February 23, 2004

S2kit: A Lite Version of SpharmonicKit

Peter J. Kostelec
Daniel N. Rockmore
Department of Mathematics
Dartmouth College
Hanover, NH 03755

S2kit, version 1.0, is a collection of C routines which compute the discrete Fourier transforms of functions defined on the sphere, S^2. Inverse transforms are also provided, as well as convolution routines. This collection, which may be downloaded from `www.cs.dartmouth.edu/~geelong/sphere/`, is a "lite" version of *SpharmonicKit* [6], hence the shorter name. (*SpharmonicKit* may also be downloaded from the same website.) *S2kit* is free software and is distributed under the terms of the GNU General Public License.

Some familiarity with *SpharmonicKit* is assumed. Basically, *SpharmonicKit* is a collection of routines which implement discrete Legendre and spherical harmonic transforms by a number of different algorithms. A detailed development and description of the algorithms implemented in *SpharmonicKit* may be found in the paper by Healy et al [4].

S2kit is lite for two reasons. Firstly, unlike *SpharmonicKit*, it contains only two types of discrete Legendre transform algorithms: the **naive** and **semi-naive** algorithms (as defined in [4] - we will repeat the definitions later on within this document). Secondly, *S2kit* **absolutely requires** that *FFTW* [3] (version 3) be installed on the user's platform. *S2kit* does not contain **any** Cooley-Tukey FFT code. This requirement gains the user two benefits. The routines in *FFTW* are more efficient than our home-grown code, and, perhaps more importantly, the user is **no longer restricted** to doing transforms at powers-of-2 bandwidths.

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, and an HP/Compaq AlphaServer running Tru64 V5.1. 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:

We hope this document, while on the terse side, will be sufficient for the user to help get started using the code. The source code itself has alot of documentation. We repeat what was mentioned earlier. For further information regarding the algorithms implemented in *S2kit* , and those contained within *SpharmonicKit*, the user is encouraged to consult [4].

Questions concerning *S2kit* can be sent to the contact person, Peter Kostelec, `geelong@cs.dartmouth.edu`.

1 Theoretical Background

There are a great many references for Legendre polynomials and spherical harmonics. One of our favourites is [7]. For the web-enabled, we mention two. There is mathworld.wolfram.com/LegendrePolynomial.html, as well as mathworld.wolfram.com/SphericalHarmonic.html.

1.1 Definitions of functions

Legendre polynomials The Legendre polynomials P_l form an orthogonal basis for the space of functions $L^2[-1,1]$:

$$\int_{-1}^{1} P_l(x)P_{l'}(x)dx = \frac{2}{2l+1}\delta_{ll'}.$$

A suitably normalized version,

$$\tilde{P}_l(x) = \sqrt{\frac{2l+1}{2}}P_l(x)$$

form an orthonormal basis. **The code uses the normalized version.**

Spherical Harmonics The spherical harmonics, Y_l^m are related to the associated Legendre functions as follows:

$$\begin{aligned} Y_l^m(\theta,\phi) &= \sqrt{\frac{(2l+1)(l-m)!}{4\pi(l+m)!}}P_l^m(\cos\theta)e^{\imath m\phi} \\ &= \sqrt{\frac{1}{2\pi}}\tilde{P}_l^m(\cos\theta)e^{\imath m\phi}. \end{aligned}$$

They satisfy the orthogonality condition

$$\int_0^{2\pi}\int_0^{\pi} Y_l^m(\theta,\phi)\,\bar{Y}_{l'}^{m'}(\theta,\phi)\,\sin\theta\,d\theta\,d\phi = \delta_{ll'}\delta_{mm'}$$

where $\bar{Y}_{l'}^{m'}(\theta,\phi)$ is the complex conjugate of $Y_{l'}^{m'}(\theta,\phi)$.

