

SOFT: $SO(3)$ Fourier Transforms

Peter J. Kostelec
 Daniel N. Rockmore
 Department of Mathematics
 Dartmouth College
 Hanover, NH 03755
 {geelong,rockmore}@cs.dartmouth.edu

SOFT, version 1.0, is a collection of C routines which compute the discrete Fourier transforms of functions defined on $SO(3)$, the group of orthogonal matrices of determinant 1, i.e. the Rotation Group. *SOFT* is free software and is distributed under the terms of the GNU General Public License.

The routines are based on the “Separation of Variables” technique, e.g. see [3]. Both forward (spatial \rightarrow spectral) and inverse (spectral \rightarrow spatial) Fourier transform routines are provided, as well as examples of how they may be used. A subset of *SpharmonicKit* [7], necessary for doing Cooley-Tukey FFTs and for some of these examples (such as correlation), is also included within this collection. Finally, variations of some routines are provided which use the more efficient *FFTW* [2] collection (version 3, to be precise), and not our home-grown code, to compute the Cooley-Tukey FFTs.

The code was developed and tested in the GNU/Linux environment. Some of the code has also been successfully compiled and executed on an SGI running Irix 6.5, an HP/Compaq Alpha running Tru64 V5.1, and even (under VMware) OpenStep 4.2 for Intel! I do not have access to a Windows machine. However, I do not see there being any reason why the code won’t compile and run under Windows. A minor modification or two might be required, but I do not believe anything drastic should be necessary.

This document is structured as follows:

1. Theoretical Background - p. 2
 - Definitions and properties of functions - p. 2
 - Recurrences satisfied - p. 4
 - Definitions of the transforms being done - p. 4
2. The *SOFT* Package - p. 5
 - How to compile - p. 5
 - Major files; Ordering of samples and coefficients - p. 6
 - Example routines and data provided - p. 9
 - Memory - p. 12
3. Correlation examples - p. 12

We provide theoretical background with the intention of giving the user a precise understanding of what it is the routines in *SOFT* are actually calculating. Within this section, on occasion, a comment on some pertinent implementation detail is made. So it behooves the reader to not skip this portion of the document!

Admittedly, the background is rather on the terse side. More theoretical details, as well as a discussion of the performance of *SOFT*, are to be found in the preprint **FFTs on the Rotation Group** [1]. This preprint is part of the Santa Fe Institute’s “Working Papers” series [6]. It may be downloaded either from the Institute itself, or through a link at www.cs.dartmouth.edu/~geelong/soft/.

The source code for *SOFT* is also available at this website. Questions concerning the software can be sent to the contact person, Peter Kostelec, geelong@cs.dartmouth.edu.

1 Theoretical Background

Since there are many conventions when dealing with functions defined on $SO(3)$, e.g. normalizations, powers of -1 , etc. etc., we say at the outset that the definitions and normalizations we give henceforth are taken from [8].

1.1 Euler Angle Decomposition

An arbitrary rotation about the origin, i.e. any element $g \in SO(3)$, may be expressed as the product of two rotations about the z -axis, and one about the y -axis. Let

$$R_z(A) = \begin{pmatrix} \cos A & -\sin A & 0 \\ \sin A & \cos A & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad R_y(A) = \begin{pmatrix} \cos A & 0 & \sin A \\ 0 & 1 & 0 \\ -\sin A & 0 & \cos A \end{pmatrix} \quad (1)$$

so $R_z(A)$ describes a rotation about the z -axis, and $R_y(A)$ describes a rotation about the y -axis. Then g has the **Euler Angle Decomposition**

$$g = R_z(\alpha) R_y(\beta) R_z(\gamma)$$

where $0 \leq \alpha, \gamma < 2\pi$ and $0 \leq \beta \leq \pi$. Note then that a function f defined on $SO(3)$ can be written as a function of the three Euler angle variables: α , β and γ .

1.2 Wigner D -functions

A Wigner D -function, $D_{MM'}^J(\alpha, \beta, \gamma)$, has three integer indices: J , M , M' . The **degree** J ranges over the non-negative integers. For each J , the **order** indices satisfy the constraint $-J \leq M, M' \leq J$. The Wigner D -function is of the form

$$D_{MM'}^J(\alpha, \beta, \gamma) = e^{-iM\alpha} d_{MM'}^J(\beta) e^{-iM'\gamma}, \quad (2)$$

where $d_{MM'}^J(\beta)$, the **Wigner- d function**, is related to a Jacobi polynomial. An exact definition will be given in the next section.

The collection of Wigner D -functions $\{D_{MM'}^J(\alpha, \beta, \gamma)\}$ form a complete set of orthogonal functions with respect to integration over $SO(3)$:

$$\int_0^{2\pi} d\alpha \int_0^\pi d\beta \sin \beta \int_0^{2\pi} d\gamma D_{M_2 M'_2}^{J_2*}(\alpha, \beta, \gamma) D_{M_1 M'_1}^{J_1}(\alpha, \beta, \gamma) = \frac{8\pi^2}{2J_1 + 1} \delta_{J_1 J_2} \delta_{M_1 M_2} \delta_{M'_1 M'_2}. \quad (3)$$

Hence, any function $f \in L^2(SO(3))$ has the following decomposition:

$$f(\alpha, \beta, \gamma) = \sum_{J \geq 0} \sum_{M=-J}^J \sum_{M'=-J}^J \hat{f}_{MM'}^J D_{MM'}^J(\alpha, \beta, \gamma) \quad (4)$$

where

$$\begin{aligned} \hat{f}_{MM'}^J &= \langle f, D_{MM'}^J \rangle \\ &= \frac{2J+1}{8\pi^2} \int_0^{2\pi} d\alpha \int_0^\pi d\beta \sin \beta \int_0^{2\pi} d\gamma f(\alpha, \beta, \gamma) D_{MM'}^{J*}(\alpha, \beta, \gamma) \end{aligned} \quad (5)$$

and $D_{MM'}^{J*}(\alpha, \beta, \gamma)$ is the complex conjugate of $D_{MM'}^J(\alpha, \beta, \gamma)$. The collection of numbers $\{\hat{f}_{MM'}^J\}$ is the **Fourier transform of f** .

Definition 1.1 A continuous function f on $SO(3)$ is **band-limited with band-limit (or bandwidth) B** if $\hat{f}_{MM'}^l = 0$ for all $l \geq B$.

