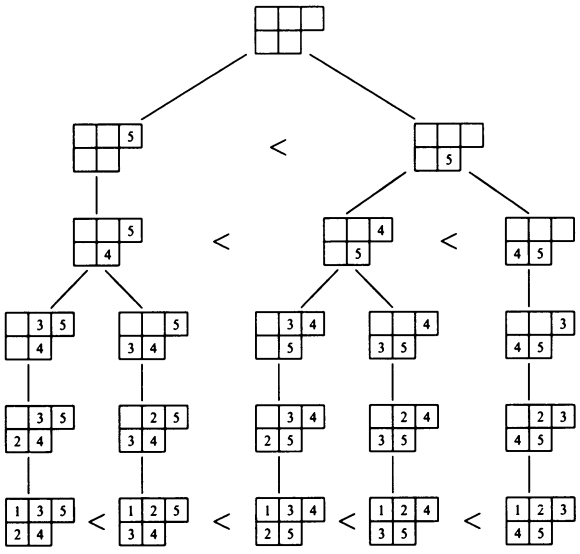
In fact, this has a representation-theoretic meaning: The celebrated branching theorem tells us that $[\alpha]\downarrow S_{n-1}$ is multiplicity-free:

$$[\alpha]\downarrow S_{n-1} \sim \bigoplus_{\beta} [\beta],$$

where the sum is over all diagrams β of $n-1$ contained in α. In particular, the degree f_α of $[\alpha]$ equals the number of paths from (1) to α in the Young lattice. Every such path can be described by a standard α-tableau, which is an α-shaped matrix whose elements $1, 2, \ldots, n$ are arranged in such a way that the entries in each row and column are strictly increasing:



In 1930, Alfred Young gave surprisingly simple explicit formulae for $\mathcal{T}_n$-adapted irreducible representations of S_n. They are based on the so-called *last letter sequence*, which is a total ordering of the standard α-tableaux. It can be described by the leaves of a certain tree. We omit a formal definition and illustrate it by an example, the last letter sequence of all standard $(3, 2)$-tableaux:



Since S_n is generated by all transpositions of consecutive numbers,

$$S_n = \langle (1,2), (2,3), \ldots, (n-1,n) \rangle,$$

every representation D of KS_n is completely determined by all $D(i, i+1)$, $i < n$. Let α be a partition of n. We are going to describe Young's seminormal form σ^α, which is an irreducible representation of KS_n of "type α". The rows and columns are parametrized by the last letter sequence $T_1 < \cdots < T_r$ of all standard α-tableaux, where $r := f_\alpha$. In order to describe for a fixed $i < n$

$$\sigma^\alpha(i, i+1) =: (\sigma_{kl})_{1 \le k,l \le r}, \tag{2.1}$$

we have to consider two cases.

Case 1. For $a \le r$ the numbers i and $i+1$ are in the same row (resp. column) of T_a: Then the only nonzero entry in the ath row and ath column of (σ_{kl}) is the diagonal position: $\sigma_{aa} = 1$, if i and $i+1$ are in the same row of T_a, whereas $\sigma_{aa} = -1$, if i and $i+1$ are in the same column of T_a.

Case 2. T_b results from T_a by interchanging i and $i+1$: Then, if $a < b$,

$$\begin{pmatrix} \sigma_{aa} & \sigma_{ab} \\ \sigma_{ba} & \sigma_{bb} \end{pmatrix} = \begin{pmatrix} d^{-1} & 1-d^{-2} \\ 1 & -d^{-1} \end{pmatrix}, \tag{2.2}$$

where $d := |u-x| + |v-y|$ is the axial distance of the positions (u, v) of i and (x, y) of $i+1$ in T_a. All other entries are zero. Consequently, the matrix (σ_{kl}) is sparse with at most two nonzero entries in each row and in each column. Table 1 shows all $\sigma^\alpha(i, i+1)$ for $n = 4$.