1.2 Definition of Transforms

Discrete Legendre Transforms Fix a bandlimit B, and sample a function f at the $2B$-many locations $\theta_j = \pi(2j+1)/4B$, where $j = 0, 1, \ldots, 2B-1$. Let f_j denote the j^{th} sample, and w_j denote the j^{th} quadrature weight.

For a given order $m > 0$, the **discrete Legendre transform** (DLT) of f is the collection of sums of the form

$$\begin{aligned} \hat{f}(k) &= \sum_{j=0}^{2B-1} w_j\, f_j\, P_k^m(\cos\theta_j) \\ &= \langle w\, f, P_k^m\rangle \end{aligned} \tag{1}$$

for $k = m, m+1, \ldots, B-1$. We may write the DLT of f as a matrix-vector product:

$$\begin{pmatrix} P_m^m(\cos\theta_0) & P_m^m(\cos\theta_1) & \ldots & P_m^m(\cos\theta_{2B-1}) \\ P_{m+1}^m(\cos\theta_0) & P_{m+1}^m(\cos\theta_1) & \ldots & P_{m+1}^m(\cos\theta_{2B-1}) \\ \vdots & & & \\ P_{B-1}^m(\cos\theta_0) & P_{B-1}^m(\cos\theta_1) & \ldots & P_{B-1}^m(\cos\theta_{2B-1}) \end{pmatrix} \cdot \begin{pmatrix} w_0 f_0 \\ w_1 f_1 \\ \vdots \\ w_{2B-1} f_{2B-1} \end{pmatrix} = \begin{pmatrix} \hat{f}(m) \\ \hat{f}(m+1) \\ \vdots \\ \hat{f}(B-1) \end{pmatrix}. \tag{2}$$

We will call evaluating (2) directly, as is, the **Naive algorithm**. This is what the naive routines of *S2kit* do.

Now, the **discrete cosine transform** (DCT), if normalized right, has the following appealing property. Let $\mathcal{C}$ denote the DCT matrix. The orthogonality of $\mathcal{C}$ implies $\langle \mathcal{C}a, \mathcal{C}b\rangle = \langle a, b\rangle$.

As explained in Section 2.4 of [4], the DCT of the Legendre polynomial P_k has at most $k+1$ many non-zero coefficients. Therefore,

$$\begin{aligned}
\hat{f}(k) &= \langle wf, P_k^m\rangle \\
&= \sum_{j=0}^{2B-1} w_j\, f_j\, P_k^m(\cos\theta_j) \\
&= \sum_{j=0}^{2B-1} [\mathcal{C}(wf)]_j\, [\mathcal{C}P_k^m]_j \\
&= \sum_{j=0}^{k} [\mathcal{C}(wf)]_j\, [\mathcal{C}P_k^m]_j. \qquad (3)
\end{aligned}$$

While we may be abusing the notation somewhat, the idea should be clear: doing the sum (3) in the cosine domain means we only need to add at most $k+1$ terms. The **Semi-naive algorithm** is precisely this: send things to the cosine domain and sum there. Previous work [1] has shown the semi-naive algorithm to be both stable and faster than the naive algorithm. In matrix lingo, taking the DCT of the rows of the matrix in (2) results in a lower-triangular matrix. So we are saving operations in the long run ... after taking the DCT of the weighted data, and assuming the DCTs of the P_l^ms have been precomputed. The semi-naive routines in *S2kit* do this.

Admittedly, we are glossing over some details, but this is the idea. Indeed, we are saving even a few more operations because a degree k Legendre polynomial has only $(k+1)/2$ many non-zero DCT coefficients. So from $2B$-many terms in the summation, we go to $k+1$ many terms, and in fact to $(k+1)/2$ many terms. Again, for a more complete description, see [4].

