SOFT: *SO*(3) Fourier Transforms

Peter J. Kostelec
and
Daniel N. Rockmore*

September 16, 2007

SOFT, **version 2.0**, is a collection of C routines which compute the discrete Fourier transforms of functions defined on $SO(3)$, the famous Rotation Group. *SOFT* is free software and is distributed under the terms of the GNU General Public License.

Theoretical details, as well as a discussion of the performance of *SOFT*, are to be found in the preprint **FFTs on the Rotation Group** [3], and the references contained therein. The routines in *SOFT* are based on the "Separation of Variables" technique [4]. Both forward (spatial → spectral) and inverse (spectral → spatial) transform routines are provided, as well as examples of how they may be used, e.g. correlation or pattern matching on the sphere. Subsets of *SpharmonicKit* [8] and *S2kit* [7], necessary for some of the routines and examples, are also included in *SOFT*. Finally, variations of some of the transform routines are provided which use the more efficient *FFTW* [2] collection (version 3, to be precise), and not our home-grown code, to perform the "standard" (i.e. Euclidean) FFTs.

Some of the differences between *SOFT* 2.0 and the original v.1.0 are worth noting right up front.

- The *FFTW*-based routines can now handle arbitrary bandwidths, i.e. the bandwidth does **not** have to be a power of 2. (And there was much rejoicing.)
- Files have been reorganized (or should we say, organized). Instead of a single directory containing lots and lots of source and header files, we have tried to place "like-minded" files into separate directories, so things won't appear so overwhelming. E.g. Header files now reside in their own directory.
- One now makes a *library*. Hopefully, having a library will make integrating the *SOFT* routines into your code easier.
- A scaling error in the correlation routines in v.1.0 has been corrected. Details can be found in both sections 2.4.3 and 2.4.4.
- A bug in one of the example routines in v.1.0 has been fixed. More information can be found in section 2.5.3.
- In addition to code being cleaned up (at least a little), some "wrapper" routines have also been written, to provide an easier (but not exclusive) means of interfacing with the *SOFT* library.

The code was developed and tested in the GNU/Linux environment. Some of the code has also been successfully compiled and executed on a Macintosh (PowerPC) running OS 10.3.9 and 10.4.10, 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 for development. (Shocking, but true.) However, I do not see there being any reason why the code won't compile and run under Windows. Some modifications might be required, but I do not believe anything drastic should be necessary.

In this document, we provide some theoretical background, hopefully a sufficient amount to give the user a precise understanding of what it is the routines in *SOFT* are calculating. Interspersed within this background are comments containing pertinent information regarding what is actually implemented. So it behooves the reader to not skip this portion of the document!

Questions concerning the software can be sent to Peter Kostelec `geelong@cs.dartmouth.edu`.

*Department of Mathematics; Dartmouth College; Hanover NH 03755

A Final Word *SOFT* 2.0 has existed, in one form or another, for considerably longer than we care to admit. To overcome inertia (and other obligations), and to finally release the software, we may have spent a little less time on this document you are reading than we would have liked. Some descriptions are less than brief. If something is not clear, look at the fairly well documented source code. That should clear things up, touch wood.

Contents

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 [9].

1.1 Euler Angle Decomposition

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

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

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 $0 \le \alpha, \gamma < 2\pi$ and $0 \le \beta \le \pi$. 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, $D^J_{MM'}(\alpha, \beta, \gamma)$, has three integer indeces: J, M, M'. The **degree** J ranges over the non-negative integers. For each J, the **order** indeces M, M', satisfy the constraint $-J \le M, M' \le J$. The Wigner D-function is of the form

$$D^J_{MM'}(\alpha, \beta, \gamma) = e^{-iM\alpha}\, d^J_{MM'}(\beta)\, e^{-iM'\gamma}, \tag{2}$$

where $d^J_{MM'}(\beta)$, 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 $\{D^J_{MM'}(\alpha, \beta, \gamma)\}$ form a complete set of orthogonal functions with respect to integration over $SO(3)$:

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

where $D^{J*}_{MM'}(\alpha, \beta, \gamma)$ denotes the complex conjugate of $D^J_{MM'}(\alpha, \beta, \gamma)$. Hence, any function $f \in L^2(SO(3))$ has the following decomposition:

$$f(\alpha, \beta, \gamma) = \sum_{J \ge 0} \sum_{M=-J}^{J} \sum_{M'=-J}^{J} \hat{f}^J_{MM'} D^J_{MM'}(\alpha, \beta, \gamma) \tag{4}$$

where

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

The collection of numbers $\{\hat{f}^J_{MM'}\}$ 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 $\hat{f}^l_{MM'} = 0$ for all $l \ge B$.*