TABLE 1

last letter sequence	$\sigma^\alpha(1,2)$	$\sigma^\alpha(2,3)$	$\sigma^\alpha(3,4)$
$\begin{array}{\|c\|c\|c\|c\|}\hline 1&2&3&4\\\hline\end{array}$	(1)	(1)	(1)
$\begin{array}{\|c\|c\|c\|}\hline 1&3&4\\\hline\end{array}$ $\begin{array}{\|c\|c\|c\|}\hline 1&2&4\\\hline\end{array}$ $\begin{array}{\|c\|c\|c\|}\hline 1&2&3\\\hline\end{array}$ $\begin{array}{\|c\|}\hline 2\\\hline\end{array}$ $\begin{array}{\|c\|}\hline 3\\\hline\end{array}$ $\begin{array}{\|c\|}\hline 4\\\hline\end{array}$	$\begin{pmatrix} -1 & & \\ & 1 & \\ & & 1 \end{pmatrix}$	$\left(\begin{array}{ccc\|c} \frac{1}{2} & \frac{3}{4} & & \\ 1 & -\frac{1}{2} & & \\ & & 1 & \\ \hline & & & 1 \end{array}\right)$	$\left(\begin{array}{c\|ccc} 1 & & & \\ \hline & \frac{1}{3} & \frac{8}{9} & \\ & 1 & -\frac{1}{3} & \\ & & & 1 \end{array}\right)$
$\begin{array}{\|c\|c\|}\hline 1&3\\\hline 2&4\\\hline\end{array}$ $\begin{array}{\|c\|c\|}\hline 1&2\\\hline 3&4\\\hline\end{array}$	$\begin{pmatrix} -1 & \\ & 1 \end{pmatrix}$	$\begin{pmatrix} \frac{1}{2} & \frac{3}{4} \\ 1 & -\frac{1}{2} \end{pmatrix}$	$\begin{pmatrix} -1 & \\ & 1 \end{pmatrix}$
$\begin{array}{\|c\|c\|}\hline 1&4\\\hline\end{array}$ $\begin{array}{\|c\|}\hline 2\\\hline 3\\\hline\end{array}$ $\begin{array}{\|c\|c\|}\hline 1&3\\\hline\end{array}$ $\begin{array}{\|c\|}\hline 2\\\hline 4\\\hline\end{array}$ $\begin{array}{\|c\|c\|}\hline 1&2\\\hline\end{array}$ $\begin{array}{\|c\|}\hline 3\\\hline 4\\\hline\end{array}$	$\begin{pmatrix} -1 & & \\ & -1 & \\ & & 1 \end{pmatrix}$	$\left(\begin{array}{c\|cc} -1 & & \\ \hline & \frac{1}{2} & \frac{3}{4} \\ & 1 & -\frac{1}{2} \end{array}\right)$	$\left(\begin{array}{cc\|c} \frac{1}{3} & \frac{8}{9} & \\ 1 & -\frac{1}{3} & \\ \hline & & -1 \end{array}\right)$
$\begin{array}{\|c\|}\hline 1\\\hline 2\\\hline 3\\\hline 4\\\hline\end{array}$	(-1)	(-1)	(-1)

It can be shown that $\sigma_n := \bigoplus_{\alpha \vdash n} \sigma^\alpha$ is a $\mathcal{T}_n$-adapted Fourier transform for KS_n (see, e.g., Theorem 3.3.10 in [11]). More precisely,

$$\sigma^\alpha \downarrow S_{n-1} = \bigoplus_{\beta \vdash n-1 \,:\, \beta \subset \alpha} \sigma^\beta, \tag{2.3}$$

where the direct summands appear in the order given by the first level of the last letter sequence tree for α. For example,

$$\sigma^{(4,2,1)} \downarrow S_6 = \sigma^{(3,2,1)} \oplus \sigma^{(4,1,1)} \oplus \sigma^{(4,2)}.$$

As each $\sigma^\alpha \downarrow S_{n-1}$ is multiplicity-free, Theorem 2.1 tells us that σ_n is—up to monomial basis transforms—the unique $\mathcal{T}_n$-adapted Fourier transform for S_n. For example, Young's *orthogonal form*, which is another well-known $\mathcal{T}_n$-adapted DFT for S_n, is obtained by a monomial basis transform from σ_n. Our DFT algorithm for S_n requires that, for all $k \le n$, the matrices $\sigma^\beta(i, i+1)$ for all $\beta \vdash k$ and $i < k$ be precomputed and stored. One might think that this takes too much memory. But this is not the case if we use the right data structure: The sparse matrix $(\sigma_{ab}) := \sigma^\beta(i, i+1)$ is represented by a table $(b_a, d_a)_{1 \le a \le f_\beta}$ of pairs of integers satisfying $1 \le b_a \le f_\beta$, $-1 \le d_a < k$. The pair $(b, d) := (b_a, d_a)$ describes row a of the matrix. If $a = b$, then $\sigma_{aa} = d \in \{\pm 1\}$ according to Case 1 above. Otherwise, we are in Case 2. If $a < b$, then $\sigma_{aa} = d^{-1}$ and $\sigma_{ab} = 1 - d^{-2}$. Finally, if $a > b$, we have $\sigma_{ab} = 1$ and $\sigma_{aa} = -d^{-1}$. This encoding is slightly redundant, but allows a more efficient multiplication by (σ_{ab}). Altogether, we have to store