Implementation Notes

- The C code uses the L^2 -**normalized** versions of the D -functions:

$$\tilde{D}_{MM'}^J(\alpha, \beta, \gamma) = \frac{1}{2\pi} \sqrt{\frac{2J+1}{2}} D_{MM'}^J(\alpha, \beta, \gamma) \quad (6)$$

This means we have (comparing with Eq. 3)

$$\int_0^{2\pi} d\alpha \int_0^\pi d\beta \sin \beta \int_0^{2\pi} d\gamma \tilde{D}_{M_2 M'_2}^{J_2*}(\alpha, \beta, \gamma) \tilde{D}_{M_1 M'_1}^{J_1}(\alpha, \beta, \gamma) = \delta_{J_1 J_2} \delta_{M_1 M_2} \delta_{M'_1 M'_2}. \quad (7)$$

In this normalized situation, we then have

$$f(\alpha, \beta, \gamma) = \sum_{J \geq 0} \sum_{M=-J}^J \sum_{M'=-J}^J \hat{f}_{MM'}^J \tilde{D}_{MM'}^J(\alpha, \beta, \gamma) \quad (8)$$

where

$$\begin{aligned} \hat{f}_{MM'}^J &= \langle f, \tilde{D}_{MM'}^J \rangle \\ &= \int_0^{2\pi} d\alpha \int_0^\pi d\beta \sin \beta \int_0^{2\pi} d\gamma f(\alpha, \beta, \gamma) \tilde{D}_{MM'}^{J*}(\alpha, \beta, \gamma) \end{aligned} \quad (9)$$

- Of the many symmetries $D_{MM'}^J(\alpha, \beta, \gamma)$ observes, one that may be used within the code, when the signal is real-valued, is the following:

$$D_{MM'}^J(\alpha, \beta, \gamma) = (-1)^{M'-M} D_{-M-M'}^{J*}(\alpha, \beta, \gamma). \quad (10)$$

1.3 Wigner d -functions

As promised earlier, we now give a precise definition of the Wigner d -function, $d_{MM'}^J(\beta)$:

$$\begin{aligned} d_{MM'}^J(\beta) &= \zeta_{MM'} \sqrt{\frac{s!(s+\mu+\nu)!}{(s+\mu)!(s+\nu)!}} \left(\sin \frac{\beta}{2} \right)^\mu \left(\cos \frac{\beta}{2} \right)^\nu \\ &\quad \times P_s^{(\mu, \nu)}(\cos \beta) \end{aligned} \quad (11)$$

where

$$\mu = |M - M'| \quad \nu = |M + M'| \quad s = J - \frac{\mu + \nu}{2}$$

and

$$\zeta_{MM'} = \begin{cases} 1 & \text{if } M' \geq M \\ (-1)^{M'-M} & \text{if } M' < M. \end{cases}$$

and $P_s^{(\mu, \nu)}(\cos \beta)$ is a Jacobi polynomial. Note that unless $J \geq \max(|M|, |M'|)$, we have $d_{MM'}^J(\beta) = 0$.

This function satisfies the following orthogonality condition:

$$\int_0^\pi d_{MM'}^J(\beta) d_{MM'}^{J'}(\beta) \sin \beta \, d\beta = \frac{2}{2J+1} \delta_{JJ'}, \quad (12)$$

Implementation Notes

- The C code uses the L^2 -**normalized** versions of the d -functions:

$$\tilde{d}_{MM'}^J(\beta) = \sqrt{\frac{2J+1}{2}} d_{MM'}^J(\beta). \quad (13)$$

- In order to reduce (by a factor of 8!) the number of Wigner- d functions necessary for performing a Fourier transform, the following symmetries are used within most of the C routines (the exceptions will be clearly stated later):

$$d_{MM'}^J(\beta) = (-1)^{M-M'} d_{-M-M'}^J(\beta) = (-1)^{M-M'} d_{M'M}^J(\beta) = d_{-M'-M}^J(\beta) \quad (14)$$

$$= (-1)^{J-M'} d_{MM'}^J(\pi - \beta) = (-1)^{J+M} d_{M-M'}^J(\pi - \beta) \quad (15)$$

$$= (-1)^{J-M'} d_{-M'M}^J(\pi - \beta) = (-1)^{J+M} d_{M'-M}^J(\pi - \beta) \quad (16)$$

1.4 Recurrences

The Wigner- d functions satisfy the following 3-term recurrence:

$$\begin{aligned} 0 &= \frac{\sqrt{[(J+1)^2 - M^2][(J+1)^2 - M'^2]}}{(J+1)(2J+1)} d_{MM'}^{J+1}(\beta) + \left(\frac{MM'}{J(J+1)} - \cos \beta \right) d_{MM'}^J(\beta) \\ &\quad + \frac{\sqrt{(J^2 - M^2)(J^2 - M'^2)}}{J(2J+1)} d_{MM'}^{J-1}(\beta) \end{aligned} \quad (17)$$

Since the C code uses the L^2 -normalized versions of the d -functions, here is the normalized version of the 3-term recurrence (which is used in the C code):

$$\begin{aligned} \tilde{d}_{MM'}^{J+1}(\beta) &= \sqrt{\frac{2J+3}{2J+1}} \frac{(J+1)(2J+1)}{\sqrt{[(J+1)^2 - M^2][(J+1)^2 - M'^2]}} \left(\cos \beta - \frac{MM'}{J(J+1)} \right) \tilde{d}_{MM'}^J(\beta) \\ &\quad - \sqrt{\frac{2J+3}{2J-1}} \frac{\sqrt{[J^2 - M^2][J^2 - M'^2]}}{\sqrt{[(J+1)^2 - M^2][(J+1)^2 - M'^2]}} \frac{J+1}{J} \tilde{d}_{MM'}^{J-1}(\beta). \end{aligned} \quad (18)$$

The recurrence has been verified stable through bandwidths $B = 512$, and it's probably still ok for $B = 1024$.

