SnFFT Documentation

Release 0.0.1

Gregory Plumb

May 23, 2017

Contents

The *SnFFT* package is Julia package designed to facilitate harmonic analysis on the symmetric group of order n, denoted $\mathbf{S}_n$. Out of the box, *SnFFT* implements:

- Group operations and factorizations for $\mathbf{S}_n$
- Functionality to set up functions over $\mathbf{S}_n$
- The fast Fourier transform with additional options if the function is sparse or bandlimited
- The inverse fast Fourier transform with additional options if the function is bandlimited or the user is only interested in the result from the top few components
- The convolution and correlation of two Fourier transforms

Contents:

CHAPTER 1

Getting Started

Installation

Method 1

Download the setup script InitialSetup.jl. Then open a Julia session and run:

```
julia> require("InitialSetUp.jl")
```

This script will install and load the SnFFT library and then run the examples.

Method 2

SnFFT can also be installed using Julia's pakage manager as follows:

```
julia> Pkg.add("SnFFT")
```

Set Up

After the installation, the user can load the library with:

```
julia> using SnFFT
```

By default, lower level functions in the source code are not made available during installation. These functions are often wrapper or helper functions that may not be relevant to most users. However, these internal functions can be made available by modifying the file SnFFT.jl.

Parallelism

SnFFT allows the user to compute fast Fourier transforms and inverse fast Fourier transforms in parralel with no change to their code. On startup, simply run:

```
julia> addprocs(p)
julia> using SnFFT
```

This will and *p* worker processes and load the SnFFT library onto them. Afterwards, parts of *SnFFT* will automatically run in parallel, if their heuristics think say that it will be beneficial.

CHAPTER 2

Examples

SnFFT comes with eight example functions that demonstrate some of the key properties of Young's Orthogonal Representations and the Fourier Transform of functions over $\mathbf{S}_n$. Furthermore, they demonstrate the syntax used to call most of the functionality of the package. The examples are in the file Examples.jl.

General Notes:

- Most of the example functions have two methods. The first method has no parameters and will run the example with default values. The second has a full set of parameters and will be explained in each function's description.
- A parameter name will appear in **bold** when it being referenced in the example's description.
- *SnFFT* represents a partition in the following way:

```
# Let X be a Partition of N
#       X::Array{Int, 1}
#       X[i] > 0 for all i
#       sum(X) == N
#       X[i] >= X[j] when i < j
```

- *SnFFT* represents a permutation in the following way:

```
# Let X be a Permutation of N
#       X::Array{Int, 1}
#       length(X) == N
#       X[i] = j indicates that the item in position i is sent to position j
```

Example Function Explanations

example1 (*N*, *partition*, *permutation*)

```
# Parameters:
#       N::Int
#       - the problem size
#       partition::Array{Int, 1}
#       - a partition of N
#       permutation::Array{Int, 1}
#       - a permutation of N
```

This example finds the Standard Tableau corresponding to **partition**. It then calculates Young's Orthogonal Representation of **permutation** corresponding to **partition**.

example2 (*N*, *partition*, *p1*, *p2*)

```
# Parameters:
#       N::Int
#       - the problem size
#       partition::Array{Int, 1}
#       - a partition of N
#       p1::Array{Int, 1}
#       - the first permutation of N
#       p2::Array{Int, 1}
#       - the second permutation of N
```

Let YOR1 and YOR2 be Young's Orthogonal Representations of **p1** and **p2** corresponding to **partition**. Let YORm be Young's Orthogonal Representations of **p1** x **p2** corresponding to **partition**. This example demonstrates that YORm = YOR1 x YOR2.

example3 ()

This example demonstrates the order of the permutations used by *SnFFT* to represent a function over $\mathbf{S}_n$. It has no parameterized version.

example4 (*N*)

```
# Parameters:
#       N::Int
#       - the problem size
```

This example constructs a random function over $\mathbf{S}_n$. It then demonstrates how to calculate the (dense) fast Fourier transform and the inverse fast Fourier transform. Finally, it shows that this process recovers the original function accurately.