$$2 \sum_{k \le n} (k-1) \sum_{\beta \vdash k} f_\beta$$

small integers, a modest quantity compared to the input size $n! = \sum_{\alpha \vdash n} f_\alpha^2$. For example, for $n = 10$, this requires space for 227,376 small integers, while the input consists of $10! = 3,628,800$ floating-point numbers in case $K = \mathbb{R}$.

Now we describe the procedure for computing all $\sigma^\beta(i, i+1)$. In a first step, we generate the Young lattice up to level n in an obvious way. This gives us all partitions of $k \leq n$ as well as the degrees and the branching behavior of the corresponding representations in negligible time and space.

Next, we compute the $\sigma^\beta(i, i+1)$ "bottom-up" for $S_2, S_3, \ldots, S_n$. Suppose $\beta \vdash k \leq n$ and $i < k$. For $i < k-1$, we have

$$\sigma^\beta(i, i+1) = (\sigma^\beta \downarrow S_{k-1})(i, i+1) = \bigoplus_{\gamma \vdash k-1 \,:\, \gamma \subset \beta} \sigma^\gamma(i, i+1)$$

according to equation (2.3). Hence, the matrix $\sigma^\beta(i, i+1)$ can be assembled without any further calculation from its direct summands already computed at stage $k-1$.

Only for $i = k-1$, the matrix has to be constructed by Young's formula using the last letter sequence of β. To do this, we only need to know the positions of $k-1$ and k in the standard β-tableaux $T_1, \ldots, T_{f_\beta}$. Hence, we do not have to construct the complete last letter sequence tree, but merely the nodes $N_1 < \cdots < N_m$ of depth 2. (These are pictures containing only the two entries $k-1$ and k.) This makes the construction of $\sigma^\beta(k-1, k)$ substantially faster.

Each N_j defines an interval $I_j \subseteq \{T_1, \ldots, T_{f_\beta}\}$, which consists of all leaves of the subtree rooted at N_j. The size of I_j equals the degree of the representation corresponding to the partition of $k-2$ which is obtained by deleting the positions $k-1$ and k from N_j. (This degree has already been computed in the first step of the algorithm.) For each $1 \leq j \leq m$, exactly one of the following two cases occurs:

(i) $k-1$ and k are contained in the same row (resp. column) of N_j. Then Case 1 of Young's construction applies to all elements T_a in I_j.

(ii) For some $l > j$, N_j is transformed into N_l by interchanging $k-1$ and k. Then Case 2 of Young's construction applies to every pair (T_a, T_b), consisting of the uth element of I_j and I_l, respectively. Obviously, the axial distance d is the same for all these pairs.

In this way, the matrix $\sigma^\beta(k-1, k)$ can be quickly constructed. For $n = 10$, the construction of $\sigma^\beta(i, i+1)$ for all partitions $\beta \vdash k$ and all $1 \leq i < k$, $k = 2, \ldots, 10$, takes 270 milliseconds on a SUN SPARCstation 1.

3. Fast evaluation

In this section, we will show how the Fourier transform σ_n can be efficiently evaluated. Given an element $a = \sum_{g \in S_n} a_g g$ of KS_n, we have to compute $\sigma_n(a)$. To begin with, we decompose S_n into left cosets of the subgroup S_{n-1}: $S_n = \dot{\bigcup}_{j \leq n} g_{jn} S_{n-1}$, where g_{jn} is any fixed permutation in S_n which maps

n to j. The choice of g_{jn} is crucial for our algorithm's efficiency and will be discussed in a moment. According to this decomposition, we can write $a = \sum_{j=1}^{n} g_{jn} a_j$, where $a_j = \sum_{h \in S_{n-1}} a_{g_{jn} h} h \in KS_{n-1}$. As σ_n is a morphism of K-algebras, we get

$$\sigma_n(a) = \sum_{j=1}^{n} \sigma_n(g_{jn})\sigma_n(a_j) = \sum_{j=1}^{n} \bigoplus_{\alpha \vdash n} \sigma^\alpha(g_{jn})\sigma^\alpha(a_j)$$
$$= \sum_{j=1}^{n} \bigoplus_{\alpha \vdash n} \sigma^\alpha(g_{jn})(\sigma^\alpha \downarrow KS_{n-1})(a_j).$$