To properly initialize the normalized recurrence, the C code uses the following identities:

$$\begin{aligned} \tilde{d}_{JM}^J(\beta) &= \sqrt{\frac{2J+1}{2}} \sqrt{\frac{(2J)!}{(J+M)!(J-M)!}} \left(\cos \frac{\beta}{2} \right)^{J+M} \left(-\sin \frac{\beta}{2} \right)^{J-M} \\ \tilde{d}_{-JM}^J(\beta) &= \sqrt{\frac{2J+1}{2}} \sqrt{\frac{(2J)!}{(J+M)!(J-M)!}} \left(\cos \frac{\beta}{2} \right)^{J-M} \left(\sin \frac{\beta}{2} \right)^{J+M} \\ \tilde{d}_{MJ}^J(\beta) &= \sqrt{\frac{2J+1}{2}} \sqrt{\frac{(2J)!}{(J+M)!(J-M)!}} \left(\cos \frac{\beta}{2} \right)^{J+M} \left(\sin \frac{\beta}{2} \right)^{J-M} \\ \tilde{d}_{M-J}^J(\beta) &= \sqrt{\frac{2J+1}{2}} \sqrt{\frac{(2J)!}{(J+M)!(J-M)!}} \left(\cos \frac{\beta}{2} \right)^{J-M} \left(-\sin \frac{\beta}{2} \right)^{J+M}. \end{aligned} \quad (19)$$

1.5 The Transforms

We first define the **quadrature weights** necessary for a bandwidth B transform:

$$w_B(j) = \frac{2}{B} \sin \left(\frac{\pi(2j+1)}{4B} \right) \sum_{k=0}^{B-1} \frac{1}{2k+1} \sin \left((2j+1)(2k+1) \frac{\pi}{4B} \right) \quad (20)$$

where $j = 0, \dots, 2B-1$.

- **Discrete Wigner- d transform:** For given integers (M, M') , define the **Discrete Wigner Transform (DWT)** of a data vector \mathbf{s} to be the collection of sums of the form

$$\hat{\mathbf{s}}(l, M, M') = \sum_{k=0}^{2B-1} w_B(k) \tilde{d}_{M, M'}^l(\beta_k) [\mathbf{s}]_k \quad \max(|M|, |M'|) \leq l < B \quad (21)$$

where $\tilde{d}_{M, M'}^l$ is a normalized Wigner d -function of degree l and orders M, M' , and $\beta_k = \frac{\pi(2k+1)}{4B}$.

Eqn. 21 is what the C code naively evaluates.

We can express the DWT in matrix terms. Let \mathbf{s} = the data vector, $\hat{\mathbf{s}}$ = the coefficient vector, \mathbf{w} = the diagonal matrix whose entries are the weights, and \mathbf{d} = the sampled Wigner- d s, $d_{ij} = \tilde{d}_{M, M'}^i(\beta_j)$. Then we can write the forward (analysis) transform as

$$\mathbf{d} * \mathbf{w} * \mathbf{s} = \hat{\mathbf{s}}.$$

The inverse (synthesis) transform is

$$\mathbf{d}^T * \hat{\mathbf{s}} = \mathbf{s}$$

where \mathbf{d}^T is the transpose of \mathbf{d} .

- **Discrete $SO(3)$ Fourier transform at bandwidth B:** The Discrete $SO(3)$ Fourier transform (*DSOFT*) at bandwidth B of a function $f \in L^2(SO(3))$, denoted $DSOFT(f)$, is the collection of sums of the form:

$$\hat{f}_{MM'}^l = \left(\frac{\pi}{B}\right)^2 \sum_{j_1=0}^{2B-1} \sum_{j_2=0}^{2B-1} \sum_{k=0}^{2B-1} w_B(k) f(\alpha_{j_1}, \beta_k, \gamma_{j_2}) \tilde{D}_{MM'}^{l*}(\alpha_{j_1}, \beta_k, \gamma_{j_2}) \quad (22)$$

$$= \frac{\pi}{(2B)^2} \sum_{k=0}^{2B-1} w_B(k) \tilde{d}_{M, M'}^l(\beta_k) \sum_{j_2=0}^{2B-1} e^{iM'\gamma_{j_2}} \sum_{j_1=0}^{2B-1} e^{iM\alpha_{j_1}} f(\alpha_{j_1}, \beta_k, \gamma_{j_2}) \quad (23)$$

where $l = 0, \dots, B-1$, and $-l \leq M, M' \leq l$. The function is sampled on the $2B \times 2B \times 2B$ grid $\alpha_{j_1} = \frac{2\pi j_1}{2B}$, $\beta_k = \frac{\pi(2k+1)}{4B}$, $\gamma_{j_2} = \frac{2\pi j_2}{2B}$. Eqn. 23 is the discrete version of Eqn. 9.

Eqn. 23 is what the C code evaluates via the **Separation of Variables technique**. The scalars in front of the summations may look odd, but they are different because of the way we defined the normalized Wigner- D and Wigner- d functions.

2 The *SOFT* Package

In this section, we cover such topics as what the package includes, some of the conventions observed (mostly having to do with the format of input and output arrays of the test routines), and how to compile the routines in the first place.

2.1 How To Compile

If all you want to do is compute the forward or inverse discrete $SO(3)$ Fourier transform, and you don't care about doing this as fast as possible, then the *SOFT* package is completely self-contained. Otherwise, you should use one of the flavours of the routines which depend on *FFTW*. Taking the discrete Fourier transform of a function defined on $SO(3)$ involves performing Cooley-Tukey FFTs. While *SOFT* includes such routines, the ones provided by *FFTW* are more efficient. This being said ...

- **If *FFTW* is on your system:** If *FFTW* is on your system, then

1. In the **Makefile**, set the variables **FFTWINC** and **FFTWLIB** so the compiler knows where to find the *FFTW* header file and libraries, e.g.

```
FFTWINC = -I/net/misc/geelong/local/linux/include
FFTWLIB = -L/net/misc/geelong/local/linux/lib -lfftw3
```

The default setting for each is blank, i.e.

```
FFTWINC =
FFTWLIB =
```

When you define them, don't forget the **-lfftw3** at the end of **FFTWLIB** !!!

2. Make sure the variable **CFLAGS** is defined the way you like. These options are passed to the compiler. The default setting is

```
CFLAGS = -O3 $FFTWINC
```

3. Type

```
make all
```

to compile all the test routines in the package.

- **If *FFTW* is not on your system:** If *FFTW* is not on your system, then

1. Make sure the variable **CFLAGS** in the **Makefile** is defined the way you like. These options are passed to the compiler. The default setting is

```
CFLAGS = -O3 $FFTWINC
```

Since, by default, **\$FFTWINC** has no value, you can still leave it in **CFLAGS**.