example5 (*N*, *SC*)

```
# Parameters:
#       N::Int
#       - the problem size
#       SC::Float64
#       - the portion of the function that is zero-valued
```

This example constructs a random sparse function over $\mathbf{S}_n$ and converts it to the format used to compute the spare fast Fourier transform. Next, it demonstrates how to compute the sparse fast Fourier transform. Finally, it shows that this produces the same result as the dense fast Fourier transform.

example6 (*N*, *permutation*)

```
# Parameters:
#       N::Int
#       - the problem size
```

```
#       Permutation::Array{Int, 1}
#       - a permutation of N
```

This example constructs a delta function over $\mathbf{S_n}$ that is centered on **permutation**. It then calculates the sparse fast Fourier transform of this function. Finally, it demonstrates that this produces the same results as computing Young's Orthogonal Representation for each partition of **N** corresponding to **permutation**.

example7 (*N*, *K*)

```
# Parameters:
#       N::Int
#       - the problem size
#       K::Int
#       - the problem is homogenous at N-K
```

This example constructs a random bandlimited function over $\mathbf{S_n}$. To save space, the bandlimited function is compressed. It then constructs the equivalent non-compressed version. Next, it demonstrates how to compute the bandlimited fast Fourier transform using the compressed function and take the bandlimited inverse fast Fourier transform. Finally, it shows that the dense and bandlimited fast Fourier transforms produce the same result and that the inverse bandlimited fast Fourier transform recovers the original bandlimited function over $\mathbf{S_n}$.

example8 (*N*, *M*)

```
# Parameters:
#       N::Int
#       - the problem size
#       M::Int
#       - the number of the top frequencies of the Fourier transform to use
```

This function constructs a random function over $\mathbf{S_n}$ and then demonstrates how to compute the paritial inverse fast Fourier transform. It prints both the original and recovered function.

CHAPTER 3

Symmetric Group Functionality

SnFFT is designed to take the Fourier transform of functions over $\mathbf{S}_n$. Although not strictly necessary to do this, having the basic functionality of the group can make testing and development much easier. These functions are in the file Element.jl.

Group Operations

sn_multiply(*P1*, *P2*)

```
# Parameters:
#       P1::Array{Int, 1}
#       - the first permutation
#       P2::Array{Int, 1}
#       - the second permutation
# Return Values:
#       Prod::Array{Int, 1}
#       - the permutation that is P1 * P2
# Notes:
#       - P1 and P2 must be permutations of the same size
```

sn_inverse(*P*)

```
# Parameters:
#       P::Array{Int, 1}
#       - a permutation
# Return Values:
#       Inv::Array{Int, 1}
#       - the permutation that is the inverse of P
```

Group Element Constructors

sn_p (*N*)

```
# Parameters:
#       N::Int
#       - the size of the permutation
# Return Values:
#       P::Array{Int, 1}
#       - a random permutation of N
```

sn_cc (*N, LB, UB*)

```
# Parameters:
#       N::Int
#       - the size of the permutation
#       LB::Int
#       - the first position that is reassigned
#       UB::Int
#       - the last position that is reassigned
# Return Values:
#       CC::Array{Int, 1}
#       - the permutation of N that is the contiguous cycle [[LB, UB]]
#       - this is the permutation that sends LB to LB + 1, LB + 1 to LB + 2, ... , UB - 1 to UB, and UB to LB
# Notes
#       - 1 <= LB <= UB <= N
```

sn_cc (*N*)

```
# Parameters:
#       N::Int
#       - the size of the permutation
# Return Values:
#       CC::Array{Int, 1}
#       - a random contiguous cycle of N
```

sn_at (*N, K*)

```
# Parameters:
#       N::Int
#       - the size of the permutation
#       K::Int
#       - the position that is being reassigned
# Return Values:
#       AT::Array{Int, 1}
#       - the permutation of N that is the adjacent transposition (K, K+1)
#       - this is the permutation that sends K to K + 1 and K + 1 to K
# Notes:
#       - 1 <= K < N
```

