# $\mathbb{S}_n$`ob` : a C++ toolkit for fast Fourier transforms on the symmetric group

Development version (unstable)

Risi Kondor
Department of Computer Science
Columbia University in the City of New York

August 20, 2006

# Contents

# Chapter 1

# Introduction

$\mathbb{S}_n$`ob` is an object oriented `C++` library for computing fast Fourier transforms (FFTs) and inverse fast Fourier transforms (iFFTs) on the symmetric group $\mathbb{S}_n$. $\mathbb{S}_n$`ob` supports partial Fourier transforms and Fourier transforms on sparse functions, making computations possible in the $n = 10 \sim 40$ range, where enumerating all $n!$ group elements is no longer feasible. $\mathbb{S}_n$`ob` can also perform operations on individual group elements, compute characters, and compute representation matrices.

$\mathbb{S}_n$`ob` is optimized for speed and designed to provide a clean, easy to use application programming interface. Where this does not hinder performance, $\mathbb{S}_n$`ob` uses the `STL` Standard Template Library. $\mathbb{S}_n$`ob` does not use any general purpose numerical analysis or matrix libraries.

> The current release of $\mathbb{S}_n$`ob` is a development version, subject to corrections and design changes. $\mathbb{S}_n$`ob` comes with absolutely no guarantees regarding the accuracy of the results of its computations, the soundness of the code, or the documentation. Please reports errors and send suggestions to `risi@cs.columbia.edu`. Feedback is greatly appreciated.

$\mathbb{S}_n$`ob` is distributed in the form of a collection of source files, and has been tested on a Linux system with the GNU C++ compiler.

$\mathbb{S}_n$`ob` is open source software, released into the public domain under the GNU General Public License. Commercial use of $\mathbb{S}_n$`ob` is prohibited. For details see `http://www.gnu.org/copyleft/gpl.html` or the file `LICENSE` distributed with the source code. The copyright to the present document is reserved by the author.

Please cite $\mathbb{S}_n$`ob` in all scientific publications reporting computations performed with $\mathbb{S}_n$`ob`, presenting extensions of $\mathbb{S}_n$`ob`, or describing $\mathbb{S}_n$`ob` itself. The appropriate `bibtex` entry is supplied with the package in the file `Snob.bib`.

Risi Kondor, New York, August, 2006

# Chapter 2

# Mathematical Background

Given $n$ distinct objects $x_1, x_2, \ldots, x_n$, a permutation of these objects is a bijective map $\sigma\colon \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$, where $\sigma(i)$ expresses which position $x_i$ appears in in the permuted sequence. Thus, the permutation of $x_1, x_2, \ldots, x_n$ induced by $\sigma$ is $\big(x_{\sigma^{-1}(1)}, x_{\sigma^{-1}(2)}, \ldots, x_{\sigma^{-1}(n)}\big)$.

The product of two permutations $\sigma_1$ and $\sigma_2$ is defined by composition, i.e, $(\sigma_2\sigma_1)(i) = \sigma_2(\sigma_1(i))$ for all $i = 1, 2, \ldots, n$. The inverse of $\sigma$ is the permutation $\sigma^{-1}$ corresponding to the inverse mapping. Under these operations the $n!$ different permutations of $n$ objects form a group called the **symmetric group** of **order** n. We denote this group $\mathbb{S}_n$. The identity element $e$ of $\mathbb{S}_n$ is the identity permutation, $e(i) = i$.

We use the letter $\rho$ to denote complex valued matrix representations of $\mathbb{S}_n$, and the dimensionality of $\rho$ we denote by $d_\rho$. Thus, $\rho$ is a map $\mathbb{S}_n \to \mathbb{C}^{d_\rho \times d_\rho}$. A complete set of inequivalent irreducible representations of $\mathbb{S}_n$ we denote by $\mathcal{R}_n$.

The **Fourier transform** (FT) of a function $f\colon \mathbb{S}_n \to \mathbb{C}$ is defined as the set of matrices

$$\widehat{f}(\rho) = \sum_{\sigma \in \mathbb{S}_n} f(\sigma)\, \rho(\sigma) \qquad \rho \in \mathcal{R}_n. \tag{2.1}$$

The inverse transform is

$$f(\sigma) = \frac{1}{n!} \sum_{\rho \in \mathcal{R}_n} d_\rho \operatorname{tr}\Big[\widehat{f}(\rho)\, \rho(\sigma)\Big] \qquad \sigma \in \mathbb{S}_n. \tag{2.2}$$

The Fourier transform $\mathfrak{F}\colon f \mapsto \widehat{f}$ is a linear transformation from $\mathbb{C}^{n!}$ to $\bigoplus_{\rho \in \mathcal{R}} \mathbb{C}^{d_\rho \times d_\rho}$, sharing several important properties with the classical Fourier transforms. In particular, $\mathfrak{F}$ is a unitary map with respect to the inner products

$$\langle f, g\rangle = \frac{1}{n!} \sum_{\sigma \in \mathbb{S}_n} f(\sigma)\, \overline{g(\sigma)} \qquad \text{and} \qquad \langle \widehat{f}, \widehat{g}\rangle = \frac{1}{(n!)^2} \sum_{\rho \in \mathcal{R}_n} d_\rho\, \langle \widehat{f}(\rho), \widehat{g}(\rho)\rangle_{\mathrm{S}}, \tag{2.3}$$

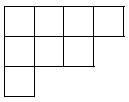
where $\langle \cdot, \cdot \rangle_{\mathrm{S}}$ is the usual inner product of matrices $\operatorname{tr}\big(A\overline{B}\big)$. Another important property of $\mathfrak{F}$ is that defining the **convolution** of $f, g\colon \mathbb{S}_n \to \mathbb{C}$ as $(g * f)(\sigma) = \sum_{\tau \in \mathbb{S}_n} g(\tau\sigma^{-1})\, f(\sigma)$,

$$\widehat{(g * f)}(\rho) = \widehat{g}(\rho) \cdot \widehat{f}(\rho) \qquad \rho \in \mathcal{R}_n, \tag{2.4}$$

where $\cdot$ denotes matrix multiplication. In fact, unitarity and the convolution theorem together uniquely define $\mathfrak{F}$, up to conjugacy of Fourier components. Much of the practical interest in Fourier transformation stems from these properties.

## 2.1 The irreducible representations of the symmetric group

A **partition** of $n$ is a sequence $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_k)$ of weakly decreasing positive integers such that $\sum_{i=1}^{k} \lambda_i = n$. We use the notation $\lambda \vdash n$ to denote that $\lambda$ is a partition of $n$. Graphically partitions are represented by **Young diagrams**, which consist of laying down $\lambda_1, \lambda_2, \ldots, \lambda_k$ boxes in consecutive rows. For example,



corresponds to $\lambda = (4, 3, 1)$. There is a natural partial order induced on partitions by inclusion, whereby $\lambda' \preceq \lambda$ if and only if $\lambda'_i \leq \lambda_i$ for $i = 1, 2, \ldots, k$ (if $k' < k$, we take $\lambda'_i = 0$ for $k' < i \leq k$). The partitions $\lambda \vdash n$ index the irreducibles of $\mathbb{S}_n$. The representation corresponding to $\lambda$ we denote $\rho_\lambda$; its dimensionality $d_\lambda$; the corresponding character $\chi_\lambda(\sigma) = \mathrm{tr}\,[\rho_\lambda(\sigma)]$; and the corresponding Fourier component $\widehat{f}(\lambda)$.

Bijectively filling in the boxes of a Young diagram with the numbers $1, 2, \ldots, n$ gives a **Young tableau**. A tableau such as

| | | |
|---|---|---|
| 1 | 2 | 4 |
| 3 | 5 | |

which satisfies the property that in each row the numbers increase from left to right and in each column they increase from top to bottom is called a **standard tableau**. A key result in the representation theory of the symmetric group asserts that the Young tableaux of shape $\lambda$ are in bijection with the dimensions of $\rho_\lambda$. We use standard tableaux to index the rows and columns of the representation matrices.

There are several alternatives for constructing the actual matrix entries. Because of its computational advantages, in $\mathbb{S}_n$ob we exclusively use **Young's orthogonal representation** (YOR). To present YOR we need a few more concepts. A **transposition** $(i, j)$ is a permutation that exhanges $i$ and $j$ and leaves everything else invariant, i.e., $(i, j)(i) = j$, $(i, j)(j) = i$ and $(i, j)(k) = k$ for all $k \notin \{i, j\}$. **Adjacent transpositions** are transpositions of the form $\tau_i = (i, i+1)$. It is not difficult to see that any $\sigma \in \mathbb{S}_n$ can be expressed as a product of adjacent transpositions, in fact, one such factorization is given in the next section. We define YOR by explicitly constructing the matrix entries of $\rho_\lambda(\tau_k)$. For general $\sigma$, the representation matrix $\rho_\lambda(\sigma)$ is then computed by factoring $\sigma$ into a product of adjacent transpositions and multiplying the corresponding $\rho_\lambda(\tau_k)$.

We define YOR through the action of permutations on tableaux: if $t$ is a standard tableau, then $\tau_k(t)$ is the tableau that we get by interchanging $k$ and $k+1$ in $t$. However, $\tau_k(t)$ might or might not be a standard tableau. Accordingly, there are two distinct cases to consider. If $\tau_k(t)$ is not a standard tableau, then the column of $\rho_\lambda(\tau_k)$ labeled by $t$ in YOR is all zero, except for the diagonal element

$$[\rho_\lambda(\tau_k)]_{t,t} = 1/d_t(k, k+1). \tag{2.5}$$

If $\tau_k(t)$ is a standard tableau, then in addition to this diagonal element, we also have a non-zero off-diagonal element, namely

$$[\rho_\lambda(\tau_k)]_{\tau_k(t),t} = \sqrt{1 - 1/d_t(k, k+1)^2}. \tag{2.6}$$

All other matrix entried of $\rho_\lambda(\tau_k)$ are zero. In the above $d_t(k, k+1)$ is a special signed distance defined on Young tableaux. If $i$ is a numeral in $t$, then the **content** of $i$, denoted $c(i)$, is the column index minus the row index of the cell where $i$ is to be found. The distance $d_t(i, j)$ is then defined as $c(j) - c(i)$. Part of the computational appeal of YOR is that the $\rho_\lambda(\tau_k)$ matrices are very sparse, and hence can be mutliplied together efficiently.