2. Type

```
make allnonfftw
```

to compile all the test routines which **do not** depend on *FFTW*. You will still be able to take the discrete Fourier transform of a function defined on *SO*(3). You just won't do it that quickly.

In Section 2.3, in case you were wondering, we list and describe the test routines the above steps compile.

2.2 Major Files; Data Conventions

While there are lots of source files within the *SOFT* package, the following files contain the functions the user will most likely want to use. The test routines (to be outlined in the next section), will exercise and provide examples of how to use these functions. The source code includes instructions as to how to use the functions, e.g. function arguments.

2.2.1 The Files

We will try to list these files in some logical order, but no promises! First, those related to the Wigner-*d* functions and the DWT (Eqn. 21).

- **makeWigner.c**: Functions necessary for generating the Wigner-*d* functions.
- **wignerTransforms.c**: Functions that compute the DWT. Also used in computing the DWT portion of the DSOF, i.e. Eqn. 23.
- **wignerTransforms_sym.c**: Functions that compute the DWT portion of the DSOF, but expected to be used in **soft_sym.c** (see below).
- **wignerTransforms_fftw.c**: Functions that compute the DWT portion of the DSOF, but expected to be used in **soft_fftw.c** (see below).

Now those having to do with taking the discrete *SO*(3) Fourier transform of a function (Eqn. 23).

- `soft.c`: Functions for computing the forward and inverse DSOFTE; uses the homegrown FFT routines; does **not** use any symmetries of the Wigner- D or Wigner- d functions; computes the necessary Wigner- d functions on the fly.
- `soft_sym.c`: As `soft.c`, but uses the symmetries of the Wigner- d functions (14-16). If the spatial data is known to be strictly real, can tell the routines to take advantage of this, and so use one of the symmetries observed by the Wigner- D function, i.e. Eqn. 10.
- `soft_fftw.c`: Just like `soft_sym.c`, but uses *FFTW*.
- `soft_fftw_pc.c`: Just like `soft_fftw.c`, but assumes that **all** the Wigner- d functions necessary for a complete Fourier transform have been precomputed.
- `soft_fftw_wo.c`: Just like `soft_fftw.c`; does **not** precompute the Wigner- d functions (computes them on the fly); writes over input as much as possible in order to conserve memory, hence it's not so fast.

And finally, those having to do with some applications of Wigner- D functions, and the DSOFTE.

- `so3_correlate_sym.c`: Functions necessary for correlating two functions $f, h \in L^2(S^2)$. The DSOFTE required uses `soft_sym.c`.
- `so3_correlate_fftw.c`: As above, but the DSOFTE required uses `soft_fftw.c`.
- `rotate_so3.c`: Functions necessary for rotating a function $f \in L^2(S^2)$ by massaging f 's spherical coefficients with Wigner- D functions.
- `rotate_so3_mem.c`: As above, but a slightly more memory friendly version, since it writes over the original signal samples with the rotated signal samples.

2.2.2 Ordering of Samples and Coefficients

For all that follows, we're dealing with a fixed bandwidth B .

Let's first deal with the samples. Recall that for a DSOFTE at bandwidth B , the function f needs to be sampled on the $2B \times 2B \times 2B$ grid

$$\{(\alpha_{j_1}, \beta_k, \gamma_{j_2}) \mid 0 \leq k, j_1, j_2 \leq 2B - 1\}$$

where $\alpha_{j_1} = \frac{2\pi j_1}{2B}$, $\beta_k = \frac{\pi(2k+1)}{4B}$, and $\gamma_{j_2} = \frac{2\pi j_2}{2B}$. The C code expects the samples to be ordered as follows:

$$\begin{aligned}
 & f(\alpha_0, \beta_0, \gamma_0) \\
 & f(\alpha_0, \beta_0, \gamma_1) \\
 & \vdots \\
 & f(\alpha_0, \beta_0, \gamma_{2B-1}) \\
 & f(\alpha_1, \beta_0, \gamma_0) \\
 & f(\alpha_1, \beta_0, \gamma_1) \\
 & \vdots \\
 & f(\alpha_{2B-1}, \beta_0, \gamma_{2B-1}) \\
 & f(\alpha_0, \beta_1, \gamma_0) \\
 & f(\alpha_0, \beta_1, \gamma_1) \\
 & \vdots \\
 & f(\alpha_{2B-1}, \beta_{2B-1}, \gamma_{2B-1})
 \end{aligned}$$

So of the three indices, j_2 iterates the fastest, and k the slowest. Think of it as sampling at all legal longitudes for each latitude. That's how the S^2 transform works.

Since the function can be complex-valued, the samples are **always** in “complex” format. Even if the function is real-valued, you still need to set the imaginary parts of the samples to 0.

Now, and this is important, some routines expect the samples to be in separate, real-imaginary arrays, and others expect the samples to be a single, interleaved real-imaginary array. The rule-of-thumb is simple: if the routine does **not** use *FFTW*, the samples are divided into separate real-imaginary arrays. Otherwise, the samples are interleaved.

For example, suppose there are four sample values: $1 + 2i$, $3 + 4i$, $5 + 6i$, $7 + 8i$. A non-*FFTW* routine will expect two arrays - one holding the real part: 1, 3, 5, 7, and the other holding the imaginary part: 2, 4, 6, 8. Otherwise, the samples are in the single, real-imaginary interleaved array 1, 2, 3, 4, 5, 6, 7, 8.

The example routines, which will be listed in the next section, include examples of both ... including one exception (sorry).

Now for the Fourier coefficients. First, to get the easy part over with, what was said about the samples is true here. The real-imaginary parts of the coefficients will be in separate arrays for the non-*FFTW* routines, and real-imaginary interleaved otherwise. And how are the coefficients ordered? Well, it might seem a little weird, but bear with me.

Consider a matrix A whose **rows** are indexed by M as follows:

$$M = 0, 1, 2, \dots, B-1, -(B-1), -(B-2), \dots, -1$$

This is the order they occur, e.g. if $B = 4$, then the fifth row corresponds to $M = -3$. Similarly for the columns, indexed by M' :

$$M' = 0, 1, 2, \dots, B-1, -(B-1), -(B-2), \dots, -1$$

E.g. the seventh column corresponds to $M' = -1$. Ok, now I reveal that the element at $A(i, j)$ is actually an **array** which contains the Fourier coefficients

$$\{\hat{f}_{ij}^l = \langle f, \tilde{D}_{ij}^l \rangle \mid \max(|i|, |j|) \leq l \leq B-1\}$$

Now, *finally*, write down this matrix A in row-major format. E.g. First write down the set of coefficients for $M = 0, M' = 0$, then for $M = 0, M' = 1$, then for ... , then for $M = 0, M' = -(B-1)$, then for ..., then for $M = 0, M' = -1$, then for $M = 1, M' = 0$, and then $M = 1, M' = 1$, and so on. You get the idea. Believe me, in some sense, this is natural.