Implementation Notes

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

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

  This means we have (comparing with Eq. 3)

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

  In this normalized situation, we then have

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

where

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

- When the signal is real-valued, some of the routines can be told, via function argument, to take advantage of the following symmetry:

$$D^J_{MM'}(\alpha,\beta,\gamma) = (-1)^{M'-M} D^{J*}_{-M-M'}(\alpha,\beta,\gamma). \tag{10}$$

  when performing a transform. The user should look at the source code of the "major" transform routines for further information. What constitutes a "major" routine is basically defined in the first three header files discussed in section 2.4.2.

1.3 Wigner d-functions

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

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

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 $d^J_{MM'}(\beta) = 0$.

This function satisfies the following orthogonality condition:

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

Implementation Notes

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

$$\tilde{d}^J_{MM'}(\beta) = \sqrt{\frac{2J+1}{2}} \; d^J_{MM'}(\beta). \tag{13}$$

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

$$\begin{aligned} d^J_{MM'}(\beta) &= (-1)^{M-M'} d^J_{-M-M'}(\beta) = (-1)^{M-M'} d^J_{M'M}(\beta) = d^J_{-M'-M}(\beta) && (14) \\ &= (-1)^{J-M'} d^J_{-MM'}(\pi-\beta) = (-1)^{J+M} d^J_{M-M'}(\pi-\beta) && (15) \\ &= (-1)^{J-M'} d^J_{-M'M}(\pi-\beta) = (-1)^{J+M} d^J_{M'-M}(\pi-\beta) && (16) \end{aligned}$$

1.4 Recurrences

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

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

Implementation Notes

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

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

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

- To properly initialize the normalized recurrence, the C code uses the following identities (where $0 \le M \le J$):

$$\tilde{d}^{J}_{JM}(\beta) = \sqrt{\frac{2J+1}{2}}\sqrt{\frac{(2J)!}{(J+M)!(J-M)!}} \left(\cos\frac{\beta}{2}\right)^{J+M} \left(-\sin\frac{\beta}{2}\right)^{J-M} \tag{19}$$

$$\tilde{d}^{J}_{-JM}(\beta) = \sqrt{\frac{2J+1}{2}}\sqrt{\frac{(2J)!}{(J+M)!(J-M)!}} \left(\cos\frac{\beta}{2}\right)^{J-M} \left(\sin\frac{\beta}{2}\right)^{J+M} \tag{20}$$

$$\tilde{d}^{J}_{MJ}(\beta) = \sqrt{\frac{2J+1}{2}}\sqrt{\frac{(2J)!}{(J+M)!(J-M)!}} \left(\cos\frac{\beta}{2}\right)^{J+M} \left(\sin\frac{\beta}{2}\right)^{J-M} \tag{21}$$

$$\tilde{d}^{J}_{M-J}(\beta) = \sqrt{\frac{2J+1}{2}}\sqrt{\frac{(2J)!}{(J+M)!(J-M)!}} \left(\cos\frac{\beta}{2}\right)^{J-M} \left(-\sin\frac{\beta}{2}\right)^{J+M}. \tag{22}$$

1.5 The Transforms

We first define the **quadrature weights** necessary for a bandwidth B transform [1]:

$$w_B(j) = \frac{2}{B}\sin\left(\frac{\pi(2j+1)}{4B}\right) \sum_{k=0}^{B-1} \frac{1}{2k+1} \sin\left((2j+1)(2k+1)\frac{\pi}{4B}\right) \tag{23}$$

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

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

$$\hat{\mathbf{s}}(l, M, M') = \sum_{k=0}^{2B-1} w_B(k)\; \tilde{d}^{l}_{M,M'}(\beta_k)[\mathbf{s}]_k \qquad \max(|M|, |M'|) \le l < B \tag{24}$$