sn_at (*N*)

```
# Parameters:
#       N::Int
#       - the size of the permutation
# Return Values:
```

```
#       AT::Array{Int, 1}
#       - a random adjacent transposition of N
```

sn_t (N, I, J)

```
# Parameters:
#       N::Int
#       - the size of the permutation
#       I::Int
#       - the first postition that is being reassigned
#       J::Int
#       - the second position that is being reassigned
# Return Values:
#       Tr::Array{Int, 1}
#       - the permutation of N that is the transposition (I, J)
#       - this is the permutation that sends I to J and J to I
# Notes:
#       - 1 <= I <= N
#       - 1 <= J <= N
```

sn_t (N)

```
# Parameters:
#       N::Int
#       - the size of the permutation
# Return Values:
#       Tr::Array{Int, 1}
#       - a random transposition of N
```

Factorizations and Related Operations on the Left-Coset Tree

permutation_ccf (P)

```
# Parameters:
#       P::Array{Int, 1}
#       - a permutation
# Return Values:
#       CCF::Array{Int, 1}
#       - the Contiguous Cycle Factoriztion of P
#       - P = product for i = 1:(N - 1) of sn_cc(N, CCF[i], N + 1 - i)
```

ccf_index (CCF)

```
# Parameters:
#       CCF::Array{Int, 1}
#       - a contiguous cycle factorization of some permutation
# Return Values:
#       Index::Int
#       - the unique index that the permutation corresponding to CCF maps to
```

permutation_index (P)

```
# Parameters:
#       P::Array{Int, 1}
#       - a permutation
```

```
# Return Values:
#       Index::Int
#       - the unique index that P maps to
```

index_ccf (*N*, *Index*)

```
# Parameters:
#       N::Int
#       - the size of the permutation that maps to Index
#       Index::Int
#       - the index of some permutation of N
# Return Values:
#       CCF::Array{Int, 1}
#       - the contiguous cycle factorization that corresponds to the permutation that␣
↪maps to Index
```

ccf_permutation (*N::Int*, *Index::Int*)

```
# Parameters:
#       CCF::Array{Int, 1}
#       - a contiguous cycle factorization of some permutation
# Return Values:
#       P::Array{Int, 1}
#       - the permutation that corresponds to CCF
```

index_permutation (*N*, *Index*)

```
# Parameters:
#       N::Int
#       - the size of the permutation that maps to Index
#       Index::Int
#       - the index of some permutation of N
# Return Values:
#       P::Array{Int, 1}
#       - the permutation of N that maps to Index
```

permutation_atf (*P*)

```
# Parameters:
#       P::Array{Int, 1}
#       - a permutation
# Return Values
#       ATF::Array{Int, 1}
#       - the adjacent transposition factorization of P
#       - P = product for i = 1:length(ATF) of sn_at(N, ATF[i])
```

Young's Orthogonal Representation of a Permutation

yor_permutation (*P*, *YORnp*)

```
# Parameters:
#       P::Array{Int, 1}
#       - a permutation
#       YORnp::Array{SparseMatrixCSC, 1}
#       - YORnp[i] is Young's Orthogonal Representation for the adjacent␣
↪transposition (i, i + 1) corresponding to the pth partition of n
```

```
# Return Values:
#       RM::Array{Float64, 2}
#       - Young's Orthogonal Representation of P corresponding to the pth partition
↪of n
```

CHAPTER 4

Functions over the Symmetric Group

SnFFT represents a function over $\mathbf{S}_\mathrm{n}$ as an array of Float64 values.

Because this representation doesn't explicitly store the permutation that corresponds to each value of the function, SnFFT has a set of standards that it uses to define the correspondence between indices of this array and the permutations.

These standards will be explained for each of the three types of fast Fourier transforms that SnFFT implements.

Dense Functions

A dense function over $\mathbf{S}_\mathrm{n}$ will have a length of n! because the dense fast Fourier transform doesn't rely on any prior knowledge of the function.

The dense fast Fourier transform assumes that the value stored at index *i* is the value associated with the permutation that permutation_index() maps to *i*.