To make things easier, here are four formulæ which will tell you where in the list the coefficient $f_{MM'}^l$ occurs. There are four functions, depending on the signs of M, M' . These formulæ can be simplified, but then they might seem a little more mysterious.

Let B denote the bandwidth, $h(M, M', B) = B - \max(|M|, |M'|)$. Then the location of $f_{MM'}^l$ in the file is

$$\sum_{k=0}^{M-1} (B^2 - k^2) + \sum_{k=0}^{M'-1} h(M, k, B) + (l - \max(M, M')) + 1 \quad \text{if } M, M' \geq 0 \quad (24)$$

$$\sum_{k=0}^M (B^2 - k^2) - \sum_{k=M'}^{-1} h(M, k, B) + (l - \max(M, |M'|)) + 1 \quad \text{if } M \geq 0, M' < 0 \quad (25)$$

$$\frac{4B^3 - B}{3} - \sum_{k=1}^{|M|} (B^2 - k^2) + \sum_{k=0}^{M'-1} h(M, k, B) + (l - \max(|M|, M')) + 1 \quad \text{if } M < 0, M' \geq 0 \quad (26)$$

$$\frac{4B^3 - B}{3} - \sum_{k=1}^{|M|-1} (B^2 - k^2) - \sum_{k=M'}^{-1} h(M, k, B) + (l - \max(|M|, |M'|)) + 1 \quad \text{if } M, M' < 0 \quad (27)$$

If you program this in C, you don't have to do that “+1”. I.e. as it's written now, the formula for $M = M' = 0$ will tell you that the location of \hat{f}_{00}^0 is 1.

There is a C version of the above formulæ. Defined in `utils_so3.c`, the function `so3CoefLoc()` takes as inputs: the bandwidth B , degree l , and orders M, M' . It returns the index of $\hat{f}_{MM'}^l$ in the coefficient array (so it does not have that “+1” in it).

2.3 The Test Routines

Here are the example routines compiled with `make all`. If you forget how the arguments go, just type the command and it will return them to you. Hopefully, the examples will provide a sufficient introduction as to how adapt the routines for your own use.

First, here are the routines which involve the Wigner- d functions.

- `test_wigSpec`: Example routine to generate the Wigner- d function to jump-start the recurrence for orders M , M' and bandwidth B , i.e. Eqns. 19. Saves the results in a user-specified file. E.g.

```
test_wigSpec m1 m2 bw output_file_name
```

- `test_genWig`: Generate all the Wigner- d functions needed for a DWT at bandwidth B , orders M , M' . Saves the results in a user-specified file. E.g.

```
test_genWig m1 m2 bw output_file_name
```

- `test_Wigner_Analysis`: Does a DWT, i.e. Eqn. 21, at a user-specified bandwidth and orders. Needs a strictly real (no 0s for imaginary part!) input array to read samples in from, and the name of an output file to write the results to. E.g.

```
test_Wigner_Analysis m1 m2 bw input_file output_file
```

- `test_Wigner_Synthesis`: Does an inverse DWT at a user-specified bandwidth and orders. Needs a strictly real (no 0s for imaginary part!) input array to read samples in from, and the name of an output to write the results to. E.g.

```
test_Wigner_Synthesis m1 m2 bw input_file output_file
```

- `test_Wigner_Naive`: To test speed and stability, does X-many inverse-forward DWTs (X defined by the user) on randomly generated Wigner- d coefficients. No user input required, can save the errors if you'd like. E.g.

```
test_Wigner_Naive m1 m2 bw loops [output_file]
```

or do 100 loops at bandwidth $B = 16$, orders $M = M' = 0$, not bothering to save the errors:

```
test_Wigner_Naive 0 0 16 100
```

Now, those routines dealing with the DSOFTE.

- `test_soft`: To test speed and stability; does X-many inverse-forward DSOFTEs at bandwidth B , via `soft.c`. The coefficients are randomly generated, resulting in a complex-valued signal. Computes the Wigner- d functions on the fly. No user input required, can save the errors if you'd like **in separate real-imaginary files!**. E.g.

```
test_soft bw loops [realError_file imagError_file]
```

or do 10 loops at bandwidth $B = 16$, not bothering to save the errors:

```
test_soft 16 10
```

- `test_soft_for`: Does a forward DSOFTE at bandwidth B via `soft.c`; user-input expected; uses separate real-imaginary arrays; can order the output coefficients in either the algorithm's order, as described in Sec. 2.2.2, or in "human order," which goes as follows:

```

for l = 0 : bw - 1
  for m1 = -1 : 1
    for m2 = -1 : 1
      coefficient of degree l, orders m1, m2

```

Set `order_flag` to 0 for the algorithm's order, 1 for human order. E.g.

```
test_soft_for bw realSam_file imagSam_file realCoef_file imagCoef_file order_flag
```

- `test_soft_inv`: Just like `test_soft_for`, but does an inverse DSOF. E.g.

```
test_soft_inv bw realCoeff imagCoeff realSam imagSam
```

The ordering of the input coefficients **must be** the algorithm's order, and not the human order.

- `test_soft_sym`: Just like `test_soft`, but uses the Wigner- d symmetries (Eqns. 14-16); `soft_sym.c` routines; uses separate real-imaginary arrays. As it's "packaged," the routines generate random coefficients with no restrictions whatsoever, in the sense that it is possible, by commenting and uncommenting the appropriate block of code within the test routine, to generate random coefficients such that the inverse transform results in a (random) real-valued signal.

If you do generate random real-valued signals this way, don't forget to adjust the "real/complex" flags in the forward and inverse routines appropriately, to take advantage of the real-valuedness, i.e. get to use a symmetry of the Wigner- D functions to make the routine a little more efficient. Look at the documentation in `soft_sym.c`.