Now we use the fact that σ_n is $\mathcal{T}_n$-adapted:

$$\sigma_n(a) = \sum_{j=1}^{n} \bigoplus_{\alpha \vdash n} \sigma^\alpha(g_{jn}) \bigoplus_{\alpha \supset \beta \vdash n-1} \sigma^\beta(a_j).$$

Thus, every $\sigma^\beta(a_j)$, once computed, can be used in the evaluation of $\sigma^\alpha(a_j)$ for all partitions α of n containing β. Because copying is free in our computational model, this means a substantial reduction in the number of operations. The assumption that copying is free is realistic for practical software implementations, as copying an element of K is usually much faster than an arithmetic operation. This is the fundamental advantage of $\mathcal{T}$-adapted Fourier transforms. (Clausen [5] employs this idea to derive improved DFT algorithms for arbitrary finite groups.)

We still have to specify the coset representatives g_{jn}. We use the cycle $g_{jn} := (j, j+1, \ldots, n)$. Why is this a good choice? As $g_{jn} = (j, j+1) \cdot (j+1, j+2) \cdots (n-1, n)$, $\sigma^\alpha(g_{jn})$ is the product of $n - j$ sparse matrices,

$$\sigma^\alpha(g_{jn}) = \sigma^\alpha(j, j+1) \cdot \sigma^\alpha(j+1, j+2) \cdots \sigma^\alpha(n-1, n).$$

Recall that $\sigma^\alpha(i, i+1)$ has at most two nonzero entries in each row and at most $\frac{3}{4}$ of its nonzero entries are $\neq \pm 1$. Hence we can multiply $\sigma^\alpha(i, i+1)$ with an arbitrary f_α-square matrix in at most $\frac{5}{2} \cdot f_\alpha^2$ operations. Instead of directly multiplying the "twiddle factor" $\sigma^\alpha(g_{jn})$ (which is typically not sparse) by $\sigma^\alpha(a_j)$, we compute the product as indicated by the expression

$$(3.1) \quad (\sigma^\alpha(j, j+1) \cdots (\sigma^\alpha(n-2, n-1) \cdot (\sigma^\alpha(n-1, n) \cdot \sigma^\alpha(a_j))) \cdots)$$

and can thus perform this multiplication with at most $\frac{5}{2} \cdot (n-j) f_\alpha^2$ arithmetic operations, compared to order f_α^3 for direct multiplication.[2]

Finally, we have to sum up the n blockdiagonal matrices $\sigma_n(g_j)\sigma_n(a_j)$. This takes

$$(n-1) \sum_{\alpha \vdash n} f_\alpha^2 = (n-1) \cdot n!$$

operations.

---

[2]Diaconis and Rockmore also use σ_n, but take $g_{jn} = (j, n)$ and directly multiply $\sigma^\alpha(g_{jn})$ by $\sigma^\alpha(a_j)$. This gives an upper bound of order $(n!)^{3/2} \cdot n$ for $L_K(S_n)$.

Let L_n denote the arithmetic cost of our algorithm to evaluate σ_n. Then we have the recursion

$$L_n \leq n \cdot L_{n-1} + \left(\sum_{j=1}^{n} \sum_{m=j+1}^{n} \sum_{\alpha \vdash n} \frac{5}{2} \cdot f_\alpha^2 \right) + (n-1) \cdot |S_n|.$$

Obviously, $L_1 = 0$. When this is combined with $\sum_{\alpha \vdash n} f_\alpha^2 = |S_n|$ and the well-known formula $\sum_{i=2}^{n} \binom{i}{2} = \binom{n+1}{3}$, induction yields the following small improvement of Clausen's original result (1.1).