Discrete Spherical Harmonic Transforms Let $f(\theta,\phi) \in L^2(S^2)$ have bandwidth B. We sample the function on the equiangular $2B \times 2B$ grid $\theta_j = \pi(2j+1)/4B$, $\phi_k = 2\pi k/2B$, where $j = 0, 1, \ldots, 2B-1$ and $k = 0, 1, \ldots, 2B-1$. (So we are sampling at twice the bandwidth.) The quadrature weights we denote, as before, w_j. The **Discrete Spherical Harmonic Transform** of f is the collection of sums of the form

$$\begin{aligned}
\hat{f}(l,m) &= \frac{\sqrt{2\pi}}{2B} \sum_{j=0}^{2B-1} \sum_{k=0}^{2B-1} w_j\, f(\theta_j,\phi_k)\, e^{-\imath m\phi_k} P_l^m(\cos\theta_j) \\
&= \frac{\sqrt{2\pi}}{2B} \sum_{j=0}^{2B-1} w_j P_l^m(\cos\theta_j) \sum_{k=0}^{2B-1} e^{-\imath m\phi_k}\, f(\theta_j,\phi_k) \qquad (4)
\end{aligned}$$

for all $|m| \le l < B$.

Note how (4) is written. It illustrates how one may perform the spherical harmonic transform. First do a bunch of "regular" FFTs, followed by a bunch of discrete Legendre transforms. This is an example of the "Separation of Variables" technique. For further information, see the paper by Maslen and Rockmore [5].

A **naive spherical harmonic transform routine** in *S2kit* does the DLT portion naively, while a **semi-naive spherical harmonic transform routine** in *S2kit* does the DLTs over in the cosine domain.

Convolution on the Sphere Let $f, h \in L^2(S^2)$. From [2], we have that the transform of the convolution of f and h is a pointwise product of their transforms:

$$(f * h)\hat{}(l,m) = 2\pi\sqrt{\frac{4\pi}{2l+1}}\, \hat{f}(l,m)\, \hat{h}(l,0) \qquad (5)$$

2 The *S2kit* 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

Recall that *S2kit* requires *FFTW*. While this might be a nuisance for some, it does enable one to do discrete Legendre and spherical harmonic transforms at arbitrary, i.e. non powers-of-2, bandwidths.

To compile the routines in *S2kit* ,

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

Within *S2kit*, the default rigor of the *FFTW* plans is `FFTW_ESTIMATE`. This may be changed, at your descretion.

The test routines may also be individually made. In Section 2.2 we list and describe the test routines the above steps compile.

2.2 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 you may adapt the routines for your own use.

First, here are the routines having to do with discrete Legendre transforms.

- `test_naive`: Does a **naive** DLT. To test speed and stability, does X-many inverse-forward naive transforms (X defined by the user) on randomly generated Legendre coefficients. The order m and bandwidth B are both provided by the user. Returns error and timing statistics.
  Usage: `test_naive m bw loops`. E.g.

  ```
  gallant 12: test_naive 0 7 10
  bw = 7  m = 0
  loops = 10
  Average r-o error:              2.9643e-15     std dev: 1.8091e-15
  Average (r-o)/o error:          2.0464e-14     std dev: 3.3774e-14
  ```

```
average forward time = 0.0000e+00
average inverse time = 0.0000e+00
gallant 13:
```

- `test_semi`: Does a **semi-naive** DLT. Just like `test_naive`. To test speed and stability, does X-many inverse-forward semi-naive transforms (X defined by the user) on randomly generated Legendre coefficients. The order m and bandwidth B are both provided by the user. Returns error and timing statistics.
  Usage: `test_semi m bw loops`. E.g.

```
gallant 13: test_semi 11 107 1000
bw = 107        m = 11
loops = 1000
Average r-o error:              5.7328e-14     std dev: 1.0441e-14
Average (r-o)/o error:          1.9845e-11     std dev: 3.8895e-10

average forward time = 2.2000e-04
average inverse time = 1.1000e-04
gallant 14:
```

Now those routines dealing with discrete spherical harmonic transforms.