- `test_soft_sym_for`: Just like `test_soft_for`, but uses the Wigner- d symmetries (Eqns. 14-16); uses separate real-imaginary arrays
- `test_soft_sym_inv`: Just like `test_soft_inv`, but uses the Wigner- d symmetries (Eqns. 14-16); uses separate real-imaginary arrays
- `test_soft_fftw`: Just like `test_soft`, but uses the Wigner- d symmetries (Eqns. 14-16) **and** *FFTW*; `soft_fftw.c` routines; uses interleaved real-imaginary arrays
- `test_soft_fftw_pc`: Just like `test_soft_fftw`, but precomputes all the Wigner- d s necessary in advance of any transforming.
- `test_soft_fftw_wo`: Just like `test_soft_fftw`, but routines try to save memory, e.g. in-place FFTs.

Now, some application-type examples.

- `test_soft_sym_correlate`: Routine to correlate two functions $f, h \in L^2(S^2)$ of bandwidth B . The inputs are the spherical (**not** $SO(3)$!!!) coefficients of f and h in **interleaved format**. The **ordering** of the coefficients is that produced by the routines in *SpharmonicKit*. The function `seanindex()`, defined in `primitive_FST.c` from *SpharmonicKit* (this file is provided in the *SOFT* distribution), takes as its arguments the bandwidth B , degree l , and order m , and returns the location of the **spherical** coefficient \hat{f}_l^m in the coefficient array. Uses `soft_sym.c`, `so3_correlate_sym.c`. Also has the additional parameter `degLim`, which allows you to choose the highest degree coefficients you're willing to use. E.g. Even though the two functions are of bandwidth $B = 8$, you might want to use only the Wigner- D s through degree 5. The routine returns the (α, β, γ) which maximizes the correlation, i.e. the $g = g(\alpha, \beta, \gamma) \in SO(3)$ which maximizes

$$C(g) = \int_{S^2} f(\omega) \overline{\Lambda(g)h(\omega)} d\omega$$

The user has the option of saving all the correlation values, E.g.

```
test_soft_sym_correlate sigCoefs patCoefs bw degLim [result]
```

The test routine assumes that the two functions f and h are **real-valued**, so the correlation values returned are strictly real numbers. If f and h are complex-valued, **you** will have to make the appropriate adjustments in `test_soft_sym_correlate.c`.

- `test_soft_sym_correlate2`: Just like `test_soft_sym_correlate` **except** the user-provided inputs are the samples values (not the coefficients!) of f and h in **interleaved format**. The functions are sampled on the following S^2 grid (which is the same as *SpharmonicKit* expects them):

$$\{(\theta_j, \phi_k) \mid 0 \leq j, k \leq 2B - 1\}$$

where $\theta_j = \frac{\pi(2j+1)}{4B}$ is colatitude, and $\phi_k = \frac{2\pi k}{2B}$ is longitude. The samples are ordered so that k iterates faster than j , e.g. $(\theta_0, \phi_0), (\theta_0, \phi_1), \dots, (\theta_1, \phi_0), (\theta_1, \phi_1), \dots, (\theta_{2B-1}, \phi_{2B-1})$. This ordering should look familiar.

Another exception to `test_soft_sym_correlate`: along with `degLim`, can also specify the **bandwidth** of the inverse $SO(3)$ Fourier transform. E.g. So you can correlate two $B = 256$ functions $f, h \in S^2$ by doing a bandwidth $B = 32$ inverse $SO(3)$ Fourier transform. However, it must be the case that `bwIn` \geq `bwOut`. As above, can save the correlation values:

```
test_soft_sym_correlate2 signalFile patternFile bwIn bwOut degLim [result]
```

- `test_soft_fftw_correlate2`: Just like `test_soft_sym_correlate2` but uses *FFTW*; uses `soft_fftw.c` and `so3_correlate_fftw.c`. E.g.

```
test_soft_fftw_correlate2 signalFile patternFile bwIn bwOut degLim [result]
```

- `test_s2_rotate`: test function to rotate a function $f \in S^2$ by specifying the three Euler angles α, β and γ . The samples input and output are interleaved. Can up- or down-sample by specifying the input and out bandwidths. To generate the Wigner- D functions necessary for massaging the spherical coefficients, we adapt an algorithm of Risbo's [5]. In some sense, in this situation it is more natural to use this algorithm than the usual 3-term recurrence.

```
test_s2_rotate bwIn bwOut degOut alpha beta gamma input_filename output_filename
```

Here are the order of rotation events:

1. First rotate by γ about the z -axis
2. Then rotate by β about the y -axis
3. And finally rotate by α about the z -axis.

- `test_s2_rotate_mem`: Just like `test_s2_rotate` but a little friendlier on the memory. Assumes that `bwIn` equals `bwOut`.

```
test_s2_rotate_mem bwIn degOut alpha beta gamma input_filename output_filename
```

2.4 The Test Data

Included in the *SOFT* distribution are the following function samples. They can be used to verify that things are working as they should.

- `D101real_bw4.dat`, `D101imag_bw4.dat`: The real and imaginary parts of $(2 + i)\tilde{D}_{01}^1(\alpha, \beta, \gamma)$, i.e. $J = 1$, $M = 0$, and $M' = 1$, sampled on the bandwidth $B = 4$ grid. This can be verified by doing

```
test_soft_for 4 D101real_bw4.dat D101imag_bw4.dat rCoeff.dat iCoeff.dat 1
```

and then checking `rCoeff.dat` and `iCoeff.dat`, the real and imaginary parts of the Fourier coefficients.

- `D3-11real_bw4.dat`, `D3-11imag_bw4.dat`: The real and imaginary parts of $\tilde{D}_{-11}^3(\alpha, \beta, \gamma)$, i.e. $J = 3$, $M = -1$, and $M' = 1$, sampled on the bandwidth $B = 4$ grid. This can be verified by doing

```
test_soft_sym_for 4 D3-11real_bw4.dat D3-11imag_bw4.dat rCoeff.dat iCoeff.dat 1
```

- `dSum_real_bw4.dat`, `dSum_imag_bw4.dat`: The real and imaginary parts of

$$(2 + i\sqrt{2})\tilde{D}_{10}^1(\alpha, \beta, \gamma) + (7 + i\sqrt{3})\tilde{D}_{0-2}^3(\alpha, \beta, \gamma) + (-\sqrt{5} + 11i)\tilde{D}_{22}^2(\alpha, \beta, \gamma)$$

sampled on the bandwidth $B = 4$ grid. This can be verified by doing

```
test_soft_sym_for 4 dSum_real_bw4.dat dSum_imag_bw4.dat rCoeff.dat iCoeff.dat 1
```