Theorem 3.1. *We have*

$$L_K(S_n) \leq L_n \leq (\tfrac{5}{12}n^3 + \tfrac{1}{2}n^2 - \tfrac{11}{12}n)n!.$$

(By a closer look at the matrices in (3.1), this upper bound can be slightly improved. However, the gain is too small to justify the much more technical proof.)

So far, we have followed a top-down approach to describe and analyze our algorithm. However, a top-down implementation recursively computing the Fourier transform would require too much memory and a great deal of bookkeeping (see the discussion in [9]). Our implementation works bottom-up to avoid these problems. It takes a global view of all computations at *layer* k of the algorithm, where the Fourier transform σ_k of S_k is evaluated once for each coset of S_k in S_n. More precisely, define $g_{jk} := (j, j+1, \ldots, k)$ for $1 \leq j \leq k \leq n$. Then the words

$$g_{\underline{j}} := g_{j_n n} g_{j_{n-1} n-1} \cdots g_{j_{k+1} k+1}, \qquad 1 \leq j_i \leq i,$$

form a transversal of left cosets of S_k in S_n. The lexicographical ordering of the vectors $(j_n, \ldots, j_{k+1})$ thus induces a total ordering of the cosets.

The data in layer k of our algorithm consists of $n!/k!$ blocks $\sigma_k(a_{\underline{j}})$ of size $k!$, where the $a_{\underline{j}} \in KS_k$ are defined by decomposing the input $a \in KS_n$ along our transversal: $a = \sum_{\underline{j}} g_{\underline{j}} a_{\underline{j}}$. As we shall see below, it is favorable for our algorithm to arrange the blocks in each layer k according to the lexicographical ordering of the cosets of S_k in S_n. Each blockdiagonal matrix $\sigma_k(a_{\underline{j}})$ is represented by a vector of length $k!$ as follows: The block constituents are ordered according to the lexicographic order of the partitions of k. Within each block, the entries are stored row-wise. With respect to this arrangement, the data in layer k can be described as a vector $v_k \in K^{n!}$. In particular, the input vector v_1 consists of the coefficients of a enumerated in the lexicographical order of the group elements.

Given v_1, our program successively computes $v_2, \ldots, v_n$ and outputs v_n. In layer k, we compute v_k from v_{k-1}. More precisely, we compute each $\sigma_k(a_{\underline{j}})$ from the k blocks $\sigma_{k-1}(a_{\underline{l}})$ with $g_{\underline{l}} S_{k-1} \subset g_{\underline{j}} S_k$ as described above. As our ordering of the $\sigma_k(a_{\underline{j}})$ is compatible with the stepwise refinement of our coset decomposition, the same linear transformation has to be applied to each block of length $k!$ in v_{k-1} to obtain the corresponding block in v_k. In a (sequential) software implementation, this is easy to realize by moving pointers

over v_k and v_{k-1}. On the other hand, it is obvious that a very regular *parallel* implementation is possible. The layered structure of the algorithm also allows pipelining.

What about memory requirements? Of course, we do not have to store $(v_1, \dots, v_n)$, but only v_k and v_{k+1} at layer k. In addition, we need storage for $f_\alpha^2 + 2f_\alpha$ elements of K for evaluating the expression (3.1). Altogether, our algorithm requires memory for

$$M(n) := 2 \cdot n! + f_n^2 + 2 \cdot f_n$$

elements of K, where f_n denotes the maximal degree of an irreducible representation of KS_n. For $n = 10$, we have $f_{10} = 768$ and $M(10) < 2.17 \cdot 10!$. In 32-bit floating-point arithmetic $(K = \mathbb{R})$, this takes less than 30 megabytes of virtual memory. In addition, we need less than one megabyte for storing the precomputed tables (see §2) and local variables. So the Fourier transform for S_{10} is already feasible on a medium-sized workstation.