- `test_s2_semi_fly`: To test speed and stability; does X-many inverse-forward spherical harmonic transforms at bandwidth B; the DLT portion of the algorithm is done **semi-naively**. The spherical coefficients are randomly generated. To make it easy on the memory, the associated Legendre functions (and their DCTs) are computed on the fly, as needed by the algorithm. Returns error and timing statistics. Errors may also be saved.
  Usage: `test_s2_semi_fly bw loops [error_file]`.

  NOTE: The spherical transform routines `FST_semi_fly` and `InvFST_semi_fly`, both in `FST_semi_fly.c`, contain the argument `cutoff` which defines at which (if any) order m the DLT is done naively and **not** semi-naively, for orders $m' \geq m$. The default behavior in the test routine is to semi-naive at all orders.

  NOTE 2: The routines also contain the argument `dataformat`. If the signal is known to be real valued, the Fourier coefficients satisfy certain symmetries, i.e. $\hat{f}(l,m) = (-1)^m \overline{\hat{f}(l,m)}$, where the bar denotes conjugate. By setting `dataformat` to `1`, the code can take advantage of this (to achieve greater efficiency). The default setting is `0`.

```
gallant 1233: test_s2_semi_fly 123 3
about to enter loop

inv time         = 3.3000e-01
forward time     = 3.2000e-01
r-o error        = 0.000000000001
(r-o)/o error    = 0.000000000003

inv time         = 3.3000e-01
forward time     = 3.2000e-01
r-o error        = 0.000000000001
(r-o)/o error    = 0.000000000009

inv time         = 3.3000e-01
forward time     = 3.2000e-01
r-o error        = 0.000000000001
(r-o)/o error    = 0.000000000033

Program: test_s2_semi_fly
```

```
Bandwidth = 123
Total elapsed cpu time :                 1.9500e+00 seconds.
Average cpu forward per iteration:       3.2000e-01 seconds.
Average cpu inverse per iteration:       3.3000e-01 seconds.
Average r-o error:               9.6791e-13     std dev: 2.4459e-14
Average (r-o)/o error:           1.4983e-11     std dev: 1.5595e-11

gallant 1234:
```

- `test_s2_semi_memo`: Just like `test_s2_semi_fly`, except that **all** the necessary associated Legendre functions (and their DCTs) are precomputed prior to doing the spherical transforms. This being the case, this routine requires more memory than the "on the fly" version. The timing results do not include the time spent precomputing the DCTs of the Legendres. There are also `cutoff` and `dataformat` parameters in the routines `FST_semi_memo` and `InvFST_semi_memo` which both live in `FST_semi_memo.c`.
  Usage: `test_s2_semi_memo bw loops [error_file]`. E.g. (compare these timings with the "on the fly" - it's the same problem size):

```
gallant 1234: test_s2_semi_memo 123 3
Generating seminaive_naive tables...
Generating trans_seminaive_naive tables...
about to enter loop

inv time         = 6.0000e-02
forward time     = 5.0000e-02
r-o error        = 0.000000000001
(r-o)/o error    = 0.000000000006

inv time         = 5.0000e-02
forward time     = 6.0000e-02
r-o error        = 0.000000000001
(r-o)/o error    = 0.000000000003

inv time         = 6.0000e-02
forward time     = 5.0000e-02
r-o error        = 0.000000000001
(r-o)/o error    = 0.000000000006

Program: test_s2_semi_memo
Bandwidth = 123
Total elapsed cpu time :                 3.3000e-01 seconds.
Average cpu forward per iteration:       5.3333e-02 seconds.
Average cpu inverse per iteration:       5.6667e-02 seconds.
Average r-o error:               9.4423e-13     std dev: 1.1306e-13
Average (r-o)/o error:           4.7985e-12     std dev: 1.6718e-12

gallant 1235:
```