- `randomS2sig_bw8.dat`: A strictly real-valued, bandlimited function on S^2 , with bandwidth $B = 8$. Since this file is expected to be used with the correlation routines, it is **interleaved**, and the imaginary parts are all 0.
- `randomS2sigA_bw8.dat`: The signal `randomS2sig_bw8.dat` rotated by the Euler angles $\alpha = \pi/8$, $\beta = 11\pi/32$, and $\gamma = \pi/4$. As with the original signal, this one is strictly real-valued, with bandlimit $B = 8$. **Note** that the angles I am rotating by are **exactly** on the $2B \times 2B \times 2B$ grid necessary for a bandlimit $B = 8$ forward or inverse DSOF. This is not a coincidence.
- `randomS2sigB_bw8.dat`: The signal `randomS2sig_bw8.dat` rotated by the Euler angles $\alpha = 0.452$, $\beta = 1.738$, and $\gamma = 2.378$. As with the original signal, this one is strictly real-valued, with bandlimit $B = 8$. **Note** that these angles are **not** on the $2B \times 2B \times 2B$ grid necessary for a bandlimit $B = 8$ forward or inverse DSOF. This is not a coincidence, either.

2.5 Memory

In Table 1 are the memory requirements for the DSOF test routines. The ones involving the Wigner- d transforms don't use that much memory, but these guys do. They are real hogs. (It might be possible to be more careful with the memory, to be less of a hog. I need to look into this.) Once you see the list, you'll understand why the sample data is of such small bandwidths (at least when compared with *SpharmonicKit*).

Now realize that this is for the test routines themselves, e.g. some of the memory is allocated for storing original values of things like samples and coefficients, in order to compare them with what's computed (e.g. for computing errors). If you're not interested in those things, if you're just using the "transform" C functions themselves, then memory use won't be as bad.

2.6 Correlation Examples

In this section, we go through a couple of examples of how to correlate two real-valued functions defined on S^2 . That is, given two functions $f, h \in L^2(S^2)$, we will determine the rotation $g = g(\alpha, \beta, \gamma) \in SO(3)$ which maximizes the correlation

$$C(g) = \int_{S^2} f(\omega) \overline{\Lambda(g)h(\omega)} d\omega.$$

where α, β, γ are the Euler angles defining the rotation. Briefly, from the S^2 Fourier coefficients of f and h , one constructs the $SO(3)$ Fourier coefficients of $C(g)$. Taking the inverse $SO(3)$ Fourier transform yields $C(g)$ evaluated on the $2B \times 2B \times 2B$ grid (where B equals the bandwidth of the inverse $SO(3)$ Fourier transform). Finding the location of the maximum value on the grid tells you how to rotate h .

2.6.1 First Example

The main purpose of this example is just to make sure the code is working properly after compilation. The two functions we will correlate are those whose samples are contained in the files `randomS2sig_bw8.dat` and `randomS2sigA_bw8.dat`.

Routines	$B = 8$	$B = 16$	$B = 32$	$B = 64$	$B = 128$	$B = 256$
test_soft test_soft_sym test_soft_fftw	0.21	2	14	107	854	6828
test_soft_for test_soft_sym_for test_soft_inv test_soft_sym_inv	0.21	2	13	101	811	6487
test_soft_fftw_pc	1	3	20	197	2252	29000 (wow!)
test_soft_fftw_wo	0.5	2	10	74	600	4780
test_soft_sym_correlate test_soft_sym_correlate2 test_soft_fftw_correlate2	< 0.5	1	9	70	560	4500
test_s2_rotate	< 0.5	< 0.5	0.5	2	12	80
test_s2_rotate_mem	< 0.5	< 0.5	0.3	1.3	8	52

Table 1: *Very* approximate memory requirements of DSOFTE-related test routines, in megabytes (2^{20} bytes = 1 megabyte), assuming using C type `double`. In those routines where it is relevant, it is assumed that the “bandwidth in” equals the “bandwidth out.” Note that I have not run all the routines at all the bandwidths listed in this table.

Ok. Let f be the function whose samples are in `randomS2sigA_bw8.dat`, and h be the function whose samples are in `randomS2sig_bw8.dat`. We wish to determine how to rotate h so that the correlation is maximized. We can think of this graphically: how do we rotate h so that its graph matches that of f ’s? We know what the answer *should* be:

$$\begin{aligned}\alpha &= \pi/8 \text{ (about 0.392699)} \\ \beta &= 11\pi/32 \text{ (about 1.07922)} \\ \gamma &= \pi/4 \text{ (about 0.785398)}.\end{aligned}$$

Hopefully this is what the answer *will* be when you run it yourself. We can use either `test_soft_sym_correlate2` or `test_soft_fftw_correlate2`. Let’s use the latter.

Now our **signal** is f and our **pattern** is h . The bandwidth is $B = 8$. Therefore, we execute the command:

```
test_soft_fftw_correlate2 randomS2sigA_bw8.dat randomS2sig_bw8.dat 8 8 7
```

I will explain the 8 8 7 shortly. Meanwhile, here’s what you should see (the name of my machine is `gallant`):

```
gallant 240: test_soft_fftw_correlate2 randomS2sigA_bw8.dat randomS2sig_bw8.dat 8 8 7
Generating seminaive_naive tables...
Reading in signal file
now taking spherical transform of signal
Reading in pattern file
now taking spherical transform of pattern
freeing seminaive_naive_table and seminaive_naive_tablespace
about to combine coefficients
combine time      = 0.0000e+00
about to inverse so(3) transform
finished inverse so(3) transform
inverse so(3) time      = 0.0000e+00
ii = 5  jj = 1  kk = 2
alpha = 0.392699
beta = 1.079922
gamma = 0.785398
gallant 241:
```

Bingo! We get the correct Euler angles! We know how to rotate h to match f . The indices **ii**, **jj** and **kk** refer to the location, in the $2B \times 2B \times 2B$ grid, where the maximum correlation value occurs: **ii** is the index for β (really - recall how the $SO(3)$ samples are arranged - if you forgot, see Sec. 2.2.2), **jj** for α , and **kk** for γ .