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

Eqn. 24 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-ds, $d_{ij} = d^i_{MM'}(\beta_j)$. 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:

$$\hat{f}^l_{MM'} = \left(\frac{\pi}{B}\right)^2 \sum_{j_1=0}^{2B-1} \sum_{j_2=0}^{2B-1} \sum_{k=0}^{2B-1} w_B(k) f(\alpha_{j_1}, \beta_k, \gamma_{j_2}) \tilde{D}^{l*}_{MM'}(\alpha_{j_1}, \beta_k, \gamma_{j_2}) \tag{25}$$

$$= \frac{\pi}{(2B)^2} \sum_{k=0}^{2B-1} w_B(k) \tilde{d}^l_{MM'}(\beta_k) \sum_{j_2=0}^{2B-1} e^{iM'\gamma_{j_2}} \sum_{j_1=0}^{2B-1} e^{iM\alpha_{j_1}} f(\alpha_{j_1}, \beta_k, \gamma_{j_2}) \tag{26}$$

where $l = 0, \ldots, B-1$, and $-l \le M, M' \le l$. 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. 26 is the discrete version of Eqn. 9.
Eqn. 26 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 Directory organization

Instead of a single directory, files have now been organized into appropriate sub-directories. The organization is as follows:

`bin/`	Where the compiled test routines live
`common/`	Contains source code common to both *FFTW*-based, and non-*FFTW*-based routines; basically Wigner d-function code
`examples/`	Contains source code for example Wigner-d function routines (generation and transform)
`examples0/`	Contains source code for example $SO(3)$ Fourier transform routines which **do not** depend on *FFTW*
`examples1/`	Contains source code for example $SO(3)$ Fourier transform routines which **do** depend on *FFTW*
`include/`	Contains header files
`lib0/`	Contains source code for $SO(3)$ Fourier transform library routines which **do not** depend on *FFTW*
`lib1/`	Contains source code for $SO(3)$ Fourier transform library routines which **do** depend on *FFTW*
`sampleData/`	Contains sample data files (ascii format)

2.2 How To Compile

If all you want to do is compute the forward or inverse DSOFT, and you don't care about doing this as fast as possible, or being restricted to powers-of-2 problem sizes, then the *SOFT* package is completely self-contained. Otherwise, you should use one of the flavours of the routines which depend on *FFTW*. Performing a DSOFT involves computing "standard" DFTs. While *SOFT* includes FFT code, the ones provided by *FFTW* are more efficient. Also, they enable the user to perform DSOFTs at arbitrary problem sizes.

2.2.1 If *FFTW* is not on your system

If you do not have *FFTW* on your system, the first thing to do is make a copy of the file `Makefile.fftw0`, and call it `Makefile` . Then ...

1. Set the variable `CFLAGS` in `Makefile` to be what you want. These options are passed to the compiler. The default setting is

   ```
   CFLAGS = -O3
   ```

2. Type

   ```
   make lib
   ```

   to create the library `libsoft0.a`, which will appear in the top-level *SOFT* directory, i.e. the directory containing the `Makefile`.

3. If the previous step worked, then type

   ```
   make tests
   ```

   to compile all the example routines. The library just made will be linked to them. The resulting binaries will be in `bin/`. In Sections 2.5.1 and 2.5.2, we list and describe the test routines this step creates.

Other possible things to `make` are:

`make all`	Make the library and all the example routines
`make clean`	Remove all the object (`*.o`) files
`make vclean`	Remove all the object (`*.o`) files, executables, and library
`make examples`	Make the example routines having to do with the discrete Wigner transform
`make examples0`	Make the example routines having to do with the SO(3) Fourier transform

2.2.2 If *FFTW* is on your system

The first thing to do is make a copy of the file `Makefile.fftw1`, and call it `Makefile` . Then ...

1. In `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 `FFTWLIB`, do not forget to link with the *FFTW* library, e.g. `-lfftw3` (or whatever the name of the *FFTW* library is). In Sections 2.4.4 and 2.5.3, when we discuss the major files in *SOFT*, we will try to note the level of rigor used to generate the `fftw_plan`. You may change the rigor at your descretion (unless the source code suggests otherwise).