See example3() for more details.

snf (*N*, *PA*, *VA*)

```
# Parameters:
#       N::Int
#       - the problem size
#       PA::Array{Array{Int, 1}, 1}
#       - PA[i] is a Permutation of N
#       VA::Array{Float64, 1}
#       - VA[i] is the Value associated with PA[i]
# Return Values:
#       SNF::Array{Float64, 1}
#       - SNF[i] is the value associated with the permutation that permutation_
↪index() maps to i
#       - this is the format for the SNF parameter of sn_fft()
# Notes:
#       - any permutation of N not represented in PA will be assigned a value of zero
```

Bandlimited Functions

A bandlimited function over $\mathbf{S}_n$ that is invariant at $\mathbf{S}_{n-k}$ will have n!/(n - k)! blocks of identical values of length (n - k)! when it is represented in the format that the dense fast Fourier transform uses.

This representation both wastes space and makes the calculation of the fast Fourier transform much slower.

Consequently, SnFFT uses a representation of such a function that stores one value from each block.

The bandlimited fast Fourier transform assumes the the value stored at index *i* is the value associated with all of the permutations that permutation_index() maps to (i - 1) * (n - k)! + 1 to i * (n - k)!.

See example7() for more details.

snf_bl (*N*, *K*, *PA*, *VA*)

```
# Parameters:
#       N::Int
#       - the problem size
#       K::Int
#       - the problem is homogenous at N-K
#       PA::Array{Array{Int, 1}, 1}
#       - PA[i] is a Permutation of N
#       VA::Array{Float64, 1}
#       - VA[i] is the Value associated with PA[i]
# Return Values:
#       SNF::Array{Float64, 1}
#       - SNF[i] is the value associated with all of the permutations that␣
↪permutation_index() maps to any value in the range ((i - 1) * factorial(N - K) +␣
↪1):(i * factorial(N - K))
#       - this is the format for the SNF parameter of sn_fft_bl()
# Notes:
#       - any homogenous coset that doesn't have a representative permutation in PA␣
↪will be assigned a value of zero
```

Sparse Functions

A sparse function over $\mathbf{S}_n$ is represented by two components.

The first is a set of values and the second is a set of indices.

The sparse fast Fourier transform assumes the the value at index *i* is the value associated with the permutation that permutation_index() maps the index at index *i*.

See example5() for more details.

snf_sp (*N*, *PA*, *VA*)

```
# Parameters:
#       N::Int
#       - the problem size
#       PA::Array{Array{Int, 1}, 1}
#       - PA[i] is a Permutation of N
#       VA::Array{Float64, 1}
#       - VA[i] is the Value associated with PA[i]
# Return Values:
#       SNF::Array{Float64, 1}
#       - SNF[i] is the value associated with the permutation that permutation_
↪index() maps to NZL[i]
```

```
#       - this is the format for the SNF parameter of sn_fft_sp()
#       NZL::Array{Int, 1}
#       - NZL must in increasing order
#       - this is the format for the NZL parameter of sn_fft_sp()
# Notes:
#       - the values in VA should be non-zero
```

CHAPTER 5

Young's Orthogonal Representations

SnFFT uses Young's Orthogonal Representations (**YOR**) to calculate the fast Fourier transform of a function over $\mathbf{S}_n$. In addition to Young's Orthogonal Representations, the fast Fourier transform needs to know the structure that determines the decomposition of Young's Orthogonal Representations (**PT**). Additionally, the bandlimited fast Fourier transform needs some information about whether or not a component is a zero-frequency component (**ZFI**). To make computing multiple fast Fourier transforms more efficient, **YOR**, **PT**, and **ZFI** are computed before calling the fast Fourier transform. They only need to be computed once because they don't depend on the specific values of the function over $\mathbf{S}_n$.