4. Fast interpolation

In this section we will design an efficient interpolation algorithm for symmetric groups that evaluates the inverse Fourier transforms σ_n^{-1}. Our algorithm is based on the well-known Fourier inversion formula [16, p. 49], which in our case reads as follows:

$$\sigma_n^{-1}\left(\bigoplus_{\alpha \vdash n} A_\alpha\right) = \frac{1}{|S_n|} \sum_{g \in S_n} \left(\sum_\alpha f_\alpha \operatorname{tr}(\sigma^\alpha(g^{-1}) \cdot A_\alpha)\right) g$$

for every blockdiagonal matrix $\bigoplus_\alpha A_\alpha \in \sigma_n(KS_n)$. We rewrite this, using our coset decomposition $S_n = \bigcup_j g_{jn} S_{n-1}$:

$$\begin{aligned}\sigma_n^{-1}\left(\bigoplus_{\alpha \vdash n} A_\alpha\right) &= \sum_{j \le n} g_{jn} \left[\frac{1}{|S_{n-1}|} \sum_{g \in S_{n-1}} \left(\sum_\alpha \frac{f_\alpha}{n} \operatorname{tr}(\sigma^\alpha(g^{-1}) \cdot \sigma^\alpha(g_{jn}^{-1}) \cdot A_\alpha)\right) g\right] \\ &=: \sum_{j \le n} g_{jn} X_{jn}.\end{aligned}$$

As σ_n is $\mathcal{T}_n$-adapted, we have

$$\operatorname{tr}(\sigma^\alpha(g^{-1}) \cdot \sigma^\alpha(g_{jn}^{-1}) \cdot A_\alpha) = \operatorname{tr}\left(\left(\bigoplus_{\alpha \supset \beta \vdash n-1} \sigma^\beta(g^{-1})\right) \cdot \sigma^\alpha(g_{jn}^{-1}) \cdot A_\alpha\right)$$

for all $g \in S_{n-1}$. Let p_α be the natural projection of $K^{f_\alpha \times f_\alpha}$ onto the blockdiagonal $\bigoplus_{\alpha \supset \beta \vdash n-1} K^{f_\beta \times f_\beta}$. Because only the diagonal elements contribute to the trace of a matrix, we have

$$\operatorname{tr}\left(\left(\bigoplus_{\alpha \supset \beta \vdash n-1} \sigma^\beta(g^{-1})\right) \cdot \sigma^\alpha(g_{jn}^{-1}) \cdot A_\alpha\right) = \sum_\beta \operatorname{tr}(\sigma^\beta(g^{-1}) \cdot A_{\alpha j}^\beta),$$

where $\bigoplus_\beta A^\beta_{\alpha j} := p_\alpha(\sigma^\alpha(g_{jn}^{-1}) \cdot A_\alpha)$. Now we see that

$$\begin{aligned} X_{jn} &= \frac{1}{|S_{n-1}|} \sum_{g \in S_{n-1}} \left(\sum_\alpha \frac{f_\alpha}{n} \operatorname{tr}(\sigma^\alpha(g^{-1}) \cdot \sigma^\alpha(g_{jn}^{-1}) \cdot A_\alpha) \right) g \\ &= \frac{1}{|S_{n-1}|} \sum_{g \in S_{n-1}} \left(\sum_\alpha \frac{f_\alpha}{n} \sum_{\beta \subset \alpha} \operatorname{tr}(\sigma^\beta(g^{-1}) \cdot A^\beta_{\alpha j}) \right) g \\ &= \frac{1}{|S_{n-1}|} \sum_{g \in S_{n-1}} \left(\sum_\beta f_\beta \operatorname{tr} \left(\sigma^\beta(g^{-1}) \cdot \left(\sum_{\alpha \supset \beta} \frac{f_\alpha}{n f_\beta} A^\beta_{\alpha j} \right) \right) \right) g \\ &= \frac{1}{|S_{n-1}|} \sum_{g \in S_{n-1}} \left(\sum_\beta f_\beta \operatorname{tr}(\sigma^\beta(g^{-1}) \cdot A^\beta_j) \right) g = \sigma_{n-1}^{-1} \left(\bigoplus_\beta A^\beta_j \right), \end{aligned}$$

where

$$A^\beta_j := \sum_{\alpha \supset \beta} \frac{f_\alpha}{n f_\beta} A^\beta_{\alpha j}.$$

Altogether,

$$\sigma_n^{-1} \left(\bigoplus_{\alpha \vdash n} A_\alpha \right) = \sum_{j \le n} g_{jn} \sigma_{n-1}^{-1} \left(\bigoplus_\beta A^\beta_j \right). \tag{4.1}$$