- `test_s2_semi_memo_for`: Does a **forward** spherical harmonic transform, using the semi-naive DLT. Precomputes all necessary associated Legendre functions prior to transforming. Requires from the user the name of a plain text file containing function samples. Will write the computed coefficients to an output file named by the user. There are two possible orderings of the coefficients: "code-ordered" (i.e. suitable for input into another *S2kit* spherical routine), or pretty "human-ordered," i.e.

```
for l = 0 : bw - 1
```

```
for m = -l : l
  print coefficient of degree l, order m
```

The default ordering is "code-ordered." Don't worry - ordering of function samples and "code-ordered" coefficients will be explained in Section 2.3.
Usage: `test_s2_semi_memo_for sampleFile outputFile bw [output_format]`

The `output_format` argument is optional. Default value is `0` for code-ordered coefficients, `1` for human-ordered. E.g.

```
gallant 1244: test_s2_semi_memo_for yMix_bw17.dat cMix_bw17.dat 17 1
Generating seminaive_naive tables...
forward time     = 0.0000e+00
about to write out coefficients
finished writing coefficients
gallant 1245:
```

- `test_s2_semi_memo_inv`: Just like `test_s2_semi_memo_for` but does an **inverse** spherical harmonic transform. Requires from the user the name of a plain text file containing function coefficients in **code-ordered** format (the default output format for `test_s2_semi_memo_for`).
  Usage: `test_s2_semi_memo_inv coeffsFile outputFile bw`. E.g.

```
gallant 1248: test_s2_semi_memo_inv cMix_bw17.dat samples_bw17.dat 17
Generating seminaive_naive tables...
Generating trans_seminaive_naive tables...
inv time         = 0.0000e+00
about to write out samples
finished writing samples
gallant 1249:
```

Now some application-type examples.

- `test_conv_semi_fly`: Convolves two real-valued functions defined on the sphere. Requires from the user the names of two files containing the function samples. Output is a file containing the samples of the convolved result. Uses the semi-naive DLT, and computes on the fly, as needed, the associated Legendre functions.
  Usage: `test_conv_semi_fly signal_file filter_file output_file bw`

  In the notation of (5), the first ("signal") file contains the samples of f, and the second ("filter") file contains the samples of h. E.g.

```
gallant 1259: test_conv_semi_fly s64.dat f64.dat c64.dat 64
Reading signal file...
Reading filter file...
Calling Conv2Sphere_semi_fly()
Writing output file...
gallant 1260:
```

- `test_conv_semi_memo`: Just like `test_conv_semi_fly` but precomputes all necessary associated Legendre functions prior to transforming. Uses the semi-naive DLT.
  Usage: `test_conv_semi_memo signal_file filter_file output_file bw`. E.g.

```
gallant 1263: test_conv_semi_memo s64.dat f64.dat c64.dat 64
Reading signal file...
Reading filter file...
Calling Conv2Sphere_semi_memo()
Writing output file...
gallant 1264:
```

2.3 File Formats

We now describe the formats of the sample and coefficient files which the (relevant) test routines expect and produce. In what follows, we assume the function $f \in L^2(S^2)$ has bandlimit B.

Function Samples For a spherical transform, a function f, with bandlimit B, is sampled on the $2B \times 2B$ grid $\theta_j = \pi(2j+1)/4B$, $\phi_k = 2\pi k/2B$, where $j = 0, 1, \ldots, 2B-1$ and $k = 0, 1, \ldots, 2B-1$.