Another special property of YOR is that the representations matrices are real. This allows us to restrict the entire implementation of $\mathbb{S}_n$ob to the real domain. In the following pages we consider the more general complex case, while in the code the user is free to choose the base field by setting a master variable at compile time.

## 2.2 Fast Fourier Transforms

The Fourier transform on $\mathbb{S}_n$ is a linear transformation between $n!$-dimensional spaces. Thus, a naive implementation of Fourier transformation would run in $O(|\,\mathbb{S}_n\,|^2) = O((n!)^2)$ time. The recent development of fast Fourier transforms (FFTs) for non-commutative groups, starting with Clausen's FFT for $\mathbb{S}_n$ [Clausen, 1989] opened a new chapter in computational group theory by reducing this computational burden to $O(|\,\mathbb{S}_n\,|\log|\,\mathbb{S}_n\,|)$. See [Maslen and Rockmore, 1997] for an excellent survey.

Clausen's algorithm uses a specific type of decomposition of permutations into a product of **contiguous cycles**. The contiguous cycle $[\![p,q]\!] \in \mathbb{S}_n$ is the permutation

$$[\![p,q]\!](i) = \begin{cases} i+1 & \text{for } p \le i \le q-1 \\ p & \text{for } i = q \\ i & \text{otherwise} \end{cases} \qquad 1 \le p \le q \le n.$$

In other words, $[\![p,q]\!]$ cyclically permutes $p, p+1, \ldots, q$ and leaves everything else invariant. $[\![p,q]\!]$ can easily be factored into a product of $q-p-1$ adjacent transpositions:

$$[\![p,q]\!] = (p,\, p+1)\ldots(q-2,\, q-1)(q-1,\, q).$$

Now observe that for any $\sigma \in \mathbb{S}_n$, the product $[\![\sigma(n),n]\!]^{-1}\sigma$ fixes $n$, i.e.,

$$\sigma_{n-1} = [\![\sigma(n),n]\!]^{-1}\sigma \;\in\; \mathbb{S}_{n-1}, \tag{2.7}$$

where (as in the following) for $k < n$, $\mathbb{S}_k$ is the subgroup of $\mathbb{S}_n$ permuting $1,2,\ldots,k$, and leaving $k+1, k+2,\ldots,n$ fixed. Applying the same manouver to $\sigma_{n-1}$,

$$\sigma_{n-2} = [\![\,\sigma_{n-1}(n-1)\,,\; n-1\,]\!]^{-1}\,[\![\sigma(n),n]\!]^{-1}\sigma \;\in\; \mathbb{S}_{n-2}.$$

Iterating down to $\mathbb{S}_1$, we get $[\![p_2,2]\!]^{-1}[\![p_3,3]\!]^{-1}\ldots[\![p_n,n]\!]^{-1}\sigma = e$ for some $p_2,p_3,\ldots,p_n$. Rather than the actual values of $p_2,p_3,\ldots,p_n$, what is important is that this procedure furnishes a unique factorization of $\sigma$,

$$\sigma = [\![p_n,n]\!]\,[\![p_{n-1},n-1]\!]\ldots[\![p_2,2]\!],$$

adapted to the chain of subgroups $\mathbb{S}_1 < \mathbb{S}_2 < \ldots < \mathbb{S}_n$ in the sense that $[\![p_k,k]\!] \in \mathbb{S}_k$. We present the recursive step at the heart of Clausen's algorithm in the form of a theorem.

**Theorem 1** *Let $f$ be a function $\mathbb{S}_n \to \mathbb{C}$ and let*

$$\widehat{f}(\rho_\lambda) = \sum_{\sigma\in\mathbb{S}_n} \rho_\lambda(\sigma)\, f(\sigma) \qquad \lambda \vdash n \tag{2.8}$$

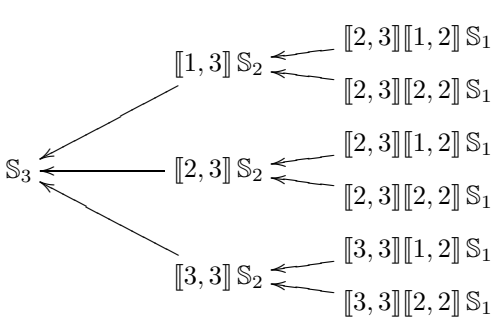
*be its Fourier transform with respect to Young's orthogonal representation. Now for $i = 1,2,\ldots,n$ define $f_i\colon \mathbb{S}_{n-1} \to \mathbb{C}$ as $f_i(\sigma') = f([\![i,n]\!]\sigma')$ for $\sigma' \in \mathbb{S}_{n-1}$, and let*

$$\widehat{f_i}(\rho_{\lambda^-}) = \sum_{\sigma'\in\mathbb{S}_{n-1}} \rho_{\lambda^-}(\sigma')\, f_i(\sigma') \qquad \lambda^- \vdash n-1 \tag{2.9}$$

*be the corresponding Fourier transforms, again with respect to Young's orthogonal representation. Then up to reordering of rows and columns,*

$$\widehat{f}(\rho_\lambda) = \sum_{i=1}^{n} \rho([\![i,n]\!]) \bigoplus_{\lambda^-\in R(\lambda)} \widehat{f_i}(\rho_{\lambda^-}), \tag{2.10}$$

*where $R(\lambda) = \{\, \lambda^- \vdash n-1 \mid \lambda^- \preceq \lambda \,\}$.*




```
function FFT(k, f) {
  for i ← 1 to k { f̂_i ← FFT(k − 1, f_i) ;}
  for each λ ⊢ k do {
    f̂(λ) ← 0_{d_λ × d_λ} ;
    for i ← 1 to k {
      f̂(λ) ← f̂(λ) + ρ_λ([[i, k]]) ⊕_{λ⁻ ∈ R(λ)} f̂_i(λ⁻) ;
    }
  }
  return f̂;
}
```

Figure 2.1: Clausen's FFT splits the Fourier transform into a combination of smaller Fourier tranforms over left cosets of $\mathbb{S}_n$, here illustrated for the case of $\mathbb{S}_3$. The pseudocode of the FFT is on the right.

**Proof.** Factoring $\sigma$ as $[\![i,n]\!]\,\sigma'$ as in (2.7) allows us to write (2.8) in the form

$$\widehat{f}(\rho_\lambda) = \sum_{i=1}^{n} \rho_\lambda([\![i,n]\!]) \sum_{\sigma' \in \mathbb{S}_{n-1}} \rho_\lambda(\sigma')\, f_i(\sigma')\,. \tag{2.11}$$

The inner summation is similar to (2.9), but the two are not the same, since $\rho_\lambda$ is in general not an irreducible of the smaller group $\mathbb{S}_{n-1}$. Rather, by standard results from representation theory, on restriction to $\mathbb{S}_{n-1}$, it splits into irreducibles of $\mathbb{S}_{n-1}$ in the form

$$T^{-1}\rho_\lambda(\sigma')\,T = \bigoplus_{\rho^-} \rho^-(\sigma') \qquad\qquad \sigma' \in \mathbb{S}_{n-1} \tag{2.12}$$

for some invertible square matrix $T$. The sum extends over some subset of $\mathcal{R}_{n-1}$, and in general may contain repeats. For computational purposes it is highly desirable to have $T = I$. Representations $\mathcal{R}_n$ and $\mathcal{R}_{n-1}$ achieving this are called **adapted representations**.

To prove that Young's orthogonal representations are adapted in this sense, consider the standard tableaux $\{t\}$ labeling the dimensions of $\rho_\lambda$. The number $n$ is always at an outer corner in these tableaux (Figure (2.2)). The critical observation is that the way we defined the action of transpositions (in general, the action of permutations) on tableaux, if $\sigma'$ is restricted to $\mathbb{S}_n$, then it leaves $n$ in the same box. Hence, $\rho_\lambda(\sigma')$ splits into blocks corresponding to which outer corner $n$ is located at in the corresponding standard tableaux. Careful examination of the definition of YOR reveals that each such block is, in fact, identical to $\rho_{\lambda^-}(\sigma')$, where $\lambda^- \vdash n-1$ is the partition that we get by removing the box containing $n$ from $\lambda$. This proves that YOR is an adapted representation; that for the symmetric group the multiplicity of each $\rho^-$ in (2.12) is just 1; and that $\rho^-$ runs over all irreducibles of $\mathbb{S}_n$ corresponding to those partitions that we get by removing one of the outer corners from $\lambda$. The latter are just $\{\rho_{\lambda^-}\}_{\lambda^- \in R(\lambda)}$.

Substituting $\rho_\lambda(\sigma') = \bigoplus_{\lambda^- \in R(\lambda)} \rho_{\lambda^-}(\sigma')$ in (2.11) and comparing with (2.9) gives exactly (2.11). ■

Clausen's FFT proceeds by recursively applying Theorem 1 down to $\mathbb{S}_1$. On $\mathbb{S}_1$ Fourier transformation is trivial, since $\mathcal{R}_1$ consists of the single irreducible $\rho_{(1)} = \rho_{\text{triv}}$, and hence $\widehat{f}(\rho) = f(e)$. In effect, the FFT employs a "divide and conquer" type strategy, dividing $\mathbb{S}_n$ into the $n$ cosets $[\![i_1,n]\!]\mathbb{S}_{n-1}$, which are in turn each divided into $n-1$ cosets $[\![i_1,n]\!][\![i_2,n-1]\!]\mathbb{S}_{n-2}$, etc.. (Figure 2.1, left). The pseudocode of the algorithm is presented in the same figure, on the right.