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}
   ```

   You **must** have `${FFTWINC}` as one of your options!

3. Type

   ```
   make lib
   ```

   to create the library `libsoft1.a`, which will appear in the top-level *SOFT* directory, i.e. the directory containing the `Makefile`.

4. If the previous step worked, then type

   ```
   make tests
   ```

   to compile all the example routines. The library just made will be linked to them. The resulting binaries will be in `bin/`. In Sections 2.5.1 amnd 2.5.3, we list and describe the test routines this step creates.

Other possible things to `make` are:

`make all`	Make the library and all the example routines
`make clean`	Remove all the object (`*.o`) files
`make vclean`	Remove all the object (`*.o`) files, executables, and library
`make examples`	Make the example routines having to do with the discrete Wigner transform
`make examples1`	Make the example routines having to do with the SO(3) Fourier transform

2.3 Data Conventions: 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 DSOFT at bandwidth B, the function f needs to be sampled on the $2B \times 2B \times 2B$ grid

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

where $\alpha_{j_1} = \dfrac{2\pi j_1}{2B}$, $\beta_k = \dfrac{\pi(2k+1)}{4B}$, and $\gamma_{j_2} = \dfrac{2\pi j_2}{2B}$. 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 indeces, 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 need to be in "complex" format. Ergo, unless otherwise stated, the example routines expect the input sample files to be in interleaved format. For example, suppose there are four sample values: $1+2\imath$, $3+4\imath$, $5+6\imath$, $7+8\imath$. The samples are arranged, in the input file, as $1, 2, 3, 4, 5, 6, 7, 8$, one number per line.

If you want to try your own sample data with an example routine that expect complex samples, and if your samples are strictly real-valued, you will have to interleave 0s into your samples.

Some *SOFT* C functions expect the input samples to be in interleaved format, others as separate real and imaginary arrays, i.e. either a pointer to one length $2N$ `double` array, or two pointers to two length N `double` arrays. Check the documented source files for details.

Now for the ordering of the Fourier coefficients. It might seem a little weird, but bear with me.

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

$$M = 0, 1, 2, ..., B-1, -(B-1), -(B-2), ..., -1$$

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':

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

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

$$\{\hat{f}^l_{ij} =< f, \tilde{D}^l_{ij} > \quad | \quad \max(|i|, |j|) \le l \le B-1\}$$

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 proceed to the second row, and write down the set of coefficients 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 $f^l_{MM'}$ occurs. 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 $f^l_{MM'}$ in the file is

$$\sum_{k=0}^{M-1}(B^2-k^2) + \sum_{k=0}^{M'-1} h(M,k,B) + (l - \max(M, M')) + 1 \qquad \text{if } M, M' \ge 0 \tag{27}$$

$$\sum_{k=0}^{M}(B^2-k^2) - \sum_{k=M'}^{-1} h(M,k,B) + (l - \max(M, |M'|)) + 1 \quad \text{if } M \ge 0,\ M' < 0 \tag{28}$$

$$\frac{4B^3-B}{3} - \sum_{k=1}^{|M|}(B^2-k^2) + \sum_{k=0}^{M'-1} h(M,k,B) + (l - \max(|M|, M')) + 1 \quad \text{if } M < 0,\ M' \ge 0 \tag{29}$$

$$\frac{4B^3-B}{3} - \sum_{k=1}^{|M|-1}(B^2-k^2) - \sum_{k=M'}^{-1} h(M,k,B) + (l - \max(|M|, |M'|)) + 1 \qquad \text{if } M, M' < 0 \tag{30}$$

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 $\hat{f}^0_{00}$ 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 $\hat{f}^l_{MM'}$ in the coefficient array (so it does not have that "+1" in it).

2.4 Major Files

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. However, since all the files are pretty well documented, feel free to look through those we do not list here. The test routines, covered in sections 2.5.1, 2.5.2 and 2.5.3, will exercise and provide examples of how to use the functions. The test data included in *SOFT* is discussed in section 2.6.

2.4.1 `common/`

First, those related to the Wigner-d functions and the DWT (Eqn. 24).

- `makeweights.c`: Functions for calculating the quadrature weights (Eqn. 23).
- `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 DSOFT, i.e. Eqn. 26.

2.4.2 `include/`

This directory contains all the *SOFT* header files. Most of the header files probably will not be of interest, but there are a few that will (hopefully) make *SOFT* easier to use in one's own code.

- `soft.h`: Header file for the "plain" forward and inverse DSOFT routines `Forward_SO3_Naive()` and `Inverse_SO3_Naive()`.
- `soft_fftw.h`: Header file for forward and inverse $SO(3)$ transforms which require *FFTW*: the routines `Forward_SO3_Naive_fftw()` and `Inverse_SO3_Naive_fftw()`.
- `soft_sym.h`: Header file for forward and inverse $SO(3)$ routines which use symmetries of the Wigner little-D functions `Forward_SO3_Naive_sym()` and `Inverse_SO3_Naive_sym()`.