Dense and Sparse Functions

Before computing a dense or sparse fast Fourier transform, construct the necessary information with:

```
julia> RA, PT = yor(N)
```

yor(*N*)

```
# Parameters
#       N::Int
#       - the problem size
# Return Values
#       YOR::Array{Array{Array{SparseMatrixCSC, 1}, 1}, 1} (Young's Orthogonal
↪Representations)
#       - YOR[n][p][k] is Young's Orthogonal Representation for the Adjacent
↪Transposition (K, K + 1) for the pth Partition of n
#       PT::Array{Array{Array{Int, 1}, 1}, 1} (Partition Tree)
#       - for each value, i, in PT[n][p], P[n][p] decomposes into P[n-1][i]
#       - length(PT[1]) = 0
```

Bandlimited Functions

Before computing a bandlimited fast Fourier transform, construct the necessary information with:

```
julia> RA, PT, ZFI = yor_bl(N, K)
```

yor_bl (*N*, *K*)

```
# Parameters:
#       N::Int
#       - the problem size
#       K::Int
#       - the problem is homogenous at N-K
# Return Values
#       YOR::Array{Array{Array{SparseMatrixCSC, 1}, 1}, 1} (Young's Orthogonal
↪Representations)
#       - YOR[n][p][k] is Young's Orthogonal Representation for the Adjacent
↪Transposition (K, K + 1) for the pth Partition of n that is needed for the
↪bandlimited functionality
#       - length(YOR[n]) = 0 for n = 1:(N - K - 1)
#       - if p < ZFI[n], length(YOR[n][p] = 1) and YOR[n][p][1,1] contains the
↪dimension of the full Young's Orthogonal Representation
#       PT::Array{Array{Array{Int, 1}, 1}, 1} (Partition Tree)
#       - for each value, i, in PT[n][j], P[n][j] decomposes into P[n-1][i]
#       - length(PT[n]) = 0 for n = 1:(N - K)
#       - length(PT[n][p]) = 0 for p <= ZFI[n]
#       ZFI::Array{Int, 1} (Zero Frequency Information)
#       - ZFI[n] = k if, for p<=k, P[n][p] is a zero frequency partition
```

CHAPTER 6

Fast Fourier Transforms

SnFFT supports three types of fast Fourier transforms. The dense fast Fourier transform can use any type of function over $\mathbf{S}_n$. The bandlimited fast Fourier transform can use only bandlimited functions over $\mathbf{S}_n$, but benefits greatly from the restriction. The sparse fast Fourier transform can run on any type of function over $\mathbf{S}_n$, but becomes faster as the function becomes increasingly sparse.

Dense Fast Fourier Transform

sn_fft (*N*, *SNF*, *YOR*, *PT*)

```
# Parameters:
#       N::Int
#       - the problem size
#       SNF::Array{Foat64, 1}
#       - SNF[i] is the value associated with the Permutation that permutation_
↪index() maps to i
#       YOR::Array{Array{Array{SparseMatrixCSC, 1}, 1}, 1}
#       - output1 from yor()
#       PT::Array{Array{Array{Int, 1}, 1}, 1}
#       - output2 from yor()
# Return Values:
#       FFT::Array{Float64, 2}
#       - FFT is the Fast Fourier Transform of SNF
```

- See snf() for a detailed explanation of the **SNF** parameter
- **YOR** and **PT** are the outputs of yor(**N**)
- See the code for example4() for an example of the complete process to compute a dense fast Fourier transform

Bandlimited Fast Fourier Transform

sn_fft_bl (*N*, *K*, *SNF*, *YOR*, *PT*, *ZFI*)

```
# Parameters:
#       N::Int
#       - the problem size N
#       K::Int
#       - the problem is homogenous at N-K
#       SNF::Array{Foat64, 1}
#       - SNF[i] is the value assigned to the ith homogenous subgroup of size N-K
#       YOR::Array{Array{Array{SparseMatrixCSC, 1}, 1}, 1}
#       - output1 from yor_bl()
#       PT::Array{Array{Array{Int, 1}, 1}, 1}
#       - output2 from yor_bl()
#       ZFI::Array{Int, 1}
#       - output3 from yor_bl()
# Return Values:
#       FFT::Array{Float64, 2}
#       - FFT is the Fast Fourier Transform of SNF
```