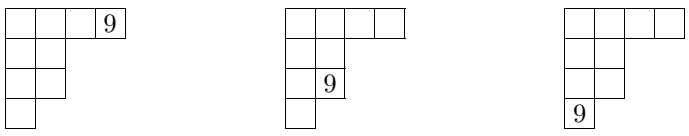



Figure 2.2: The possible positions of $n = 9$ in standard tableaux of shape $\lambda = (4,2,2,1)$.

While the algorithmic implementation of the FFT is depth first, for purposes of analysis we can also look at it as a sequence of transformations from a collection of FTs over $\mathbb{S}_{k-1}$ cosets to $\mathbb{S}_k$ cosets. Given that each of these transformations is a linear map $\mathfrak{F}_k \colon \mathbb{C}^{n!} \to \mathbb{C}^{n!}$, it is remarkable that the total number of scalar operations required by the FFT is just $\frac{(n+1)n(n-1)}{3} n!$. The algorithm owes its efficiency to the following factors:

1. As $k$ decreases, the various $\widehat{f}(\lambda)$ matrices rapidly decrease in size.
2. Thanks to the special properties of YOR, multiplying by $\rho_\lambda([\![i,k]\!])$ can be accomplished very fast.
3. There is considerable overlap between $R(\lambda)$ for different partitions, so we don't have to recompute a different set of $\{\widehat{f}(\lambda^-)\}_{\lambda^- \in \mathcal{R}_{k-1}}$ for each $\lambda \in \mathcal{R}_k$.

As explained in the proof of Theorem 1, $R(\lambda)$ are those partitions that we get by removing an outer corner from $\lambda$. This splitting behavior is represented by a so-called Bratelli diagram (Figure 2.3, left). A schematic of the entire FFT on $\mathbb{S}_n$ is presented in the same figure on the right. Such diagrams will be important for understanding the partial Fourier transforms of the next section.

The ideas behind Clausen's FFT are easily adapted to computing the inverse Fourier transform as well. The key observation is that each $\mathfrak{F}_i$ is itself a unitary transformation, hence the FFT can be reversed simply by applying the conjugate transpose transformation in each stage, and reversing the order of the stages. The pseudocode for the inverse FFT is the following.

```
function iFFT(k, f̂, σ){
  if k = 1 { f(σ) ← f̂; }
  else {
    for i ← 1 to k {
      for each λ⁻ ⊢ k − 1  do { f̂_i(λ⁻) = 0; }
      for each λ ⊢ k do {
        M ← ρ_λ([[i,k]]^{-1}) · f̂(λ);
        c ← 1;
        for each λ⁻ ∈ R(λ) do {
          f̂_i(λ⁻) ← f̂_i(λ⁻) + (d_λ / (k d_{λ⁻})) M(c : c + d_{λ⁻} − 1, c : c + d_{λ⁻} − 1);
          c ← c + d_λ;
      }}
      iFFT(k − 1, f̂_i, σ · [[i,k]]);
}}}
```

The inverse FFT appears to be more complicated than the forward FFT, but that is mostly due to the fact that decomposing $M$ into blocks of specified sizes is more difficult to notate than it is to notate assembling it from similar blocks. The notation $M(a:b,\ c:d)$ stands for the block of elements in $M$ from row $a$ to row $b$ and from column $c$ to column $d$ (inclusive). The factor $\frac{d_\lambda}{k\, d_{\lambda^-}}$ is required to compensate for the $d_\lambda/n!$ multipliers in the inner products in (2.3). Note that in YOR $\rho_\lambda([\![i,k]\!]^{-1}) = [\rho_\lambda([\![i,k]\!])]^\top$.

## 2.3 Two forms of sparsity

While the $O(n!n^3)$ run time of the FFT is much less than the run time of a naive Fourier Transform, for $n$ greater than about 12 it is still forbiddingly expensive. Considering that 12! is close to 500 million, even storing $f$ in memory becomes problematic in this range. One way to exploit the theory of harmonic analysis on $\mathbb{S}_n$ is to appeal to smoothness properties of $f$ and only maintain the first few components of $\widehat{f}$. As in the classical case, this will correspond to band-limited functions on $\mathbb{S}_n$ [Kueh et al., 1999]. Since now the components are not conveninetly labeled by real numbers, which of them are "low-frequency" components is a somewhat delicate question that we cannot discuss here in detail. We content ourselves by stating that in general, the smoothest components are $(n)$, $(n-1,1)$, $(n-2,2)$, $(n-2,1,1)$, etc..

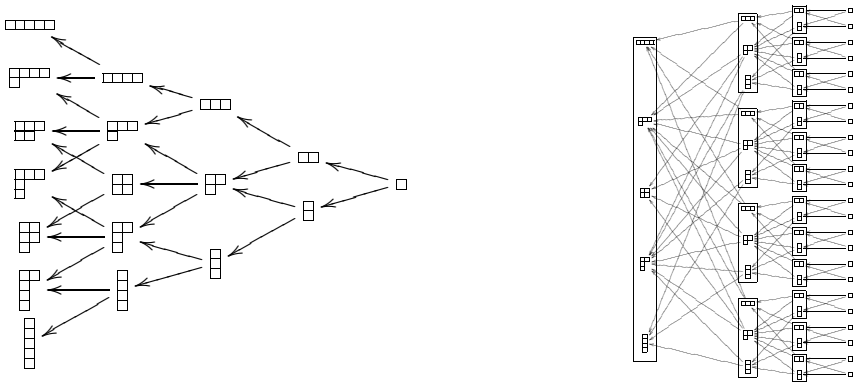


Figure 2.3: The Bratelli diagram for $\mathbb{S}_5$ and the information flow in the FFT on $\mathbb{S}_4$.

One of the ways in which $\mathbb{S}_n$ob extends Clausen's FFT is by implementing **fast partial Fourier transforms**, which compute $\widehat{f}$ only at a subset $P$ of the partitions $\{\lambda \vdash n\}$. This makes non-trivial computations possible on a regular desktop machine with $n$ well into the twenties.

Fast partial Fourier transforms are computed by restricting Clausen's algorithm to those paths in the Bratelli diagram that lead to the desired components (Figure 2.4, left). More explicitly, if $f$ is a $P$**-band-limited** function on $\mathbb{S}_n$, then at each level $k$ only $\{ f_{\lambda^-} \mid \lambda^- \vdash k, \;\; \lambda^- \preceq \lambda \;\text{ for some }\; \lambda \in P \}$ components are non-zero, so only these need to be computed. Otherwise, the FFT and its inverse are much the same as described in the previous section.

In practical applications sparsity in $\widehat{f}$ is likely to co-occur with sparsity in $f$. While the former restricts the algorithm to a subset of the paths in the Bratelli diagram, the latter restricts it to a subset of the paths connecting the various cosets (Figure (2.4), right). $\mathbb{S}_n$ob contains a special recursive data structure, `Sn::Ftree` to handle both types of sparsity.

In general, the philosophy behind $\mathbb{S}_n$ob is to regard the graph shown in Figure (2.3) as a network on which computations pass messages from one node to another. The `Sn::Ftree` data structure is used to construct the active subnetwork for a given combination of sparsity and band limits. Fourier transforms and other operations are then carried out on this structure, as opposed to the original network. Computations starting and finishing at internal nodes are also possible.

## 2.4 Double cosets and twisted transforms

Clausen's algorithm, as described in Section 2.2, hinges on splitting $\mathbb{S}_n$ into a union of cosets $[\![1,n]\!]\,\mathbb{S}_{n-1}$, $[\![2,n]\!]\,\mathbb{S}_{n-1}, \ldots, [\![n,n]\!]\,\mathbb{S}_{n-1}$, but this is not the only possible $\mathbb{S}_{n-1}$-coset partition of $\mathbb{S}_n$. We could equally

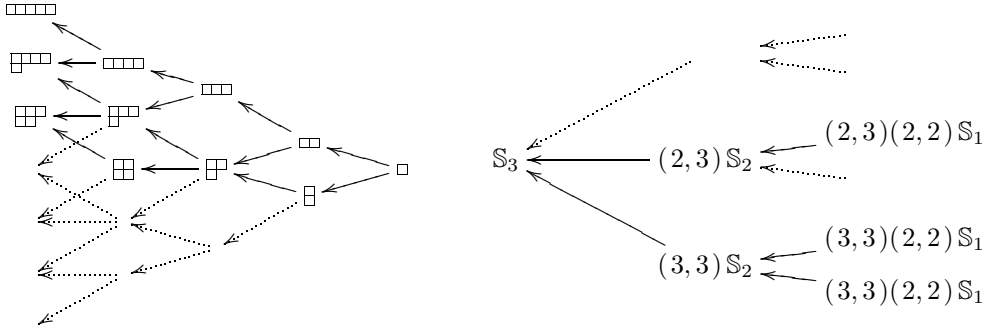



Figure 2.4: Two forms of sparsity: partial Fourier transforms restrict us to a subtree of the Bratelli diagram, while sparse functions restrict us to a subtree of the coset tree.

well use the right cosets $\mathbb{S}_{n-1}[\![1,n]\!]$, $\mathbb{S}_{n-1}[\![2,n]\!]$, …, $\mathbb{S}_{n-1}[\![n,n]\!]$, or, at a slight computational penalty, for fixed $j$, we could use the double cosets $[\![1,n]\!]\,\mathbb{S}_{n-1}[\![j,n]\!]$, $[\![2,n]\!]\,\mathbb{S}_{n-1}[\![j,n]\!]$, …, $[\![n,n]\!]\,\mathbb{S}_{n-1}[\![j,n]\!]$.