- `wrap.h`: Header file for the "wrapper" versions of *SOFT* routines which do not require *FFTW*. These wrapper functions hide a lot of details behind the scenes, e.g. allocating memory for temporary storage. Using these functions (with their simplified function arguments) provides an easy means of doing forward and inverse SO(3) transforms. However, if you plan on doing lots of SO(3) transforms, you might want to consider using "unwrapped" versions of the functions, e.g. to avoid allocating (and freeing) temporary memory over and over again.
- `wrap_fftw.h`: Just like above, but instead a header file for *SOFT* routines which do require *FFTW*.

2.4.3 `lib0/`

This directory contains files having to do with computing the forward and inverse DSOFT, i.e. (Eqn. 26). The Fourier transforms are evaluated using our home-grown FFT code, and **not** the faster *FFTW*. These routines handle only powers-of-2 bandwidths.

Now, about that scaling error in the correlation routine in *SOFT* 1.0. The derivation in [3] of the recipe for combining S^2 coefficients to produce the $SO(3)$ coefficient assumes that the Wigner-D functions are not normalized. The transform routines, however, do. In v.1.0, the combined S^2 coefficients were not scaled to account for this. In v.2.0, they are. The relevent source file is `so3_correlate_sym.c`.

- `wignerTransforms_sym.c`: Functions that compute the DWT portion of the DSOFT, but expected to be used in `soft_sym.c` (see below).
- `soft.c`: Functions for computing the forward and inverse DSOFT; 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.
- `so3_correlate_sym.c`: Function necessary for correlating two functions $f, h \in L^2(S^2)$. The DSOFT required uses `soft_sym.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.
- `wrap_s2_rotate.c`: Very basic wrapper functions for functions in `rotate_so3.c`.
- `wrap_soft.c`: Very basic wrapper functions for functions in `soft.c`.
- `wrap_soft_sym.c`: Very basic wrapper functions for functions in `soft_sym.c`.
- `wrap_soft_sym_cor2.c`: Very basic wrapper functions for function in `so3_correlate_sym.c`.

2.4.4 `lib1/`

This directory contains files having to do with computing the forward and inverse DSOFT, i.e. (Eqn. 26). The Fourier transforms are evaluated using *FFTW*. These routines handle non powers-of-2 bandwidths.

Now, about that scaling error in the correlation routine in *SOFT* 1.0. The derivation in [3] of the recipe for combining S^2 coefficients to produce the $SO(3)$ coefficient assumes that the Wigner-D functions are not normalized. The transform routines, however, do. In v.1.0, the combined S^2 coefficients were not scaled to account for this. In v.2.0, they are. The relevent source file is `so3_correlate_fftw.c`.

- `wignerTransforms_fftw.c`: Functions that compute the DWT portion of the DSOFT, but expected to be used in `soft_fftw.c` (see below).
- `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_nt.c`: Used to be called `soft_fftw_wo.c` in version 1.0 of *SOFT*. 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 ... or not? Your mileage may vary. You see, one unique thing about the transforms in this file is that there is no explicit function call to perform a matrix transpose. (The "`_nt`" stands for "no transpose.") A cleverly chosen *FFTW* plan is required (see the test routine `test_soft_fftw_nt.c` discussed below in section 2.5.3).
- `so3_correlate_fftw.c`: Function necessary for correlating two functions $f, h \in L^2(S^2)$. The DSOFT required uses `soft_fftw.c`.
- `rotate_so3_fftw.c`: Functions necessary for rotating a function $f \in L^2(S^2)$ by massaging f's spherical coefficients with Wigner-D functions, but uses *FFTW*. Default `fftw_plan` rigor is `FFTW_ESTIMATE`.
- `wrap_s2_rotate_fftw.c`: Very basic wrapper functions for functions in `rotate_so3_fftw.c`.
- `wrap_soft_fftw.c`: Very basic wrapper functions for functions in `soft_fftw.c`. Default `fftw_plan` rigor is `FFTW_MEASURE`.
- `wrap_soft_fftw_cor2.c`: Very basic wrapper functions for function in `so3_correlate_fftw.c`. Default `fftw_plan` rigor is `FFTW_ESTIMATE`.

2.5 The Test Routines

Here are the example routines compiled with `make all` or `make tests`. If you forget how the input arguments go, just execute the command without any function arguments, and they will be returned to you. Hopefully, the examples will provide a sufficient introduction as to how adapt the routines for your own use. Remember that the compiled example routines live in `bin/`.

2.5.1 `examples/`

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

- `test_Wigner_Analysis.c`: Does a DWT, i.e. Eqn. 24, at a user-specified bandwidth and orders. Needs a strictly real (no 0s for imaginary part!) input array to read in samples 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_Naive.c`: 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
  ```