- See snf_bl() for a detailed explanation of the **SNF** parameter
- **YOR**, **PT**, and **ZFI** are the outputs of yor_bl(**N**, **K**)
- See the code for example7() for an example of the complete process to compute a bandlimited fast Fourier transform

Sparse Fast Fourier Transform

sn_fft_sp (*N*, *SNF*, *NZL*, *YOR*, *PT*)

```
# Parameters:
#       N::Int
#       - the problem size is N
#       SNF::Array{Foat64, 1}
#       - SNF[i] is the value associated with the Permutation that permutation_
↪index() maps to NZL[i]
#       NZL::Array{Int, 1}
#       - NZL[i] the set of NonZeroLocations for the sparse function over Sn
#       - NZL must be in increasing order
#       YOR::Array{Array{Array{SparseMatrixCSC, 1}, 1}, 1}
#       - output1 from yor()
#       PT::Array{Array{Array{Int, 1}, 1}, 1}
#       - output2 from yor()
# Return Values:
#       FFT::Array{Float64, 2}
#       - FFT is the Fast Fourier Transform of SNF
```

- See snf_sp() for a detailed explanation of the **SNF** and **NZL** parameters
- **YOR** and **PT** are the outputs of yor(**N**)
- See the code for example5() for an example of the complete process to compute a dense fast Fourier transform

CHAPTER 7

Inverse Fast Fourier Transform

SnFFT support three types of inverse fast Fourier transforms. The dense inverse fast Fourier transform can take the inverse of the output of either the sparse or dense fast Fourier transform. The bandlimited inverse fast Fourier transform can only take the inverse of the output of the bandlimited fast Fourier transform, but benefits greatly from the restriction. The partial inverse fast Fourier transform can take the inverse of the output of either the sparse or dense fast Fourier transform.

Dense Inverse Fast Fourier Transform

sn_ifft (*N*, *FFT*, *YOR*, *PT*)

```
# Parameters:
#       N::Int
#       - the problem size
#       FFT::Array{Array{Float64, 2}, 1}
#       - a Fast Fourier Transform of size N
#       - should be the output of sn_fft() or sn_fft_sp()
#       YOR::Array{Array{Array{SparseMatrixCSC, 1}, 1}, 1}
#       - output1 from yor()
#       PT::Array{Array{Array{Int, 1}, 1}, 1}
#       - output2 from yor()
# Return Values:
#       SNF::Array{Float64, 1}
#       - the function over Sn that corresponds to FFT
```

- **FFT** is the output of sn_ftt() or sn_fft_sp()
- **YOR** and **PT** are the outputs of yor(**N**)
- See the code for example4() for an example of the complete process to compute a dense inverse fast Fourier transform

Bandlimited Inverse Fast Fourier Transform

sn_ifft_bl (*N*, *K*, *FFT*, *YOR*, *PT*, *ZFI*)

```
# Parameters:
#       N::Int
#       - the problem size
#       K::Int
#       - the problem is homogenous at N-K
#       FFT::Array{Array{Float64, 2}, 1}
#       - a Fast Fourier Transform of size N
#       - should be the output of sn_fft_bl()
#       YOR::Array{Array{Array{SparseMatrixCSC, 1}, 1}, 1}
#       - output1 from yor_bl()
#       PT::Array{Array{Array{Int, 1}, 1}, 1}
#       - output2 from yor_bl()
#       ZFI::Array{Int, 1}
#       - output3 from yor_bl()
# Return Values:
#       SNF::Array{Float64, 1}
#       - the bandlimited function over Sn that corresponds to FFT
```

- **FFT** is the output of sn_fft_bl()
- **YOR**, **PT**, and **ZFI** are the outputs of yor_bl(**N**, **K**)
- See the code for example7() for an example of the complete process to compute a bandlimited inverse fast Fourier transform