**Theorem 2** *Let $f$ be a function $\mathbb{S}_n \to \mathbb{C}$ and let*

$$\widehat{f}(\lambda) = \sum_{\sigma\in\mathbb{S}_n} \rho_\lambda(\sigma)\, f(\sigma) \qquad \lambda \vdash n \tag{2.13}$$

*be its Fourier transform with respect to Young's orthogonal representation. Let $j$ be a fixed integer $1 \le j \le n$. Now for $i = 1, 2, \ldots, n$ define $f_i^L, f_i^R \colon \mathbb{S}_{n-1} \to \mathbb{C}$ as $f_i^L(\sigma) = f([\![i,n]\!]\,\sigma\,[\![j,n]\!])$ and $f_i^R(\sigma) = f([\![j,n]\!]\,\sigma\,[\![i,n]\!])$. Let*

$$\widehat{f}_i^L(\lambda^-) = \sum_{\sigma'\in\mathbb{S}_{n-1}} \rho_{\lambda^-}(\sigma')\, f_i^L(\sigma') \qquad \text{and} \qquad \widehat{f}_i^R(\lambda^-) = \sum_{\sigma'\in\mathbb{S}_{n-1}} \rho_{\lambda^-}(\sigma')\, f_i^R(\sigma') \qquad \lambda \vdash n-1$$

*be the corresponding Fourier transforms, again with respect to Young's orthogonal representation. Then up to reordering of rows and columns,*

$$\widehat{f}(\lambda) = \left[ \sum_{i=1}^{n} \rho([\![i,n]\!]) \left[ \bigoplus_{\lambda^-\in R(\lambda)} \widehat{f}_i^L(\rho_{\lambda^-}) \right] \right] \cdot \rho([\![j,n]\!]), \tag{2.14}$$

*and*

$$\widehat{f}(\lambda) = \rho([\![j,n]\!]) \cdot \left[ \sum_{i=1}^{n} \left[ \bigoplus_{\lambda^-\in R(\lambda)} \widehat{f}_i^R(\rho_{\lambda^-}) \right] \rho([\![i,n]\!]) \right], \tag{2.15}$$

*where $R(\lambda) = \{\, \lambda^- \vdash n-1 \mid \lambda^- \preceq \lambda \,\}$.*

The proof of this theorem is very similar to the proof of Theorem 1. Just as before, (2.14) and (2.15) can be applied recursively down to $\mathbb{S}_1$ to yield a fast Fourier transform.

To distinguish the various cases, FFTs based on (2.14) and (2.15) we respectively call **left-hand** and **right-hand** FFTs. When $j = n$ (i.e., when $\rho([\![j,n]\!])$ disappears), we say that the FFT is **straight**, otherwise we say that it is **twisted**. When forming the full FFT, right-hand and left-hand steps may in priciple be mixed, and $j$ can be different at each level. $\mathbb{S}_n$ob supports all of these variations.

The purpose of these generalizations is to be able to compute Fourier transforms at various combinations of double cosets. In general, given $k \le n$, and two permutations $\sigma_{\mathrm{L}}, \sigma_{\mathrm{R}} \in \mathbb{S}_n$ with the property $\sigma_{\mathrm{L}}(i) = i$ and $\sigma_{\mathrm{R}}(i) = i$ for $1 \le i \le k$, and a (partial) Fourier transform $\widehat{f} \colon \mathbb{S}_n \to \mathbb{C}$, we compute the Fourier transform at $\sigma_{\mathrm{L}}\mathbb{S}_k\sigma_{\mathrm{R}}$,

$$\widehat{f}^{\sigma_{\mathrm{R}}}_{\sigma_{\mathrm{L}}}(\lambda) = \sum_{\tau\in\mathbb{S}_k} \rho_\lambda(\tau)\, f(\sigma_{\mathrm{L}}\tau\,\sigma_{\mathrm{R}}) \qquad \lambda \vdash k,$$

by factoring $\sigma_L$ and $\sigma_R$ as

$$\begin{aligned} \sigma_L &= [\![p_n,n]\!]\,[\![p_{n-1},n-1]\!]\ldots[\![p_{k+1},k+1]\!] \\ \sigma_R &= [\![q_{k+1},k+1]\!]\,[\![q_{k+2},k+2]\!]\ldots[\![q_n,n]\!] \end{aligned}$$

and descending from $\widehat{f}$ through a succession of double cosets

$$\begin{array}{c} \mathbb{S}_n, \\ [\![p_n,n]\!]\,\mathbb{S}_{n-1}\,[\![q_n,n]\!], \\ [\![p_n,n]\!]\,[\![p_{n-1},n-1]\!]\,\mathbb{S}_{n-2}\,[\![q_{n-1},n-1]\!][\![q_n,n]\!], \\ \vdots \qquad\quad \vdots \qquad\qquad\quad \vdots \qquad\quad \vdots \\ [\![p_n,n]\!]\,[\![p_{n-1},n-1]\!]\ldots[\![p_{k+1},k+1]\!]\,\mathbb{S}_k\,[\![q_{k+1},k+1]\!]\ldots[\![q_{n-1},n-1]\!][\![q_n,n]\!]. \end{array}$$

# Chapter 3

# Using $\mathbb{S}_n$ob

The source code of all the examples in this chapter may be found in the `examples` directory of the $\mathbb{S}_n$ob package. Wherever possible we show both the complete source and the output of the examples in the text.

## 3.1 Operations on group elements

In $\mathbb{S}_n$ob , permutations $\sigma \in \mathbb{S}_n$ are represented by objects of class `Sn::Element`. The constructor `Sn::Element(n)` returns the unit element of $\mathbb{S}_n$, while `Sn::Element(p1,p2,...,pn,NULL)` constructs $\sigma \in \mathbb{S}_n$ which would permute $(1, 2, \ldots, n)$ to $(p_1, p_2, \ldots, p_n)$, i.e., $\sigma^{-1}(i) = p_i$. The degree $n$ is determined automatically from the number of arguments to the constructor. Whenever a function takes a variable number of arguments in $\mathbb{S}_n$ob , the argument list must be terminated with `NULL`.

Our first example demonstrates multiplying permutations together and taking their inverses:

```
#include "SnElement.hpp"
#include <iostream>

main(){

  Sn::Element p1(1,2,4,3,5,NULL);
  Sn::Element p2(2,3,1,4,5,NULL);

  Sn::Element* a=p2*p1;
  Sn::Element* b=p2.inverse();
  Sn::Element* c=(*b)*p1;

  cout<<"p1           : "<<p1.str()<<endl;
  cout<<"p2           : "<<p2.str()<<endl;
  cout<<"p2*p1        : "<<a->str()<<endl;
  cout<<"p2^{-1}      : "<<b->str()<<endl;
  cout<<"(p2^{-1})*p1: "<<c->str()<<endl;

  delete a,b,c;
}
```

Note that the `*` operator and the `inverse()` method return pointers to new `Sn::Element` objects. This is the standard way of returning new objects in $\mathbb{S}_n$ob . The new objects are dynamically allocated, which means that unless they are explicitly deleted, they remain in memory until the program terminates. It is the responsibility of the user to keep track of dynamically allocated objects and delete them as soon as they are no longer needed. Given the large size of some of the data structures in typical $\mathbb{S}_n$ob applications, failure to delete unneeded objects can quickly lead to the system runing out of memory.

The str() function, which is common to all classes in $\mathbb{S}_n$ob, returns a short string identifying the object, similarly to toString in JAVA. For permutations str() returns the numbers $\sigma^{-1}(1), \sigma^{-1}(2), \ldots, \sigma^{-1}(n)$ enclosed in square brackets. Thus, the output of our code above is the following:

```
p1           : [ 1 2 4 3 5 ]
p2           : [ 2 3 1 4 5 ]
p2*p1        : [ 2 4 1 3 5 ]
p2^{-1}      : [ 3 1 2 4 5 ]
(p2^{-1})*p1: [ 4 1 2 3 5 ]
```

An easier to read way of specifying permutations is in terms of **cycle notation**. Section 2.2 described a way of uniquely factoring $\sigma$ into a product of contiguous cycles. In contrast, we now factor $\sigma$ into a product of disjoint but not necessarily contiguous cycles. In general, a cycle is a sequence $(c_1, c_2, \ldots, c_k)$ such that $\sigma(c_j) = c_{j+1}$ for $j = 1, 2, \ldots, k-1$, and $\sigma(c_k) = c_1$. Factorization into disjoint cycles is unique up to permuting the cycles between each other and cyclically permuting the numbers within each cycle. In $\mathbb{S}_n$ob this factorization is performed using the class Sn::Cycles.

```
#include "SnElement.hpp"
#include "Cycles.hpp"
#include <iostream>

main(){

  Sn::Element p1(1,2,4,3,5,NULL);
  Sn::Element p2(2,3,1,4,5,NULL);

  cout<<"v1 : "<<Cycles(p1).str()<<endl;
  cout<<"v2 : "<<Cycles(p2).str()<<endl;

}
```

```
v1 : (1)(2)(3,4)(5)
v2 : (1,3,2)(4)(5)
```

$\mathbb{S}_n$ob provides several methods to access the map $\sigma\colon \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$ itself. The member function action(i) returns $\sigma(i)$, while iaction(i) returns $\sigma^{-1}(i)$. Similarly, effect() returns an $n$-element C++-style vector $(\sigma(1), \sigma(2), \ldots, \sigma(n))$, while ieffect() returns $\left(\sigma^{-1}(1), \sigma^{-1}(2), \ldots, \sigma^{-1}(n)\right)$.

## 3.2 The group $\mathbb{S}_n$ and its irreducibles

To compute Fourier transforms, $\mathbb{S}_n$ob needs to construct the objects corresponding to the groups $\mathbb{S}_n > \mathbb{S}_{n-1} > \ldots > \mathbb{S}_1$ and each of their irreducibles. This is all done automatically when the user defines a single object of type `Sn::Sn` to represent the largest group $\mathbb{S}_n$. The resulting objects are typically kept in memory until the termination of the program.