- `test_Wigner_Synthesis.c`: 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_genWig.c`: 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_wigSpec.c`: Example routine to generate the Wigner-d function to jump-start the recurrence for orders M, M' and bandwidth B, i.e. Eqns. 19-22. Saves the results in a user-specified file. E.g.

  ```
  test_wigSpec m1 m2 bw output_file_name
  ```

- `test_Wigner_angle.c`: Example routine to evaluate the Wigner-d functions at user-specified angles. Given orders M, M', bandwidth B, and angles (in **radians**) $\alpha_0, \alpha_1, \ldots, \alpha_N$, the routine will evaluate $d^l_{MM'}(\alpha)$ at the provided angles for $l = \max(|M|, |M'|), ..., B$, and write them to a user-specified text file. The output will be written as an array of size $(B - \max(|M|, |M'|)) \times N$.

  ```
  test_Wigner_angle m1 m2 bw flag output_file_name a0 [ a1 a2 ... aN ]
  ```

  If `flag` is set to 0, the Wigner-d functions will **not** be L^2-normalized. If `flag` is set to 1, they will.

2.5.2 `examples0/`

Now, those routines dealing with the DSOFT which do not require *FFTW*. Keep in mind the bandwidths these routines handle must be a power of 2.

- `test_soft.c`: To test speed and stability; does X-many inverse-forward DSOFTs at bandwidth B, via `soft.c`; uses interleaved real-imaginary arrays. 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 (differences between real and imaginary parts of the coefficients) if you'd like. E.g.

  ```
  test_soft bw loops [error_file]
  ```

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

```
test_soft 16 10
```

- `test_soft_for.c`: Does a forward DSOFT at bandwidth B via `soft.c`; user-input expected; uses interleaved real-imaginary arrays; can order the output coefficients in either the algorithm's order, as described in Sec. 2.3, or in "human order," which goes as follows:

```
for l = 0 : bw - 1
 for m1 = -l : l
  for m2 = -l : l
   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 sampleFile coefFile order_flag
```

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

```
test_soft_inv bw coefFile sampleFile
```

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

- `test_soft_sym.c`: Just like `test_soft`, but uses the Wigner-d symmetries (Eqns. 14-16); `soft_sym.c` routines. 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.c`: Just like `test_soft_for`, but uses the Wigner-d symmetries (Eqns. 14-16); uses interleaved real-imaginary arrays.

- `test_soft_sym_inv.c`: Just like `test_soft_inv`, but uses the Wigner-d symmetries (Eqns. 14-16); uses interleaved real-imaginary arrays.

Now, some application-type examples.

- `test_soft_sym_correlate.c`: 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-Ds through degree 5 (i.e. by setting equal to 0 the higher degree coefficients). 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.c`: 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. $(\theta_0, \phi_0), (\theta_0, \phi_1), ..., (\theta_1, \phi_0), (\theta_1, \phi_1), ..., (\theta_{2B-1}, \phi_{2B-1})$. 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]`

Read Section 3.1, especially the latter half, for a reason why you might want to perform the inverse $SO(3)$ Fourier transform at a bandwidth `bwOut` less than the input S^2 bandwidth `bwIn`.

- `test_soft_sym_correlate2_wrap.c`: A simplified version of `test_soft_sym_correlate2`, but shows how to use one of the "wrapper" routines:

`test_soft_sym_correlate2_wrap signalFile patternFile bw isReal`

If `isReal` is 0, that means the signal and pattern are both complex samples, and so the input sample values are in interleaved format. If `isReal` is 1, then the signal and pattern are real, and the input sample values consist of just the real samples. There is no need to interleave them with 0s.