Since f can be complex-valued, the samples within the plain text file are arranged in the following, **interleaved**, order (so one number per line):

$$
\begin{aligned}
&\text{real part of } f(\theta_0, \phi_0)\\
&\text{imaginary part of } f(\theta_0, \phi_0)\\
&\text{real part of } f(\theta_0, \phi_1)\\
&\text{imaginary part of } f(\theta_0, \phi_1)\\
&\vdots\\
&\text{real part of } f(\theta_0, \phi_{2B-1})\\
&\text{imaginary part of } f(\theta_0, \phi_{2B-1})\\
&\text{real part of } f(\theta_1, \phi_0)\\
&\text{imaginary part of } f(\theta_1, \phi_0)\\
&\text{real part of } f(\theta_1, \phi_1)\\
&\text{imaginary part of } f(\theta_1, \phi_1)\\
&\vdots\\
&\text{real part of } f(\theta_{2B-1}, \phi_{2B-1})\\
&\text{imaginary part of } f(\theta_{2B-1}, \phi_{2B-1})
\end{aligned}
$$

As written, the test routine `test_s2_semi_memo_for` expects the input samples to be in this interleaved format, **even if your function is strictly real-valued**. In that case, every other entry in your sample file would be `0`. Also as written, the samples the test routine `test_s2_semi_memo_inv` produces are written in this same ordering.

The **exception** to the interleaving are the two convolution routines. Since they assume the functions you are convolving are real-valued, the input sample files are expected to have just the real parts of the sample values. You do not have to interleave the `0`s as you would for `test_s2_semi_memo_for`. The output of the convolution routines also just the real parts of the results (since the input is real-valued, so should the output).

Function Coefficients As with the samples, the coefficients are arranged in interleaved real/imaginary format. To set some notation, $\hat{f}(l, m)$ is the coefficient of degree l and order m. The coefficients will be ordered according to m (read the table from left to right, top to bottom):

$$
\begin{array}{ccccc}
\hat{f}(0,0) & \hat{f}(1,0) & \hat{f}(2,0) & \ldots & \hat{f}(B-1,0)\\
 & \hat{f}(1,1) & \hat{f}(2,1) & \ldots & \hat{f}(B-1,1)\\
 & & & & \vdots\\
 & & & \hat{f}(B-2,B-2) & \hat{f}(B-1,B-2)\\
 & & & & \hat{f}(B-1,B-1)\\
 & & & & \hat{f}(B-1,-(B-1))\\
 & & & \hat{f}(B-2,-(B-2)) & \hat{f}(B-1,-(B-2))\\
 & & & & \vdots\\
 & & \hat{f}(2,-2) & \ldots & \hat{f}(B-1,-2)\\
 & \hat{f}(1,-1) & \hat{f}(2,-1) & \ldots & \hat{f}(B-1,-1)
\end{array}
$$

Hopefully you can see the pattern. In some sense, this is natural. This ordering is the **code-ordered** format mentioned earlier.

In case things are a little unclear, here is the function (called `seanindex` in the source code) which, given a bandwidth B, degree l and order m, gives the index of $\hat{f}(l,m)$ within the C array (note that the first element in an array has index 0, the second element index 1, and so on). So the location of $\hat{f}(l,m)$ is

$$\begin{array}{r} m*B-((m*(m-1))/2)+(l-m)) \quad \text{if } m \geq 0 \\ ((B-1)*(B+2))/2)+1+(((B-1)+m)*(B+m)/2)+(l-|m|) \quad \text{if } m<0 \end{array}$$

2.4 Major Files; Data Provided

The following source files within *S2kit* contain the functions the user will most likely want to use. The test routines discussed previously provide examples of how to use these functions. The source code includes instructions as to how to use the functions, e.g. function arguments.

2.4.1 The Files

We will try to list these files in some logical order, but no promises!