To compute $\sigma_n^{-1}(\bigoplus_\alpha A_\alpha)$, we proceed as indicated by the previous equations: In a first step, we compute all the products $\sigma^\alpha(g_{jn}^{-1}) \cdot A_\alpha$. Decomposing $g_{jn}^{-1} = (n-1, n)(n-2, n-1) \cdots (j, j+1)$ and proceeding as in (3.1), we obtain all these products (and hence the blocks $A^\beta_{\alpha j}$) in at most $\frac{5}{2}\binom{n}{2} \cdot n!$ arithmetic operations. Next, we compute all A^β_j in the obvious way using at most

$$\begin{aligned} & n \sum_{\beta \vdash n-1} (2 \cdot |\{\alpha \vdash n \mid \alpha \supset \beta\}| - 1) \cdot f_\beta^2 \\ & \quad \le n \cdot \left(2 \max_\beta (|\{\alpha \vdash n \mid \alpha \supset \beta\}|) - 1 \right) \sum_{\beta \vdash n-1} f_\beta^2 \\ & \quad \le n \cdot (2(1 + \sqrt{2(n-1)}) - 1) \cdot (n-1)! = (1 + 2\sqrt{2(n-1)}) \cdot n! \end{aligned}$$

operations. Finally, we compute $\sigma_{n-1}^{-1}(\bigoplus_\beta A^\beta_j)$ for all j, recursively applying our algorithm.

Let Λ_n denote the arithmetic cost of our algorithm to evaluate σ_n^{-1}. Then we have the recursion

$$\Lambda_n \le \frac{5}{2}\binom{n}{2} \cdot n! + (1 + 2\sqrt{2(n-1)}) \cdot n! + n \cdot \Lambda_{n-1}$$

with $\Lambda_1 = 0$. Induction shows that

$$\Lambda_n \le \left(\frac{5}{2}\binom{n+1}{3} + (n-1) + 2\sqrt{2} \sum_{k=1}^{n-1} \sqrt{k} \right) \cdot n!.$$

Further estimation, using $\sum_{k=1}^{n-1} \sqrt{k} \leq \int_1^n \sqrt{x}\,dx < n^{3/2}$, yields the final result.

Theorem 4.1. *There holds*

$$L_K(\sigma_n^{-1}) < (\tfrac{5}{12}n^3 + \tfrac{4}{3}\sqrt{2}n^{3/2} + \tfrac{7}{12}n) \cdot n!.$$

(This upper bound can also be slightly improved at the expense of a much more technical proof.)

Our interpolation program is very similar to our evaluation program "read backwards" and uses exactly the same data structures. Hence, all comments on the implementation made in §3 apply respectively. In particular, our data arrangement is also compatible with the successive coset decompositions implied by equation (4.1).

5. Running times and comparisons

We have implemented our algorithm in C on a SUN SPARCstation 1. The C program is less than 1000 lines long. It works over $K = \mathbb{R}$, using 32-bit `float` arithmetic. For $6 \leq n \leq 10$, Table 2 shows the precomputation time (in CPU seconds), size of precomputed tables (in kilobytes), evaluation time (in CPU seconds), number of arithmetic operations, and the theoretical upper bound of our FFT algorithm. (For total memory requirements, see the discussion at the end of §3.) Table 3 gives the same data for the fast Fourier inversion algorithm. The precomputation times and memory requirements are the same as before. We see that our theoretical analysis comes rather close to reality: the running times of our algorithms are about proportional to the number of arithmetic operations. This is made possible by our choice of data structures, which results in low bookkeeping costs.

Let us now look at the performance of the other two algorithms mentioned. Unfortunately, Diaconis and Rockmore [9, 15] do not present any running times of their algorithm. To be able to compare actual running times, we have implemented the "dynamic programming" variant (Algorithm 5 in [9]) of Diaconis

Table 2

n	precomp. (s)	table size (kb)	eval. (s)	arith. ops	upper bound
6	0.01	2	0.08	55,440	73,800
7	0.02	7	0.79	623,952	811,440
8	0.03	28	8.67	7,507,836	9,596,160
9	0.08	110	105.68	96,756,840	121,927,680
10	0.26	444	1518.62	1,333,294,380	1,660,176,000

Table 3

n	eval. (s)	arith. ops	upper bound
6	0.11	60,696	87,273
7	0.99	663,600	916,887
8	10.20	7,823,868	10,510,079
9	118.03	99,337,932	130,604,753
10	1630.98	1,354,098,380	1,749,547,826