- `test_s2_rotate.c`: 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 output bandwidths. To generate the Wigner-D functions necessary for massaging the spherical coefficients, we adapt an algorithm of Risbo's [6]. 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.c`: 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`

- `test_s2_rotate_wrap.c`: A simplified version of `test_s2_rotate`, but shows how to use one of the "wrapper" routines:

`test_s2_rotate_wrap bw alpha beta gamma input_filename output_filename isReal`

If `isReal` is 0, that means the signal and pattern are both complex samples, and so the input sample values are in interleaved format. If `isReal` is 1, then the signal and pattern are real, and the input sample values consist of just the real samples. There is no need to interleave them with 0s.

2.5.3 examples1/

These example routines require *FFTW*. These routines handle arbitrary bandwidths, i.e. they do not have to be a power of 2. The routine `test_soft_fftw_wo.c` did **not** work in version 1.0 of *SOFT*. It has been corrected, and the name has changed to `test_soft_fftw_nt.c`. Details a little further down.

- `test_soft_fftw.c`: Just like `test_soft`, but uses the Wigner-d symmetries (Eqns. 14-16) **and** *FFTW*; uses interleaved real-imaginary arrays.

  ```
  test_soft_fftw bw loops [error_file]
  ```

  Default `fftw_plan` rigor is `FFTW_MEASURE`.

- `test_soft_fftw_for.c`: Just like `test_soft_for`, but uses the Wigner-d symmetries (Eqns. 14-16) **and** *FFTW*; uses interleaved real-imaginary arrays.

  ```
  test_soft_fftw_for bw inputFile coef_file isReal order_flag
  ```

  If `isReal` is 0, the function samples are complex, the sample values are interleaved, otherwise, just have the real samples (i.e. do not have to interleave 0s).

- `test_soft_fftw_inv.c`: Just like `test_soft_inv`, but uses the Wigner-d symmetries (Eqns. 14-16) **and** *FFTW*; uses interleaved real-imaginary arrays.

  ```
  test_soft_fftw_inv bw coefFile sample_file isReal
  ```

  If `isReal` is 1, the coefficients are for a strictly real function, and so symmetries will be used when doing the inverse transform (to cut down on the computations.

- `test_soft_fftw_pc.c`: Just like `test_soft_fftw`, but precomputes all the Wigner-ds necessary in advance of any transforming. Default `fftw_plan` rigor is `FFTW_MEASURE`.

- `test_soft_fftw_nt.c`: Was once the broken `test_soft_fftw_wo.c` in *SOFT* 1.0. Just like `test_soft_fftw`, but routines try to save memory, e.g. forward transform writes over inputs, and we let *FFTW* do the matrix transposes by a cleverly chosen plan. So there is no explicit function call in `soft_fftw_nt.c` to perform a matrix transpose. **The routine `test_soft_fftw_wo.c` did not work in *SOFT* 1.0.** There was a problem with our `fftw_plan`, resulting in a **bus error** when doing the transform in the forward direction. A work around has now been implemented. Default `fftw_plan` rigor is `FFTW_MEASURE`.

- `test_soft_fftw_correlate2.c`: 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]
  ```

  Default `fftw_plan` rigor is `FFTW_ESTIMATE`.

- `test_soft_fftw_correlate2_wrap.c`: A simplified version of `test_soft_fft_correlate2`, but shows how to use one of the "wrapper" routines:

  ```
  test_soft_fft_correlate2_wrap signalFile patternFile bw isReal
  ```

  If `isReal` is 0, that means the signal and pattern are both complex samples, and so the input sample values are in interleaved format. If `isReal` is 1, then the signal and pattern are real, and the input sample values consist of just the real samples. There is no need to interleave them with 0s.

- `test_s2_rotate_fftw.c`: Just like `test_s2_rotate` but uses *FFTW*.

  ```
  test_s2_rotate_fftw bwIn bwOut degOut alpha beta gamma input_filename output_filename
  ```

- `test_s2_rotate_fftw_mem.c`: Just like `test_s2_rotate_mem` but uses *FFTW*.

  ```
  test_s2_rotate_fftw_mem bw degOut alpha beta gamma  input_filename output_filename
  ```

- `test_s2_rotate_fftw_wrap.c`: Just like `test_s2_rotate_wrap` but uses *FFTW*.