The following program constructs $\mathbb{S}_3$ and prints out its elements.

```
#include "Sn.hpp"
#include "SnElement.hpp"
#include <iostream>

main(){

  Sn::Sn G(3);

  for(int i=0; i<G.order; i++)
    cout<<G[i]->str()<<endl;

}
```

```
[ 1 2 3 ]
[ 2 1 3 ]
[ 1 3 2 ]
[ 2 3 1 ]
[ 3 2 1 ]
[ 3 1 2 ]
```

The following program constructs $\mathbb{S}_5$ and goes through each of its irreducibles $\rho_\lambda \in \mathcal{R}_n$, printing out their identifiers. The identifier of $\rho_\lambda$ is just the partition $\lambda$, so the program lists the set $\{\lambda \vdash 5\}$.

```
#include "Sn.hpp"
#include "SnIrreducible.hpp"
#include <iostream>

main(){

  Sn::Sn G(5);

  for(int i=0; i<G.irreducibles.size(); i++)
    cout<<G.irreducibles[i]->str()<<endl;

}
```

```
(5)
(4,1)
(3,2)
(3,1,1)
(2,2,1)
(2,1,1,1)
(1,1,1,1,1)
```

The subgroups $\mathbb{S}_{n-1}, \ldots, \mathbb{S}_1$ are accessed by repeated reference to the `subgroup` variable of `Sn`. The following code finds the $\mathbb{S}_3$ object that was created when $\mathbb{S}_5$ was constructed, and prints out the identifiers of its irreducibles.

```
#include "Sn.hpp"
#include "SnIrreducible.hpp"
#include <iostream>

main(){

  Sn::Sn G(5);

  Sn::Sn& Gsub=*G.subgroup->subgroup;

  for(int i=0; i<Gsub.irreducibles.size(); i++)
    cout<<Gsub.irreducibles[i]->str()<<endl;

}
```

```
(3)
(2,1)
(1,1,1)
```

Each irreducible $\rho_\lambda$ of $\mathbb{S}_k$ has pointers to the irreducibles $\{\, \rho_{\lambda^-} \mid \lambda^- \in R(\lambda) \,\}$ into which it decomposes by Young's rule on restriction to $\mathbb{S}_{k-1}$ (see equation (2.12)). The following program, the output of which is too lengthy to reproduce here, uses this mechanism to print out a textual text version of the Bratelli diagram of $\mathbb{S}_5$ (Figure 2.3):

```
#include "Sn.hpp"
#include "SnIrreducible.hpp"
#include <iostream>

string printAncestors(Sn::Irreducible& rho, string indenter){
  ostringstream result;
  result<<indenter<<rho.str()<<endl;
  for(int i=0; i<rho.eta.size(); i++){
    result<<printAncestors(*rho.eta[i],indenter+"  ");
  }
  return result.str();
}


main(){

  Sn::Sn G(5);

  for(int i=0; i<G.irreducibles.size(); i++)
    cout<<printAncestors(*G.irreducibles[i],"");

}
```

Other features of `Sn::Irreducible` include functions for returning the degree of $\rho_\lambda$, the standard tableaux indexing its dimensions, the characters $\chi_\lambda(\mu)$ for $\mu \vdash \lambda$, and the representation matrices $\rho_\lambda(\sigma)$ for $\sigma \in \mathbb{S}_n$. The following example demonstrates all this for the $(3,2)$ irreducible of $\mathbb{S}_5$.

```
#include "SnIrreducible.hpp"
#include <iostream>

main(){

  Sn::Sn G(5);
  Sn::Irreducible& rho=*G.irreducibles[2];
  Partition lambda(2,2,1,NULL);
  Partition mu(3,1,1,NULL);
  Sn::Element sigma(2,3,1,4,5,NULL);

  cout<<"Irreducible: "<<rho.str()<<endl<<endl;
  cout<<"Degree: "<<rho.degree<<endl<<endl;
  cout<<"Tableaux: "<<endl<<endl;
  for(int i=0; i<rho.degree; i++)
    cout<<rho.tableau(i)->str()<<endl;
  cout<<"Character at mu: "<<rho.character(mu)<<endl<<endl;
  cout<<"Representation matrix at sigma: "<<endl<<endl<<rho.rho(sigma)->str()<<endl;

}
```

```
Irreducible: (3,2)

Degree: 5

Tableaux:

1 3 5
2 4

1 2 5
3 4

1 3 4
2 5

1 2 4
3 5

1 2 3
4 5

Character at mu: -1

Representation matrix at sigma:

-0.5 0.866025 0 0 0
-0.866025 -0.5 -0 -0 -0
0 0 -0.5 0.866025 0
-0 -0 -0.866025 -0.5 -0
0 0 0 0 1
```

Note that the representation matrices are given in Young's orthogonal representation, and the columns/rows are indexed according to the same order as the standard tableaux are ordered by the `tableau` function. The `rho` function returns a pointer to a `Matrix` object. `Matrix` is a custom $\mathbb{S}_n$ob matrix class supporting standard operations, such as addition, matrix mutliplication, etc..

An easy way to test the representation matrices is to verify the identity $\rho_\lambda(\sigma_2)\,\rho_\lambda(\sigma_1) = \rho_\lambda(\sigma_2\,\sigma_1)$ using the `*` operator of the `Matrix` class.

```
#include "SnIrreducible.hpp"
#include <iostream>

main(){

  Sn::Sn G(5);
  Sn::Irreducible& rho=*G.irreducibles[2];

  Sn::Element sigma1(1,2,4,3,5,NULL);
  Sn::Element sigma2(2,3,1,4,5,NULL);


  cout<<rho.rho(*(sigma2*sigma1))->str()<<endl;

  cout<<((*rho.rho(sigma2))*(*rho.rho(sigma1)))->str()<<endl;

}
```

```
0.5 0.866025 0 0 0
0.866025 -0.5 0 0 0
0 0 -0.5 0.288675 0.816497
-0 -0 -0.866025 -0.166667 -0.471405
0 0 0 0.942809 -0.333333

0.5 0.866025 0 0 0
0.866025 -0.5 0 0 0
0 0 -0.5 0.288675 0.816497
0 0 -0.866025 -0.166667 -0.471405
0 0 0 0.942809 -0.333333
```

Our final example in this section is the program `chartable.cpp`, which takes $n$ as an argument and prints out the character table of $\mathbb{S}_n$.

```
#include "SnIrreducible.hpp"

main(int argc, char** argv){

  int n=5;
  if(argc>=2) sscanf(argv[1],"%d",&n);

  Sn G(n);

  int Npartitions=G.irreducibles.size();
  Matrix<double> chartable(Npartitions,Npartitions);

  for(int i=0; i<Npartitions; i++)
    for(int j=0; j<Npartitions; j++)
      chartable(i,j)=round(G.irreducibles[i]->character(G.irreducibles[j]->partition));

  cout<<chartable.str()<<endl;

}
```

Applying `round` to the computed values eliminates round-off errors but does not corrupt the characters, since, thanks to yet another special property of $\mathbb{S}_n$, the characters are always integer valued. Below is the output of `chartable` for $n = 6$.

```
1 1 1 1 1 1 1 1 1 1 1
-1 0 -1 1 -1 0 2 -1 1 3 5
0 -1 1 -1 -0 0 0 3 1 3 9
1 -0 0 0 1 -1 1 -2 -2 2 10
0 0 -1 -1 2 1 -1 -3 1 1 5
0 1 -0 -0 -2 0 -2 0 0 0 16
-1 0 0 0 1 1 1 2 -2 -2 10
0 0 -1 1 2 -1 -1 3 1 -1 5
0 -1 1 1 -0 0 0 -3 1 -3 9
1 0 -1 -1 -1 0 2 1 1 -3 5
-1 1 1 -1 1 -1 1 -1 1 -1 1
```

## 3.3 Computing Fourier transforms

The principal devices for computing Fourier transforms in $\mathbb{S}_n$ob are the classes `Sn::Function` and `Sn::Fourier Transform`. The following program generates a random function $f$ on $\mathbb{S}_4$, computes its Fourier transform $F$, and then recovers $f' = f$ by the inverse transform. To save space we truncated the output, omitting the result of the inverse transform (the result is identical to to the original function).

```
#include "SnFunction.hpp"
#include "SnFourierTransform.hpp"
#include <iostream>

main(){

  Sn::Sn G(4);

  Sn::Function f(G);
  f.randomize();
  cout<<f.str()<<endl;

  Sn::FourierTransform* F=f.FFT();
  cout<<F->str()<<endl;

  Sn::Function* fdash=F->iFFT();
  cout<<fdash->str()<<endl;

}
```

```
[ 1 2 3 4 ] : 0.840188
[ 2 1 3 4 ] : 0.394383
[ 1 3 2 4 ] : 0.783099
[ 2 3 1 4 ] : 0.79844
[ 3 1 2 4 ] : 0.911647
[ 3 2 1 4 ] : 0.197551
[ 1 2 4 3 ] : 0.335223
[ 2 1 4 3 ] : 0.76823
[ 1 3 4 2 ] : 0.277775
[ 2 3 4 1 ] : 0.55397
[ 3 1 4 2 ] : 0.477397
[ 3 2 4 1 ] : 0.628871
[ 1 4 2 3 ] : 0.364784
[ 2 4 1 3 ] : 0.513401
[ 1 4 3 2 ] : 0.95223
[ 2 4 3 1 ] : 0.916195
[ 3 4 1 2 ] : 0.635712
[ 3 4 2 1 ] : 0.717297
[ 4 1 2 3 ] : 0.141603
[ 4 2 1 3 ] : 0.606969
[ 4 1 3 2 ] : 0.0163006
[ 4 2 3 1 ] : 0.242887
[ 4 3 1 2 ] : 0.137232
[ 4 3 2 1 ] : 0.804177

13.0156

-0.124596 0.43564 1.75584
0.327114 -0.0937104 0.0160957
-1.11595 -0.424087 0.895225

1.13475 -0.634588
0.931515 0.44088

0.859317 -0.613531 -1.20201
0.0108037 -0.415673 1.07824
-0.956903 0.665582 -0.495672

2.12302
```