Remark Note that you will get an answer different from the one above if you instead do

```
test_soft_fftw_correlate2 randomS2sig_bw8.dat randomS2sigA_bw8.dat 8 8 7
```

This will tell you how much to rotate f to match h . Be careful not to get confused!

If in addition to the Euler angles, for whatever reasons, you want to save all the correlation values, too, say in a file called `corValues.dat`, then instead execute

```
test_soft_fftw_correlate2 randomS2sigA_bw8.dat randomS2sig_bw8.dat 8 8 7 corValues.dat
```

We now address the 8 8 7. The first 8 refers to the bandwidth of the two input functions. The second 8 refers to the bandwidth you want the inverse DSOFIT done at. Why wouldn't you always want the bandwidth for DSOFIT equal to the bandwidth of the input signals? To answer in a word: memory. Suppose the two S^2 functions you want to correlate are of bandwidth $B = 256$. A quick check of Table 1 will probably show that your machine does not have sufficient memory for a $SO(3)$ Fourier transform at bandwidth $B = 256$.

However, all is not lost. You could instead do the DSOFIT at $B = 32$, e.g.

```
test_soft_fftw_correlate2 signal.dat pattern.dat 256 32 31
```

While this will not use all the information you have available in each of the two S^2 functions, you will still be able to get a (hopefully useful) result. And you will need only barely 32 megs of RAM.

Ok, now for that 7 (to return to the original example). This refers to the maximum **degree** of Wigner- D coefficient the $SO(3)$ transform will use. That is, for a $B = 8$ transform, you are considering $SO(3)$ functions (which $C(g)$ is)

$$f(\alpha, \beta, \gamma) = \sum_{J=0}^7 \sum_{M=-J}^J \sum_{M'=-J}^J \hat{f}_{MM'}^J D_{MM'}^J(\alpha, \beta, \gamma)$$

The 7 in the J -summation is the 7 in the input line. The maximum degree is one less the bandwidth.

Now suppose, for whatever reasons, you may not want to use all the Wigner- D functions. You may still want to perform the DSOFIT at $B = 8$, but you're fine with going through degree, say 4. In this case, then, you want to consider

$$f(\alpha, \beta, \gamma) = \sum_{J=0}^4 \sum_{M=-J}^J \sum_{M'=-J}^J \hat{f}_{MM'}^J D_{MM'}^J(\alpha, \beta, \gamma),$$

basically setting all the $\hat{f}_{MM'}^J$ equal to 0 for $5 \leq J \leq 7$. In this case you would do

```
test_soft_fftw_correlate2 signal.dat pattern.dat 8 8 4
```

and that's it. You're still doing a $SO(3)$ transform at $B = 8$. You're just not using all the coefficients you can.

2.6.2 Second Example

This will be like the first example, except the signal f will be that whose samples live in `randomS2sigB_bw8.dat`. Again, we know how the function was rotated:

$$\begin{aligned} \alpha &= 0.452 \\ \beta &= 1.738 \\ \gamma &= 2.378. \end{aligned}$$

However, the critical difference between here and the previous example is that these rotation angles are **not** on the $B = 8$ grid used when doing the DSOFIT. Therefore, we will **not** get these exact numbers from the test routine. The Euler angles returned will be those **on the grid** which will yield the largest correlation:

```

gallant 258: test_soft_fftw_correlate2 randomS2sigB_bw8.dat randomS2sig_bw8.dat 8 8 7
Generating seminaive_naive tables...
Reading in signal file
now taking spherical transform of signal
Reading in pattern file
now taking spherical transform of pattern
freeing seminaive_naive_table and seminaive_naive_tablespace
about to combine coefficients
combine time      = 0.0000e+00
about to inverse so(3) transform
finished inverse so(3) transform
inverse so(3) time      = 0.0000e+00
ii = 8  jj = 1  kk = 6
alpha = 0.392699
beta = 1.668971
gamma = 2.356194
gallant 259:

```

The Euler angles returned are still pretty close to the truth. You could then rotate the pattern by that amount, e.g.

```

gallant 260: test_s2_rotate 8 8 7 0.392699 1.668971 2.356194 randomS2sig_bw8.dat xxx.dat
Generating seminaive_naive tables...
Generating seminaive_naive tables...
Generating trans_seминаive_naive tables...
reading in signal ...
about to rotate ...
finished rotating ...
rotation time      = 0.0000e+00
finished writing ...
gallant 261:

```

and xxx.dat contains the rotated pattern.

2.6.3 Third Example

Try correlating a function with itself, e.g.

```
test_soft_fftw_correlate2 randomS2sig_bw8.dat randomS2sig_bw8.dat 8 8 7
```

You might not get the answer you expect, but it is correct. Hint: Where are we sampling $C(g)$? Also, add together the α and γ you get, and then think about the rotations these correspond to, i.e. which axis are you rotating about?

3 Bibliography

Here are the references. Enjoy!

References

- [1] P. Kostelec and D. Rockmore, FFTs on the Rotation Group, Santa Fe Institute's Working Papers series, Paper #: 03-11-060, 2003, www.santafe.edu/sfi/publications/wpabstract/200311060
- [2] *FFTW* is a free collection of fast C routines for computing the Discrete Fourier Transform in one or more dimensions. It includes complex, real, symmetric, and parallel transforms, and can handle arbitrary array sizes efficiently. *FFTW* is available at www.fftw.org/.

- [3] D. Maslen and D. Rockmore, Generalized FFTs, in *Proceedings of the DIMACS Workshop on Groups and Computation, June 7-10, 1995*, L. Finkelstein and W. Kantor (eds.) (1997), 183-237.
- [4] D. Maslen and D. Rockmore, Separation of Variables and the Computation of Fourier Transforms on Finite Groups I, *Journal of the American Math Society*, **10**(1), (1997), 169-214.
- [5] T. Risbo, Fourier transform summation of Legendre series and D-functions, *Journal of Geodesy*, **70** (1996), p. 383 - 396.
- [6] www.santafe.edu/sfi/publications/working-papers.html.
- [7] *SpharmonicKit* is a freely available collection of C programs for doing Legendre and scalar spherical transforms. Developed at Dartmouth College by S. Moore, D. Healy, D. Rockmore and P. Kostelec, it is available at www.cs.dartmouth.edu/~geelong/sphere/
- [8] D. A. Varshalovich, A. N. Moskalev and V. K. Khersonskii, *Quantum Theory of Angular Momentum*, World Scientific Publishing, Singapore, 1988.