  ```
  test_s2_rotate_fftw_wrap bw alpha beta gamma input_filename output_filename isReal
  ```

2.6 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.

- `D101_bw4.dat`: The real and imaginary parts (interleaved) of $(2+\imath)\tilde{D}^1_{01}(\alpha,\beta,\gamma)$, 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 D101_bw4.dat Coeff.dat 1
  ```

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

- `D3-11_bw4.dat`: The real and imaginary parts (interleaved) of $\tilde{D}^3_{-11}(\alpha,\beta,\gamma)$, 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 D3-11_bw4.dat Coeff.dat 1
  ```

- `dSum_bw4.dat`: The real and imaginary parts (interleaved) of

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

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

  ```
  test_soft_sym_for 4 dSum_bw4.dat Coeff.dat 1
  ```

- `D751_bw9.dat`: The real and imaginary parts (interleaved) of $\tilde{D}^7_{51}(\alpha,\beta,\gamma)$, i.e. $J=7$, $M=5$, and $M'=1$, sampled on the bandwidth $B=9$ grid. This can be verified by doing

  ```
  test_soft_fft_for 9 D751_bw9.dat Coeff.dat 0 1
  ```

- `dMix_bw10.dat`: The real and imaginary parts (interleaved) of

  $$(\sqrt{2}+\imath)\tilde{D}^3_{12}(\alpha,\beta,\gamma)+\imath\sqrt{3}\tilde{D}^5_{1-4}(\alpha,\beta,\gamma)+(2+\imath\pi)\tilde{D}^6_{-32}(\alpha,\beta,\gamma)+\frac{3}{4}\tilde{D}^8_{47}(\alpha,\beta,\gamma)+\tilde{D}^9_{-5-5}(\alpha,\beta,\gamma)$$

  sampled on the bandwidth $B=10$ grid. This can be verified by doing

  ```
  test_soft_fft_for 10 D751_bw10.dat Coeff.dat 0 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 DSOFT. This is not a coincidence.

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 DSOFT-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.

- `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 DSOFT. This is not a coincidence, either.
- `randomS2sigCX_bw7.dat`: A complex-valued, bandlimited function on S^2, with bandwidth $B = 7$. Since this file is expected to be used with the correlation routines, it is **interleaved**.
- `randomS2sigCXA_bw7.dat`: The signal `randomS2sigCX_bw7.dat` rotated by the Euler angles $\alpha = 6\pi/7$, $\beta = 11\pi/28$, and $\gamma = 3\pi/7$. As with the original signal, this one is complex-valued, with bandlimit $B = 7$. **Note** that the angles I am rotating by are **exactly** on the $2B \times 2B \times 2B$ grid necessary for a bandlimit $B = 7$ forward or inverse DSOFT. This is not a coincidence.

2.7 Memory

In Table 1 are the memory requirements for the DSOFT 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.

3 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.

3.1 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`.

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 indeces `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. 2.3), `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 DSOFT done at. Why wouldn't you always want the bandwidth for DSOFT 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 DSOFT 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)

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

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 DSOFT at $B = 8$, but you're fine with going through degree, say 4. In this case, then, you want to consider

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

basically setting all the $\hat{f}^J_{MM'}$ 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.

3.2 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 DSOFT. 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_seminaive_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.

3.3 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?

4 Bibliography

Here are the references. Enjoy!

References

[1] J. R. Driscoll and D. Healy, Computing Fourier transforms and convolutions on the 2-sphere. (extended abstract) in *Proc.* 34^{th} *IEEE FOCS*, (1989) 344-349; *Adv. in Appl. Math.*, **15** (1994), 202-250.

[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] P. Kostelec and D. Rockmore, FFTs on the Rotation Group, Santa Fe Institute's Working Papers series, Paper #: 03-11-060, 2003.

[4] 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.

[5] 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.

[6] T. Risbo, Fourier transform summation of Legendre series and D-functions, *Journal of Geodesy*, **70** (1996), p. 383 - 396.

[7] *S2kit* is a freely available collection of C programs for doing Legendre and scalar spherical transforms. Developed at Dartmouth College by P. Kostelec and D. Rockmore, derived from *SpharmonicKit* [8], it is available at www.cs.dartmouth.edu/~geelong/sphere/.

[8] *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/.

[9] D. A. Varshalovich, A. N. Moskalev and V. K. Khersonskii, *Quantum Theory of Angular Momentum*, World Scientific Publishing, Singapore, 1988.