The Fourier transform of a function with unit mass concentrated on a single permutation $\sigma$ reproduces the $\rho_\lambda(\sigma)$ representation matrices. This is demonstrated by the following code.

```
#include "SnFunction.hpp"
#include "SnFourierTransform.hpp"
#include "SnIrreducible.hpp"
#include <iostream>

main(){

  Sn::Sn G(5);

  Sn::Element sigma(4,1,3,5,2,NULL);

  Sn::Function f(G);
  f[sigma]=1;
  Sn::FourierTransform F(f);

  Sn::Irreducible& rho=*G.irreducibles[3];

  cout<<F.matrix[3]->str()<<endl;

  cout<<rho.rho(sigma)->str()<<endl;

}
```

```
-0.333 0.471 0.816 0 0 0
0.118 0.208 -0.0722 -0.484 -0.28 -0.791
-0.204 0.0722 -0.125 0.28 0.807 -0.456
0.456 0.807 -0.28 0.125 0.0722 0.204
-0.791 0.28 -0.484 -0.0722 -0.208 0.118
0 0 0 -0.816 0.471 0.333

-0.333 0.471 0.816 0 0 0
0.118 0.208 -0.0722 -0.484 -0.28 -0.791
-0.204 0.0722 -0.125 0.28 0.807 -0.456
0.456 0.807 -0.28 0.125 0.0722 0.204
-0.791 0.28 -0.484 -0.0722 -0.208 0.118
-0 -0 -0 -0.816 0.471 0.333
```

Yet another way to test the Fourier transform is via the convolution property. `Sn::Function` has a `convolve` member, which performs convolution using the convolution theorem (2.4). The following program computes $g * f$ and compares it to a traditional slow convolution on $\mathbb{S}_4$. To save space we omit the output.

```
#include "SnFunction.hpp"
#include "SnFourierTransform.hpp"
#include <iostream>

main(){

  Sn::Sn G(4);

  Sn::Function f(G);
  f.randomize();

  Sn::Function g(G);
  g.randomize();

  Sn::Function* h=g.convolve(f);
  cout<<h->str()<<endl;

  Sn::Function hdash(G);

  for(int i=0; i<G.order; i++)
    for(int j=0; j<G.order; j++){
      Sn::Element* x=G[j];
      Sn::Element* z=G[i];
      Sn::Element* xinv=x->inverse();
      Sn::Element* y=(*z)*(*xinv);
      hdash[*z]+=g[*y]*f[*x];
      delete x,xinv,y,z;
    }

  cout<<hdash.str()<<endl;
}
```

.

Finally, the `diffuse(beta)` member of `Sn::Function` implements diffusion smoothing on the Cayley graph of $\mathbb{S}_n$ generated by transpositions.

```
#include "SnFunction.hpp"
#include "SnFourierTransform.hpp"
#include <iostream>

main(){

  Sn::Sn G(4);

  Sn::Function f(G);
  f.randomize();
  cout<<f.str()<<endl;

  f.diffuse(0.1);
  cout<<f.str()<<endl;
}
```

## 3.4 Partial and sparse Fourier transforms

One of the most flexible features of $\mathbb{S}_n$ob is the Fourier tree data structure. A Fourier tree on $\mathbb{S}_n$ is a rooted tree with at most $n$ levels, labeled $n, n-1, n-2, \ldots$, in which each node at level $k$ corresponds to some double coset $\sigma_L \mathbb{S}_k \sigma_R$ of $\mathbb{S}_n$. In particular, the root corresponds to the entire group, while leaves at level 1 (if there are any) stand for individual group elements.

Which double coset a particular node corresponds to is determined by the labels on the edges. Specifically, each edge between levels $k$ and $k-1$ is labeled with a pair of contiguous cycles $[\![i,k]\!]$ and $[\![j,k]\!]$. The semantics is that if the level $k$ parent node corresponds to $\sigma_L \mathbb{S}_k \sigma_R$, then the level $k-1$ child note will stand for the double coset $\sigma_L [\![i,k]\!]^{-1} \mathbb{S}_{k-1} [\![j,k]\!]^{-1} \sigma_R$.

The nodes of the tree are objects of class `Sn::Ftree`. Each node at level $k$ can store the Fourier transform of a function on the corrsponding $\mathbb{S}_k$ double coset. Computations are performed on the tree by passing messages along the edges and updating the Fourier transforms attached to the nodes.

A function $f$ on the entire group $\mathbb{S}_n$ would be represented by constructing a complete tree with $n!$ nodes. A possible choice for the edge labels is to always take $j = k$, while $i$ takes on all possible values $1, 2, \ldots, k$. The function values $f(\sigma)$ are then stored as trivial Fourier transforms at the leaves. By propagating messages up the tree, Fourier transforms are computed at intermediate nodes, until finally the global Fourier transform appears at the root. This is just a message passing implementation of Clausen's algorithm described in Section 2.2 (left-hand version). It is apparent that we could equally well have taken $i \equiv k$ and split the tree at each level according to $\sigma_R = [\![i,k]\!] \in \mathbb{S} \backslash \mathbb{S}_{k-1}$, which would have lead to the right-hand FFT.

The novelty in the Fourier tree data structure is its ability to represent sparse functions on $\mathbb{S}_n$ by selectively including only those branches in the tree that lead to non-zero leaves. The Fourier transform is computed by the same message passing scheme as before, but thanks to sparsity, the resulting algorithm is potentially much more efficient than a complete FFT would be.

Sparsity is often coupled with partial Fourier transforms. `Sn::Ftree` handles this by maintaining a vector `Iindex` listing the indices of the "active" Fourier components. Inactive components are not stored and not updated in the message passing sweeps.

The following example constructs a function on $\mathbb{S}_4$ which is non-zero only at $\sigma_1$ and $\sigma_2$, and compares taking its Fourier transform with the Fourier tree data structure exploiting its sparsity to taking an ordinary Fourier transform.

```
#include "SnFtree.hpp"

main(){

  Sn::Sn G(4);
  Sn::Function f(G);
  f[Sn::Element(1,2,4,3,NULL)]=3;
  f[Sn::Element(2,3,1,4,NULL)]=7;

  cout<<"The dense Fourier transform:"<<endl;
  cout<<Sn::FourierTransform(f).str()<<endl;

  Sn::Ftree fsparse(f);
  for(int i=0; i<G.irreducibles.size(); i++)
    fsparse.Iindex.push_back(i);

  fsparse.FFT();
  cout<<"The sparse Fourier transform:"<<endl;
  cout<<fsparse.str()<<endl;

}
```

```
The dense Fourier transform:
10

-0.5 6.06 0
-6.06 -2.5 2.83
0 2.83 6

-6.5 6.06
-6.06 -0.5

8 2.83 0
2.83 -4.5 6.06
0 -6.06 -6.5

4


The sparse Fourier transform:
(4)
10

(3,1)
-0.5 6.06 0
-6.06 -2.5 2.83
0 2.83 6

(2,2)
-6.5 6.06
-6.06 -0.5

(2,1,1)
8 2.83 0
2.83 -4.5 6.06
0 -6.06 -6.5

(1,1,1,1)
4
```

To take partial Fourier transforms, all we need to do, is to restrict `Iindex` to the indices of those components that we are interested in. The following example computes only the first $m$ Fourier components of a sparse function on $\mathbb{S}_5$. It then performs an inverse transform. Unless $m$ is set to the total number of irreducibles of $\mathbb{S}_5$, which is 7, the reconstruction of the original function will only be approximate. The output of the code is shown for $m = 5$.

```
#include "SnFtree.hpp"

main(int argc, char** argv){

  int m=5;
  if (argc>=2) sscanf(argv[1],"%d",&m);

  Sn::Sn G(5);
  Sn::Function f(G);
  f[Sn::Element(1,2,3,4,5,NULL)]=9;
  f[Sn::Element(1,2,4,3,5,NULL)]=3;
  f[Sn::Element(2,3,1,4,5,NULL)]=7;

  Sn::Ftree fsparse(f);
  for(int i=0; i<G.irreducibles.size() && i<m; i++)
    fsparse.Iindex.push_back(i);

  cout<<"The function:"<<endl<<endl;
  cout<<fsparse.str()<<endl;

  fsparse.FFT();
  cout<<"The sparse partial Fourier transform:"<<endl<<endl;
  cout<<fsparse.str()<<endl;

  fsparse.iFFT();
  cout<<"The result of the inverse transform:"<<endl<<endl;
  cout<<fsparse.str()<<endl;

}
```

```
The function:

[ 1 2 4 3 5 ] : 3
[ 2 3 1 4 5 ] : 7
[ 1 2 3 4 5 ] : 9

The sparse partial Fourier transform:

(5)
19

(4,1)
8.5 6.06 0 0
-6.06 6.5 2.83 0
0 2.83 15 0
0 0 0 19

(3,2)
2.5 6.06 0 0 0
-6.06 8.5 0 0 0
0 0 8.5 6.06 0
0 0 -6.06 6.5 2.83
0 0 0 2.83 15

(3,1,1)
17 2.83 0 0 0 0
2.83 4.5 6.06 0 0 0
0 -6.06 2.5 0 0 0
0 0 0 8.5 6.06 0
0 0 0 -6.06 6.5 2.83
0 0 0 0 2.83 15

(2,2,1)
17 2.83 0 0 0
2.83 4.5 6.06 0 0
0 -6.06 2.5 0 0
0 0 0 2.5 6.06
0 0 0 -6.06 8.5


The result of the inverse transform:

[ 1 2 4 3 5 ] : 3.30833
[ 2 3 1 4 5 ] : 5.65833
[ 1 2 3 4 5 ] : 7.65833
```

For more information about the `Sn::Ftree` class, see the Reference section.