Partial Inverse Fast Fourier Transform

sn_ifft_p (*N*, *M*, *FFT*, *YOR*, *PT*)

```
# Parameters:
#       N::Int
#       - the problem size
#       M::Int
#       - the number of the top components of FFT to use
#       FFT::Array{Array{Float64, 2}, 1}
#       - a Fast Fourier Transform of size N
#       - should be the output of sn_fft() or sn_fft_sp()
#       YOR::Array{Array{Array{SparseMatrixCSC, 1}, 1}, 1}
#       - output1 from yor()
#       PT::Array{Array{Array{Int, 1}, 1}, 1}
#       - output2 from yor()
# Return Values:
#       SNF::Array{Float64, 1}
#       - the function over Sn that corresponds to FFT that has been smoothed to only␣
↪use the top M componenets
```

- **FFT** is the output of sn_ftt() or sn_fft_sp()
- **YOR** and **PT** are the outputs of yor(**N**)
- See the code for example8() for an example of the complete process to compute a partial inverse fast Fourier transform

CHAPTER 8

Convolution and Correlation

SnFFT implements two functions to help analyze the Fourier transform of functions over $\mathbf{S}_{\mathrm{n}}$. They are correlation and convolution. It is important to keep track of the degree of bandlimitedness of the input Fourier transforms because the result will have the higer degree of bandlimitedness.

sn_convoluation(*FFT1*, *FFT2*)

```
# Parameters:
#       FFT1::Array{Array{Float64, 2}, 1}
#       - the first Fourier transform
#       FFT2::Array{Array{Float64, 2}, 1}
#       - the second Fourier transform
# Return Values:
#       Convolution::Array{Array{Float64, 2}, 1}
#       - the convolution of FFT1 and FFT2
# Notes:
#       - FFT1 and FFT2 have to be Fourier transforms of functions over the same Sn
#       - However, the don't have to have the same degree of bandlimitedness
```

sn_correlation(*FFT1*, *FFT2*)

```
# Parameters:
#       FFT1::Array{Array{Float64, 2}, 1}
#       - the first Fourier transform
#       FFT2::Array{Array{Float64, 2}, 1}
#       - the second Fourier transform
# Return Values:
#       Correlation::Array{Array{Float64, 2}, 1}
#       - the correlation of FFT1 and FFT2
# Notes:
#       - FFT1 and FFT2 have to be Fourier transforms of functions over the same Sn
#       - However, the don't have to have the same degree of bandlimitedness
```

CHAPTER 9

Miscellaneous Functions

Partition Construction

Although perhaps not computationally useful, *SnFFT* does export the function to construct the set of paritions of 1:N. Generally, this is useful for giving the output of other code easier to interpret.

partitions(*N*)

```
# Parameters:
#       N::Int
#       - the problem size
# Return Values:
#       P::Array{Array{Array{Int, 1}, 1}, 1} (Partitions)
#       - P[n][p] contains the pth Partition of n
#       WI::Array{Int, 2} (Width Information)
#       - WI[n, w] contains the number of Paritions of n whose first element is less
↪than or equal to w
```

Preference Matrices

Constructs the preference matrix for a permutation.

preferencematrix(*P*)

```
# Parameters:
#       P::Array{Int, 1}
#       - a permutation
# Return Values:
#       Q::Array{Int, 2}
#       - the preference matrix for P
#       - Q[i,j] = 1 if and only if j precedes i in P
```

Kendall Tau Distance

Computes the Kendall Tau distance between two permutations using the permutation's preference matrices.

kendalldistance(*Q1*, *Q2*)

```
# Parameters:
#       Q1::Array{Int, 2}
#       - the preference matrix for the first permutation
#       Q2::Array{Int, 2}
#       - the preference matrix for the second permutation
# Return Values:
#       D::Int
#       - the Kendall Tau Distance between two permutations
#       - the the number of pairs (i, j) such that: P1[i] < P1[j] and P2[i] > P2[j]
```

Mallow's Distribution

Constructs a Mallow's Distribution centered around a specified permutation with a given spreading factor.

mallowsdistribution(*P*, *Gamma*)

```
# Parameters:
#       P::Array{Int, 1}
#       - a permutation
#       Gamma::Float64
#       - the spreading factor
# Return Values:
#       MD::Array{Float64, 1}
#       - the mallows distribution with spreading factor Gamma centered at P
```