TABLE 4

n	precomp. (s)	table size (kb)	eval. (s)	arith. ops
6	0.03	16	0.12	61,920
7	0.15	134	1.64	1,110,144
8	1.43	1236	28.52	23,489,536
9	15.75	12576	633.34	576,440,064
10	195.79	140152	18493.50	16,532,519,760

TABLE 5

n	eval. (s)
6	10.27
7	35.81
8	643.94
9	44791.26

and Rockmore, because it takes less memory and runs faster than the other variants. In fact, this is very similar to our algorithm, as we have already seen in §3. The major difference is that they use the transpositions $g_{jn} := (j, n)$ as coset representatives and store all $\sigma^{\alpha}(j, n)$. As these matrices are typically not sparse, this takes an enormous amount of memory, e.g., about 140 megabytes for $n = 10$. Moreover, direct multiplication with such a matrix takes $O(d^3)$ operations, where d is its dimension. Our implementation of their algorithm uses the same data structures as with our algorithm, except that the $\sigma^{\alpha}(j, n)$ are stored as full matrices and the data arrangement in each layer is adapted to the coset representatives (j, n). On a SPARCstation 1, we obtained the results shown in Table 4.

The running times of Linton, Michler, and Olsson's [13] algorithm on an IBM RISC 6000/540, which is faster than our SPARCstation 1 (evaluation times in CPU seconds), are given in Table 5. They do not present any information on their algorithm's memory requirements.

BIBLIOGRAPHY

1. U. Baum, *Existence and efficient construction of fast Fourier transforms on supersolvable groups*, Comput. Complex. **1/3** (1992), 235–256.
2. U. Baum and M. Clausen, *Some lower and upper complexity bounds for generalized Fourier transforms and their inverses*, SIAM J. Comput. **2** (1991), 451–459.
3. L. I. Bluestein, *A linear filtering approach to the computation of the discrete Fourier transform*, IEEE Trans. **AU-18** (1970), 451–455.
4. M. Clausen, *Beiträge zum Entwurf schneller Spektraltransformationen*, Habilitationsschrift, Universität Karlsruhe, 1988.
5. ———, *Fast generalized Fourier transforms*, Theoret. Comput. Sci. **67** (1989), 55–63.
6. J. W. Cooley and J. W. Tukey, *An algorithm for the machine calculation of complex Fourier series*, Math. Comp. **19** (1965), 297–301.
7. P. Diaconis, *A generalization of spectral analysis with application to ranked data*, Ann. Statist. **17** (1989), 949–979.

8. ———, *Group representations in probability and statistics*, IMS Lecture Notes—Monograph Ser., vol. 11, Inst. Math. Statist., Hayward, CA, 1988.
9. P. Diaconis and D. Rockmore, *Efficient computation of the Fourier transform on finite groups*, J. Amer. Math. Soc. **3** (1990), 297–332.
10. N. F. J. Inglis, R. W. Richardson, and J. Saxl, *An explicit model for the complex representations of* S_n, Arch. Math. (Basel) **54** (1990), 258–259.
11. G. D. James and A. Kerber, *The representation theory of the symmetric group*, Addison-Wesley, Reading, MA, 1981.
12. D. J. Klein, C. H. Carlisle, and F. A. Matsen, *Symmetry adaptation to sequences of finite groups*, Adv. Quantum Chemistry, vol. 5, Academic Press, New York, 1970, pp. 219–260.
13. S. A. Linton, G. O. Michler, and J. B. Olsson, *Fast Fourier transforms on symmetric groups*, preprint, Universität Essen, 1991.
14. K. Murota and K. Ikeda, *Computational use of group theory in bifurcation analysis of symmetric structures*, SIAM J. Sci. Statist. Comput. **12** (1991), 273–297.
15. D. Rockmore, *Computation of Fourier transforms on the symmetric group*, Computers and Mathematics (E. Kaltofen and S. M. Watt, eds.), Springer, Berlin and New York, 1989, pp. 156–165.
16. J. P. Serre, *Linear representations of finite groups*, Springer, Berlin and New York, 1977.
17. S. Winograd, *On computing the discrete Fourier transform*, Math. Comp. **32** (1978), 175–199.

INSTITUT FÜR INFORMATIK, UNIVERSITÄT BONN, RÖMERSTRASSE 164, 5300 BONN, GERMANY
E-mail address, M. Clausen: clausen@leon.cs.uni-bonn.de
E-mail address, U. Baum: uli@leon.cs.uni-bonn.de