# Chapter 4

# Reference

This chapter contains a brief description of the $\mathbb{S}_n$`ob` classes and their most important public member functions and member variables. Copy constructors, destructors, and private members are not listed. Whether members are `const` or not is generally not indicated.

Optional arguments are enclosed in `[ ]`. A variable number of paramaters is indicated by `...`. Note that in $\mathbb{S}_n$`ob` such a sequence of arguments must always be terminated with `NULL`.

$\mathbb{S}_n$`ob` uses inheritance, for example, `Sn` inherits some members from the more general class `FiniteGroup`. However, the current version of $\mathbb{S}_n$`ob` being focused solely on the symmetric group, this mechanism has little operational significance. To simplify the reference section, we therefore list inherited members under the description of the child class. The parent classes are not listed separately in the reference.

$\mathbb{S}_n$`ob` also uses nesting. In particular, the `Element`, `Irreducible`, `Function` and

## The header file `Sn.h`

The global header file `Sn.h` defines the following constants.

| Constant | Default | Description |
|---|---|---|
| `FIELD` | `double` | The base field $\mathbb{F}$ for the irreducible representations. The most general choice would be `complex`, but due to the special properties of $\mathbb{S}_n$, it is possible to limit $\rho \in \mathcal{R}_n$ to real representations. In particular, YOR is a real representation. In this case, `FIELD` can be chosen to be `double`. In fact, the current version of $\mathbb{S}_n$`ob` has only been tested with this setting. |
| `STR_PRECISION` | `int` | The number of significant digits printed by the various `str` functions. |

# The Sn classes

# Sn::Sn

This class represents the symmetric group $\mathbb{S}_n$. The object `Sn::Sn(n)` must be constructed before accessing any irreducible representations of $\mathbb{S}_n$, or constructing functions or Fourier transforms on $\mathbb{S}_n$.

**Parent class:** `FiniteGroup`

## CONSTRUCTORS

`Sn(const int n)`
: The symmetric group $\mathbb{S}_n$. Automatically constructs $\mathbb{S}_{n-1}, \mathbb{S}_{n-2}, \ldots$ down to $\mathbb{S}_1$, together with their irreducibles.

## MEMBER FUNCTIONS

`Element* operator[](const int i)`
: Returns the group element with index $i$, where $0 \leq i \leq n!-1$. The indexing scheme is specially adapted to the left-hand FFT described in Section 2.2 and is identical to that used in `Sn::Function`.

`Irreducible*(const Partition& lambda, int& index)`
: Returns a pointer to the `Sn::Irreducible` of shape `lambda`, and returns its index in `index`.

`string str()`
: Identifies the group by its degree.

## MEMBER VARIABLES

`const int n`
: Degree of group.

`Sn* subgroup`
: Pointer to the object representing $\mathbb{S}_{n-1}$ (for $n > 1$).

`vector<Irreducible*> irreducibles`
: Irreducible representations of $\mathbb{S}_n$.

# Sn::Element

Represents a group element $\sigma \in \mathbb{S}_n$, i.e., a permutation $\sigma\colon \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$.

**Parent class:** `FiniteGroup::Element`

## CONSTRUCTORS

`Element(int n)`
The identity element of $\mathbb{S}_n$.

`Element(const Sn& G)`
The identity element of the group `G`.

`Element(int p1, int p2, ... , NULL)`
The permutation $\sigma$ for which $\sigma^{-1}(i)$ is given by the $i$'th integer argument; $n$ is determined by the total number of arguments.

`Element(int n, int* v)`
The permutation $\sigma \in \mathbb{S}_n$ for which $\sigma^{-1}(i) =$ `v[i-1]`. The pointer `v` must point to an $n$-element C-style array.

## MEMBER FUNCTIONS

`Element* operator*(Sn::Element& tau)`
Returns a new permutation object corresponding to $\sigma\tau$.

`Element* inverse()`
Returns a new permutation object corresponding to $\sigma^{-1}$.

`int action(const int i)`
Returns $\sigma(i)$.

`int action(const int i)`
Returns $\sigma^{-1}(i)$.

`vector<int> effect()`
Returns the vector $(\sigma(1), \sigma(2), \ldots, \sigma(n))$.

`vector<int> ieffect()`
Returns the vector $(\sigma^{-1}(1), \sigma^{-1}(2), \ldots, \sigma^{-1}(n))$.

`string str()`
Prints $(\sigma^{-1}(1), \sigma^{-1}(2), \ldots, \sigma^{-1}(n))$ to string.

## MEMBER VARIABLES

`const int n`
n

# Sn::Irreducible

Represents an irreducible representation $\rho_\lambda$ of $\mathbb{S}_n$.

**Parent class:** `FiniteGroup::Irreducible`

## CONSTRUCTORS

`Irreducible(Sn* G, Partition& lambda)`
: Construct the irreducible representation of the symmetric group `G` corresponding to the partition `lambda`.

## MEMBER FUNCTIONS

`Matrix<FIELD >* rho(const Sn::Element& sigma)`
: Returns $\rho(\sigma)$, the representation matrix of permutation `sigma` in Young's orthogonal representation.

`FIELD character(const Partition& mu)`
: Returns $\chi(\mu)$, the character of this representation at permutations of cycle type $\mu$.

`void computeTableaux()`
: Compute the standard tableaux of this irreducible if they have not already been computed. Because this is an expensive operation, it is postponed until some function is called (such as `rho` or `character`) which requires the tableaux of this particular irreducible. `computeTableaux()` is called automatically by these functions, and once the tableaux have been computed they are stored for the lifetime of the `Irreducible`.

`StandardTableau* tableau(const int t)`
: Return a new standard tableaux of index `t`. This works even if `tableauV` has not been computed.

`void computeYOR()`
: Compute and store the coefficients (2.5) and (2.6) in Young's orthogonal representation for all adjacent transpositions $\tau_k$ and all tableau $t$ of shape $\lambda$. Because this is an expensive operation, these coefficients are not normally computed until they are demanded by functions such as `rho` or `character`. `computeYOR()` is called automatically by these functions, and once the tableaux have been computed they are stored for the lifetime of the `Irreducible`. `computeYOR()` also requires the tableaux, so it calls `computeTableaux()` if those have not been computed yet.

`void applyCycle(const int j, Matrix<FIELD>& M[, int m])`
`void applyCycle(const int j, Matrix<FIELD>& M[, int m])`
: Multiply `M` by the representation matrix of the cycle $(m, j, j+1, \ldots, m-1)$. These are specialized, fast, compute-in-place functions, and they are at the heart of the FFT and the iFFT. The default value of $m$ is $n$.

`string str()`
: Identifies the irreducible by returning the partition $\lambda$.

## MEMBER VARIABLES

`Partition partition`
: The partition indexing this irreducible.

`int degree`
Dimensionality of this irreducible.

`vector<int> etaindex`
The integer indices of those irreducibles of $\mathbb{S}_{n-1}$, into which this irreducible decomposes when restricted to $\mathbb{S}_{n-1}$.

`vector<Sn::Irreducible*> eta`
Pointers to irreducibles of $\mathbb{S}_{n-1}$ into which this irreducible decomposes when restricted to $\mathbb{S}_{n-1}$.

`bool tableauxComputed`
Flag to show that the standard tableaux have been computed.

`bool YORComputed`
Flag to show that the YOR coefficients have been computed.

`vector<StandardTableau> tableauV`
The standard tableaux of this representation.

# Sn::Function

Represents a function $f\colon \mathbb{S}_n \to \mathbb{F}$.

**Parent class:** `FiniteGroup::Function`

## CONSTRUCTORS

`Function(const Sn& group)`
Construct a function on the symmetric group `group` and set $f(\sigma) = 0$ for all $\sigma \in \mathbb{S}_n$.

`Function(const FourierTransform& F)`
Compute the inverse Fourier Transform of $F$ with Clausen's iFFT.

## MEMBER FUNCTIONS

`FIELD& operator[](const Element& sigma)`
Return reference to $f(\sigma)$.

`void randomize()`
Fill $(f(\sigma))_{\sigma \in \mathbb{S}_n}$ with random numbers uniformly distributed on the interval $[0, 1)$.

`FourierTransform* FFT()`
Return a new `Sn::FourierTransform` which holds the Fourier transform of $f$. Alternative (syntactic sugar) to computing the FFT with the constructor `FourierTransform::FourierTransform(const Function& f)`.

`Function* convolve(Function& g)`
Return a new function $h = f * g$ computed by FFT and the convolution theorem.

`void diffuse(const double beta)`
Diffuse $f$ with parameter $\beta$ on the Cayley graph generated by transpositions

`string str()`
Print $f(\sigma)$ for each $\sigma \in \mathbb{S}_n$.

## MEMBER VARIABLES

`const Sn* group`
Pointer to $\mathbb{S}_n$

# Sn::FourierTransform

Represents the Fourier transform $\widehat{f}$ of a function $f\colon \mathbb{S}_n \to \mathbb{F}$.

**Parent class:** `FiniteGroup::FourierTransform`

## CONSTRUCTORS

`FourierTransform(const Sn& group)`
: Construct the Fourier transform of the zero function on the symmetric group `group`.

`FourierTransform(const Sn& group, const vector<Matrix<FIELD>*> matrices)`
: Construct a `FourierTransform` with components `matrices`. The components must be listed in the same order as the `Irreducible`s in `group`, and they must have the right sizes. Warning: the matrices are not copied, only their addresses are duplicated in the new `FourierTransform` object. This matrices are deleted when the `FourierTransform` object is destroyed. Attempting to delete them elswhere in the code is liable to cause a segmentation fault.