Printing Methods

permutation_string(*Permutation*)

```
# Parameters:
#       Permutation::Array{Int, 1}
#       - a permutation
# Return Values:
#       ST::String
#       - the string representation of permutation
```

partition_strign(*Partition*)

```
# Parameters:
#       Partition::Array{Int, 1}
#       - a partition
# Return Values:
#       ST::String
#       - the string representation of partition
```

use sphinx.ext.mathbase

CHAPTER 10

Machine Learning Applications

Clustering Ranked Data - Synthetic Data

example_clustering(C, S, F, N)

```
# Parameters:
#       C::Int
#       - number of clusters
#       S::Int
#       - sizes of the clusters
#       F::Float64
#       - spreading factors for the distributions
#       N::Int
#       - the problem size
# Return Values:
#       None - saves the data to CSV files and runs the clustering script
# Notes: For Julia V0.4
# - the TmStruct and related code can be replaced with "now()"
# - the appropriate version number needs to be used to define $loc
```

As discussed in the main paper, we show clustering results on synthetic ranking data using Fourier features. The data is generated by randomly choosing **C** permutations on $\mathbf{S}_n$ as the cluster centers $\{\sigma^1, \sigma^2, ..., \sigma^C\}$. For each cluster i, $(1 \leq i \leq C)$, we created **S** rankings by applying a random transposition on σ^i. Therefore, the total number of rankings in our synthetic dataset is **D** = **C** x **S**. Further, each ranking instance is represented as a function f_i on $\mathbf{S}_n$. In particular, we used the Mallow's model with spread parameter **F** for each ranking. Then, the Fourier transform is taken for each f_i and the vectorized transform is used for the features. These spectral features are used by the sparcl library to perform the sparse hierarchical clustering of the ranking data.

Multi-Object Tracking

The goal of multi-object tracking is to map paths $\{r_1, r_2, ..., r_n\}$ to real objects $\{o_1, o_2, ..., o_n\}$. This problem becomes difficult when two objects come close to one another or when data about them isn't available, the match mapping from paths to objects becomes uncertain. A natural way to model this problem as a probability distribution over $\mathbf{S}_\mathrm{n}$ because the true mapping isn't always known. Generally, the probability distribution will spread as the time progresses between obeservations because the certainty of the matching decreases as the time between observations increases. Then, once there is another observation, the probability distribution will contract to match the observed data. However, the factorial nature of $\mathbf{S}_\mathrm{n}$ makes this computationally intractable when the number of objects grows much beyond 10 or 11. The Fourier transform over $\mathbf{S}_\mathrm{n}$ becomes useful because a function over $\mathbf{S}_\mathrm{n}$ can be represented very accurately using a small number of the leading coefficients of the Fourier transform. Further, it has been shown that the type of observations seen in this problem can be applied directly to the lower dimensional representation of $\mathbf{S}_\mathrm{n}$. For more information, see Risi Kondor's paper on Multi-Object Tracking here.

Disease Progression

Another potential application to machine learning of the Fourier transform over $\mathbf{S}_\mathrm{n}$ is in finding a disease progression from a dataset. Consider a problem where the features are some test results for a collection of subjects. An interesting, and medically signficant, question is in which order does the disease affect the tests results. Once again, it is clear that this problem can be approached using a probability distribution over $\mathbf{S}_\mathrm{n}$ by representing each subject as such a distribution by sorting their tests results based on severity and then extrapolating probable variants of that original ranking. Unlike in the multi-object tracking case, the benifit of the taking the Fourier transform isn't that it can be used a lower dimensional representation. In this case, the main advantage of using a Fourier representation is that it can identify when one patient is further along in the disease progression than another. This is because a subject who is further along in the disease progression has a probability distribution that is concentrated into a smaller coset within the coset that less advanced patient's probability distribution is concentrated in. This "nested" behavior is observable in the Fourier transform because the coset structure is what the fast Fourier transform uses to become computationally efficient. Consequently, working in the Fourier domain can efficiently capture this kind of relationship between subjects.

CHAPTER 11

Supplement

A theoretical supplement is available here Supplement.pdf.

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

Index