- `primitive.c`: Functions which define the 3-term recurrence coefficients, as well as sample locations and initializing the recurrence, i.e. definition for P_m^m.
- `makeweights.c`: Code which computes the quadrature weights. See [2] for a proof/derivation of the formula, at least for a slightly different grid. Adapting the formula for the grid used in *S2kit* and *SpharmonicKit* is straightforward.
- `plms.c`: Functions for computing the associated Legendre functions via the 3-term recurrence. Used for the naive algorithm.
- `cosplms.c`: Functions for computing the associated Legendre functions **and** their discrete cosine transforms. Used for the semi-naive algorithm.
- `naive_synthesis.c`: Despite its name, this file contains code which implements the forward (analysis) and inverse (synthesis) naive DLT.
- `seminaive.c`: Functions which implement the forward and inverse semi-naive DLT.
- `FST_semi_fly.c`: Functions for computing the forward and inverse spherical harmonic transforms, as well as convolution, assuming the associated Legendre functions are computed on the fly.
- `FST_semi_memo.c`: Functions for computing the forward and inverse spherical harmonic transforms, as well as convolution, assuming the associated Legendre functions are precomputed prior to any transforming.

2.4.2 The Data

Included in the *S2kit* distribution are the following function samples. They can be used to verify that things are working as they should.

- `y20_bw8.dat`: The real and imaginary parts of Y_2^0 sampled on the bandwidth $B = 8$ grid.
- `y31_bw8.dat`: The real and imaginary parts of Y_3^1 sampled on the bandwidth $B = 8$ grid.
- `y43_bw23.dat`: The real and imaginary parts of $(\sqrt{2}+\pi\imath)Y_4^3$ sampled on the bandwidth $B = 23$ grid.
- `yMix_bw17.dat`: The real and imaginary parts of $Y_1^1 + (3-2\imath)Y_5^{-2}$ sampled on the bandwidth $B = 17$ grid.
- `s64.dat`: Real-valued data; samples on the bandwidth $B = 64$ grid. Basically, it's a noisy bump on the sphere.

- `f64.dat`: Real-valued data; samples on the bandwidth $B = 64$ grid. It's a smooth, rotationally symmetric bump centered at the north pole. In the convolution routines, this would be the filter file you would use with the signal file `s64.dat`.
- `s128.dat` and `f128.dat`: Just like `s64.dat` and `f64.dat`, except at bandwidth $B = 128$.

2.5 Memory

In the table below we give approximately how much memory, in megabytes, is required to hold all the associated Legendre functions you need for a **full forward** spherical transform. (This figure does not include other memory, e.g. to hold arrays as "scratch space, etc etc, but it does constitute the bulk of it, especially as the bandwidth grows.) We give figures for both "pure" semi-naive and "pure" naive spherical transforms. A mix of both (e.g. semi-naive DLTs through order m, naive DLTs for remaining orders) will be somewhere in the middle.

If you want to do forward and inverse spherical transforms, precomputing all the Legendres in both cases, multiply by 2.

Bandlimit	Pure Naive	Pure Semi-naive
8	< 1	< 1
16	< 1	< 1
32	< 1	< 1
64	2	< 1
128	16	3
256	128	22
512	1026	171
1024	8200	1368

To provide some comparison between precomputing and computing on the fly, the routine `test_s2_semi_fly` uses a **grand total** of 22 megabytes for a bandwidth $bw = 1024$ transform. However, the transform will take awhile to run. E.g. On our 2.4 Ghz Xeon, the forward transform takes about 40 seconds, and the inverse transform about 45 seconds. Your mileage may vary.

3 Bibliography

Here are the references. Enjoy!

References

[1] G. A. Dilts, Computation of spherical harmonic expansion coefficients via FFTs, *Journal of Computational Physics*, **57**(3) (1985), 439-453.

[2] 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.

[3] *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/`.

[4] D. Healy Jr., D. Rockmore, P. Kostelec and S. Moore, FFTs for the 2-Sphere - Improvements and Variations, *The Journal of Fourier Analysis and Applications*, **9**(4) (2003), p. 341-385. An earlier version of this paper may be downloaded from `www.cs.dartmouth.edu/~geelong/sphere/`.

[5] 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. This paper may also be obtained at `www.cs.dartmouth.edu/~rockmore/dimacs-0.ps`. A pdf version is also available. Simply replace the `.ps` with `.pdf` in the url.

[6] *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/

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