`FourierTransform(const Function& f)`
: Compute the Fourier transform of the function $f$ with Clausen's FFT.

## MEMBER FUNCTIONS

`Function* iFFT()`
: Return a new `Sn::Function` which holds the inverse Fourier transform of $F$. Alternative (syntactic sugar) to computing iFFT with the constructor `Function::Function(const FourierTransform& F)`.

`double norm2()`
: Squared norm of $\widehat{f}$, as in (2.3).

`string str()`
: Print each of the $\widehat{f}(\lambda)$ matrices to string.

## MEMBER VARIABLES

`vector<Matrix<FIELD>*> matrix`
: The matrices $\widehat{f}(\rho)$.

# Sn::Ftree

SnFtree objects represent individual nodes of the Fourier tree data structure described in Section 3.4. Each intermediate node is at once a node in a larger tree and the root of its own tree consisiting of its descendants. Thus, depending on the context, it either represents a specific $\sigma_L \mathbb{S}_k \sigma_R$ double coset in a larger group $\mathbb{S}_n$, or the group $\mathbb{S}_k$ itself. Each node can hold an $\mathbb{S}_k$ (partial) Fourier transform in its matrix member. Which $\widehat{f}(\lambda)$ components are maintained is determined by the Iindex member, which lists the indices of the corresponding irreducibles of $\mathbb{S}_k$.

**Parent class:** FiniteGroup::Ftree

## CONSTRUCTORS

Ftree(const Sn& group, [const vector<int>& Iindex,] [int left, [int right] ] )
: If group is $\mathbb{S}_n$, then this constructs an $\mathbb{S}_n$-coset node. Iindex lists the Fourier components active at this node. If this node is descended from an $\mathbb{S}_{n+1}$ double coset node, then it will stand for the $[\![\texttt{left}, n+1]\!]\, \mathbb{S}_n\, [\![\texttt{right}, n+1]\!]$ coset in $\mathbb{S}_{n+1}$.

Ftree(const Function& f)
: Construct a sparse representation of $f$ by only growing branches that lead to leaves for which $f(\sigma)$ is non-zero.

Ftree(const FourierTransform& F, int l1, int l2,...,NULL)
: Construct an Ftree node which holds the restriction of F to the l1, l2,... components of $F$. Caveat: if component 0 is to be in the list, it must be listed first, otherwise it will be confused with the end-of-list NULL marker.

Ftree(const FourierTransform& F, const vector<int>& Iindex)
: Construct an Ftree node which holds the restriction of F to the components in Iindex.

## MEMBER FUNCTIONS

FFT()
: Compute the (partial) FFT by deleting the content of interior nodes and propagating the values at the leaves to the root. Which components ae computed is determined by Iindex. All nodes will be left empty except the root.

iFFT()
: Compute the inverse (partial) Fourier transform by distributing the content of the root to the leaves. All nodes will be left empty except for the leaves.

FourierTransform* fourierTransform()
: A new FourierTransform object which is the "full version" of this node. Whenever a component is not present, it will be substitued with a zero matrix.

SnFunction* function()
: A new SnFunction object holding the dense representation of the information at the leaves.

scout([bool zerobottom])
: Descend the tree and then climb back up again setting each Iindex to the minimal set of active Fourier components at each intermediate node required to compute the active components at this node from the active components at the leaves, or vica versa. When zerobottom is set, the active components at

the leaves are set to the maximal set of non-zero components computable from the components at the root. This function is usually called before `collect()` or `distribute()`.

`unscout()`
Delete the `Iindex` at each node between this node and the leaves (each internal node).

`collect()`
Collect $\widehat{f}$ from the descendants of this node. This is a recursive function used to compute fast (partial and sparse) Fourier transforms on the Fourier tree data structure. At each node, only components listed in that node's `Iindex` will be computed.

`distribute()`
Distribute $\widehat{f}$ to the descendants of this node. This is a recursive function used to compute fast (partial and sparse) inverse Fourier transforms on the Fourier tree data structure. At each node, only components listed in that node's `Iindex` will be computed.

`norm2()`
The squared $L_2$ norm of the partial Fourier transform at this node.

`Element* max(double& val)`
Returns a pointer to the element of $\mathbb{S}_n$ maximizing $f(\sigma)$ and sets `val` to be the maximum value.

`str()`
Print the non-zero elements of $f$ to string.

## MEMBER VARIABLES

`int n`
n

`const Sn* group`
pointer to $\mathbb{S}_n$

`int left,right`
The left and right labels of the edge leading from its parent to this node.

`bool protect`
For efficient memory management, by default `distribute()` only leaves data at the leaves and `collect()` only leaves data at the root. At all other nodes the Fourier matrices are deleted and the vector `matrix` is left empty. However, when `protect` is set, the matrices at this node will be preserved.

`bool addto`
By default, `collect()` and `distribute()` pass through the Fourier tree, they delete pre-existing matrices at intermediate nodes (except respectively the leaves and the root). When `addto` is set, they will instead add the results of the computations to what is already stored at this node.

`vector<Ftree*> child`
Pointers to each child.

`vector<int> Iindex`
Indices of the Fourier components active at this node.

`vector<Matrix<FIELD >*> matrix`
The $\widehat{f}(\lambda)$ Fourier components.

# Helper classes

## Cycles

An object for storing the cycle representation of $\sigma \in \mathbb{S}_n$.

**Parent class:** `vector<vector<int>>`

### CONSTRUCTORS

`Cycles(Element& sigma)`
: Compute the cycles making up $\sigma$.

### MEMBER FUNCTIONS

`size()`
: The number of cycles.

`str()`
: Print the cycles to string.

# Partition

This class represents integer partitions $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_k)$ of $n$. It is derived from the STL class `vector<int>`, hence all operations applicable to vectors of integers are applicable to `Partition` as well.

**Parent class:** `vector<int>`

## CONSTRUCTORS

`Partition()`
The empty partition $\lambda = ()$.

`Partition(int p1, int p2, ..., NULL)`
The partition $\lambda = (p_1, p_2, \ldots, p_k)$

`Partition(const Sn::Element& p)`
The cycle type of the permutation `sigma`.

## MEMBER FUNCTIONS

`n()`
Compute $n = \sum_i \lambda_i$.

`vector<Partition> restrictions`
The partitions $R(\lambda) = \{\, \lambda^- \vdash n-1 \mid \lambda^- \prec \lambda \,\}$.

`str()`
Print partition to string.

# StandardTableau

Represents a standard Young tableau $t$ of shape $\lambda$. This class is descended from `vector<vector<int>>`, hence all operations applicable to vector of this class are also applicable to `StandardTableau`.

**Parent class:** `vector<vector<int>>`

## CONSTRUCTORS

`StandardTableau()`
: The empty tableau.

`StandardTableau(const Partition& lambda)`
: The "first" standard tableau of shape `lambda`, i.e., the one in which the numbers increase sequentially from left to right and then continue in the next row.

`StandardTableau(int y1, int y2, ..., NULL)`
: The standard tableau with Yamanouchi symbol $(y_1, y_2, \ldots, y_k)$, i.e., the tableau in which 1 is in row $y_1$, 2 is in row $y_2$, e.t.c..

## MEMBER FUNCTIONS

`str()`
: Print tableau to string.

# Matrix<FIELD>

A matrix $M$ with elements of class `FIELD`. `FIELD` is set in `Sn.h`, typically either to `double` or `complex`. At this time, $\mathbb{S}_n$ob has only been tested with the former.

## CONSTRUCTORS

`Matrix<FIELD>(int n)`
Initialize $M$ to be the $n$-dimensional identity matrix.

`Matrix<FIELD>(int n, int m)`
An $n \times m$ matrix with unitialized elements.

`Matrix<FIELD>(int n, int m, FIELD a)`
Initialize $M$ to be an $n \times m$ matrix with elements $[M]_{i,j} = a$.

`Matrix<FIELD>(string filename)`
Load $M$ from an ASCII file.

## MEMBER FUNCTIONS

`FIELD& operator()(int i, int j)`
`FIELD& at(int i, int j)`
Return reference to $[M]_{i,j}$. (There is no difference between the two functions.)

`Matrix<FIELD>* operator+(Matrix<FIELD>& P)`
`Matrix<FIELD>* operator*(Matrix<FIELD>& P)`
Return a pointer to a new matrix respectively equal to the sum $M+P$ and the matrix product $MP$.

`Matrix<FIELD>& operator+=(Matrix<FIELD>& P)`
Increment $M$ by $P$ and return a reference to it.

`Matrix<FIELD>& operator*=(FIELD& a)`
Multiply $M$ by $a$ and return a reference to the result.

`FIELD trace()`
Return the trace of $M$.

`FIELD norm2()`
Return the squared norm of $M$.

`int save(string filename)`
Save $M$ in ASCII format.

`str()`
Print matrix to string.

## MEMBER VARIABLES

`int n`
: The row dimension of $M$.

`int m`
: The column dimension of $M$.

`FIELD* array`
: Pointer to a an $nm$-element array storing the elements of $M$ in row major order.

# Bibliography

Michael Clausen. Fast generalized Fourier transforms. *Theoretical Computer Science*, 67:55–63, 1989.

K.-L. Kueh, T. Olson, D. Rockmore, and K.-S. Tan. Nonlinear approximation theory on finite groups. Technical Report PMA-TR99-191, Department of Mathematics, Dartmouth College, 1999.

D. K. Maslen and D. N. Rockmore. Generalized FFTs — a survey of some recent results. In *Groups and Computation II*, volume 28 of *DIMACS Ser. Discrete MAth. Theor. Comput. Sci.*, pages 183–287. AMS, Providence, RI, 1997.